Fully supports Xcode 11, Swift 5.1 and iOS 13

Intermediate iOS 13 Programming with Swift

Take your Swift & iOS programming skills to the next level with these code-rich solutions

SIMON NG

APPCODA

Table of Contents

Preface

Chapter 1 - Building Adaptive User Interfaces

Chapter 2 - Adding Sections and Index list in UITableView

Chapter 3 - Animating Table View Cells

Chapter 4 - Working with JSON and Codable

Chapter 5 - How to Integrate the Twitter and Facebook SDK for Social Sharing

Chapter 6 - Working with Email and Attachments

Chapter 7 - Sending SMS and MMS Using MessageUI Framework

Chapter 8 - How to Get Direction and Draw Route on Maps

Chapter 9 - Search Nearby Points of Interest Using Local Search

Chapter 10 - Audio Recording and Playback

Chapter 11 - Scan QR Code Using AVFoundation Framework

Chapter 12 - Working with URL Schemes

Chapter 13 - Building a Full Screen Camera with Gesture-based Controls

Chapter 14 - Video Capturing and Playback Using AVKit

Chapter 15 - Displaying Banner Ads using Google AdMob

Chapter 16 - Working with Custom Fonts

Chapter 17 - Working with AirDrop, UIActivityViewController and Uniform Type Identifiers

Chapter 18 - Building Grid Layouts with Collection Views

Chapter 19 - Interacting with Collection Views

Chapter 20 - Adaptive Collection Views Using Size Classes and UITraitCollection

Chapter 21 - Building a Today Widget Using App Extensions

Chapter 22 - Building Slide Out Sidebar Menus

Chapter 23 - View Controller Transitions and Animations
Chapter 24 - Building a Slide Down Menu
Chapter 25 - Self Sizing Cells and Dynamic Type
Chapter 26 - XML Parsing, RSS and Expandable Table View Cells
Chapter 27 - Applying a Blurred Background Using UIVisualEffect
Chapter 28 - Using Touch ID and Face ID For Authentication
Chapter 29 - Building a Carousel-Like User Interface
Chapter 30 - Working with Parse
Chapter 31 - Parsing CSV and Preloading a SQLite Database Using Core Data
Chapter 32 - Connecting Multiple Annotations with Polylines and Routes
Chapter 33 - Using CocoaPods in Swift Projects
Chapter 34 - Building a Simple Sticker App
Chapter 35 - Building iMessage Apps Using Messages Framework
Chapter 36 - Building Custom UI Components Using IBDesignable and IBInspectable
Chapter 37- Using Firebase for User Authentication
Chapter 38 - Google and Facebook Authentication Using Firebase
Chapter 39 - Using Firebase Database and Storage to Build an Instagram-like App
Chapter 40 - Building a Real-time Image Recognition App Using Core ML
Chapter 41 - Building AR Apps with ARKit and SpriteKit
Chapter 42 - Working with 3D Objects in Augmented Reality Using ARKit and SceneKit
Chapter 43 - Using Create ML to Train Your Own Machine Learning Model for Image Recognition
Chapter 44 - Building a Sentiment Classifier Using Create ML to Classify User Reviews
Chapter 45 - Working with Image Tracking in ARKit

Copyright ©2020 by AppCoda Limited

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

Published by AppCoda Limited

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

Update History

Release Date	Description
21 Jan, 2018	Updated all chapters of the book for Swift 4 and Xcode 9.
20 Mar, 2018	Added two new chapters for ARKit
16 Apr, 2018	Added a new chapter for Core ML
25 Sep, 2018	Updated for iOS 12 and Swift 4
31 Jan, 2020	Updated for iOS 13 and Swift 5.1

Fully supports Xcode 11, Swift 5.1 and iOS 13
Intermediate
with
iOS 13 Programming
Swift
Take your Swift & iOS programming skills to the next level with these code-rich solutions
SIMON NG
APPCODA

Preface

At the time of this writing, the Swift programming language has been around for more than four years. 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. On top of that, they provide clear and easy-to-follow instructions to prepare a dish. 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 13 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, train a machine learning model, interact with virtual objects with ARKit, 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 dives 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 https://www.appcoda.com/swift.

What version of Xcode do you need?

Most of the chapters have been updated for iOS 13, Xcode 11, and Swift 5.1. Therefore, make sure you use Xcode 11 (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 11.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 11 is released (e.g. Xcode 11.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 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 landscape). 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 2020: Apple's iPhone and iPad lineup has changed a lot. With the launch of the iPhone 11, 11 Pro, and 11 Pro Max, 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/7/8 (4.7-inch)
- iPhone 6/6s/7/8 Plus (5.5-inch)
- iPhone X/XS/11 Pro (5.8-inch)
- iPhone XR/11 (6.1-inch)
- iPhone XS Max/11 Pro Max (6.5-inch)
- iPad (9.7-inch)
- iPad Mini (7.9-inch)
- iPad Pro (10.5/12.9-inch)

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*. Starting from Xcode 7, the development tool allows developers to build an app UI that adapts to all different devices, screen sizes and orientation using Interface Builder. From Xcode 8, the Interface Builder is further re-engineered to streamline the making of an adaptive user interface. It even comes with a full live preview of how things will render on any iOS device. You will understand what I mean when we check out the new Interface Builder.

To achieve adaptive layout, you will need to make use of a concept called *Size Classes*, which is available on iOS 8 or up. 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, 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.

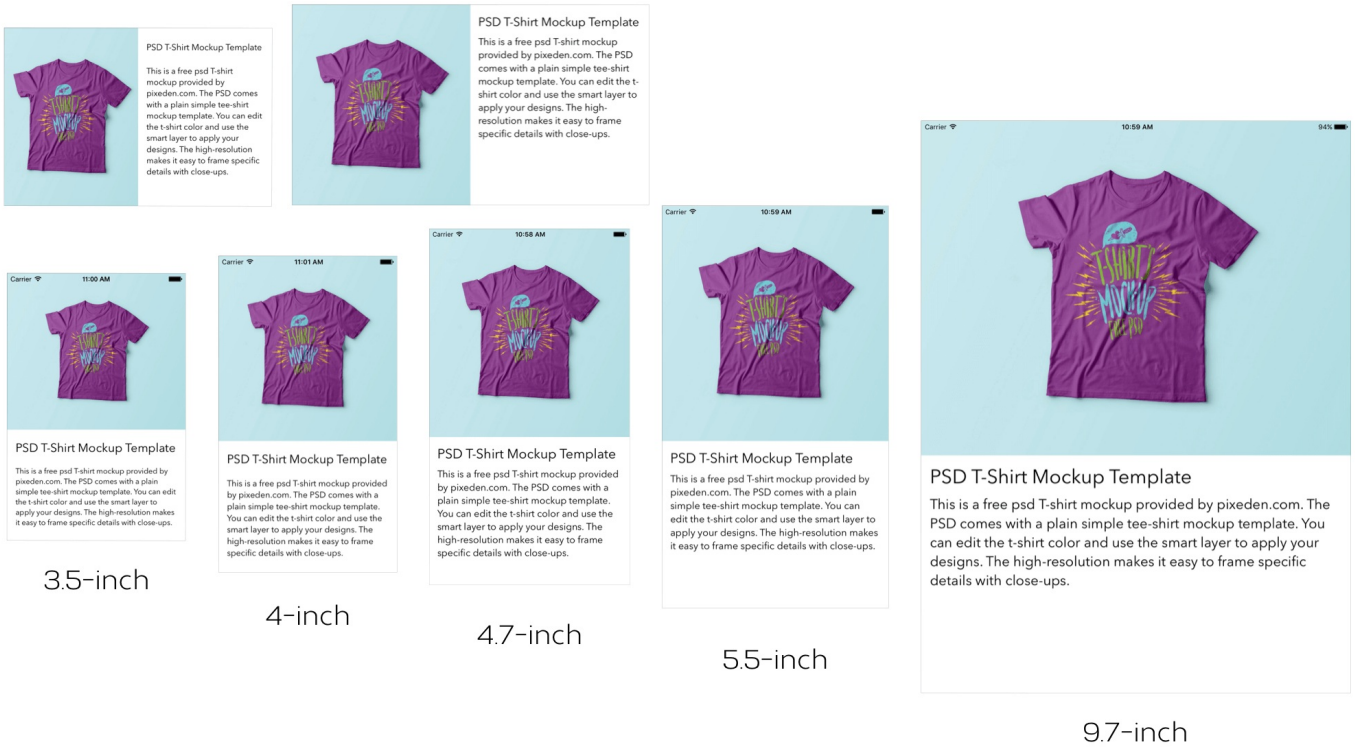


Figure 1.1. Adaptive UI demo

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 *Storyboard* for the User Interface option.

Once the project is created, go to the project setting and set the *Deployment Target* from 13.0 to 9.3 (or lower). This allows you to test your app on iPhone 4s because iOS 10 (or up) no longer supports the 3.5-inch devices. You probably don't want to support the older generation of devices too, but in this demo, I would like to demonstrate how to build an adaptive UI for all screen sizes.

Now, open `Main.storyboard`. In Interface Builder, you should find a view controller that has the size of an iPhone 8. The Interface Builder lets you lay out the user interface in a simulated device and you can easily switch between different devices using the device

configuration controls in the bottom bar. For example, select *View as* to open the device configuration pane, and then select iPhone 8 Plus. This changes the size of the view controller in the storyboard to the selected device.

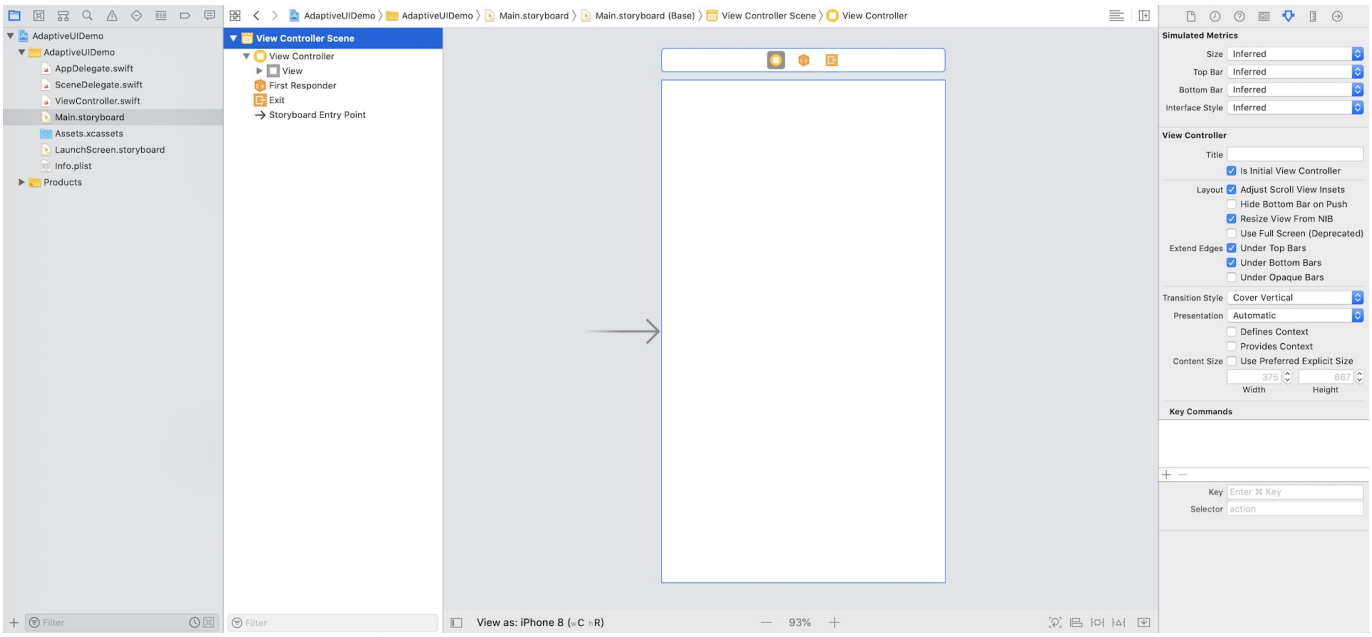


Figure 1.2. Interface Builder in Xcode

Now we'll start to design the app UI. First, download the image pack from http://www.appcoda.com/resources/swift4/adaptiveui-images.zip and import the images to `Assets.xcassets`.

Next, go back to the storyboard. I usually start with iPhone 8 (4.7-inch) to lay out the user interface and then add layout specializations for other screen sizes. Therefore, if you have chosen other device (e.g. iPhone 11), I suggest you change the device setting to iPhone 8.

Now, drag an image view from the Object library to the view controller. Set its width to `375` 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 the later section. In Size inspector, make sure you set the width to `375` and height to `277`.

Throughout the chapter, I will refer to this view as *Product Info View*. Your layout should be similar to the figure below.

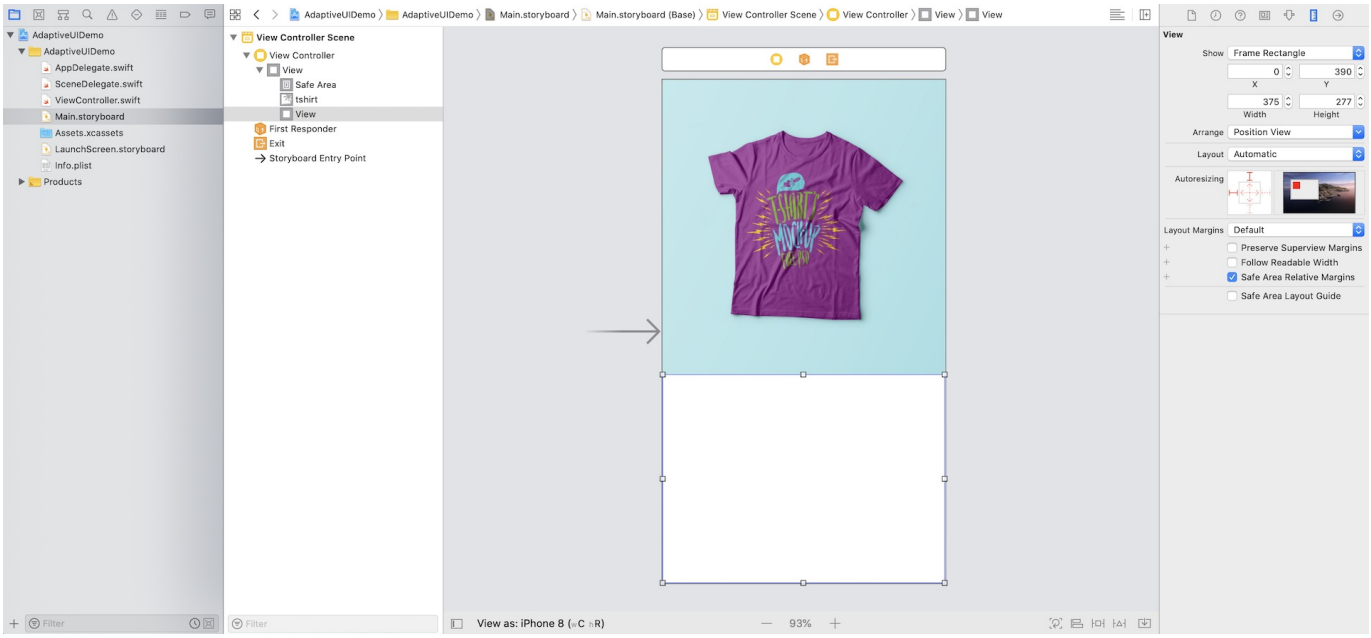


Figure 1.3. Adding the Product Info View to the view controller

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 `25` points. Press `command+=` to resize the label and make it fit. In the Size inspector, change the value of X to `15` and Y to `15`.

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 `15` and Y to `58`. Set the width to `352` and height to `182`.

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:

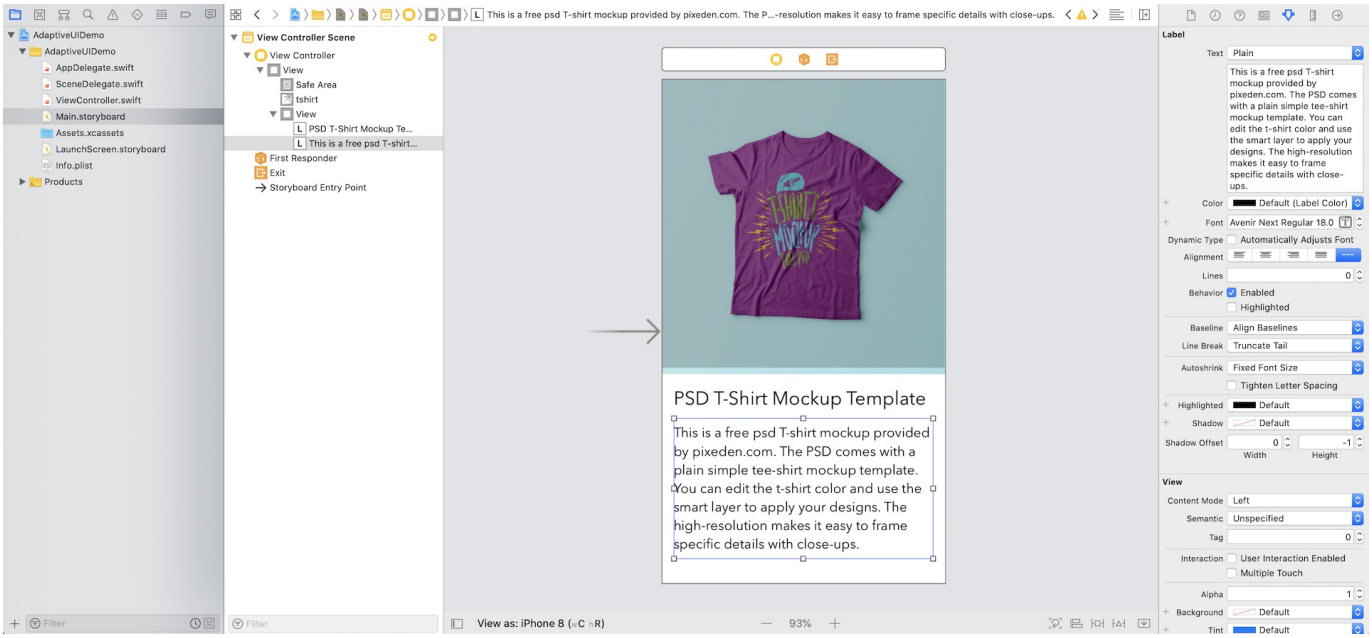


Figure 1.4. The sample app UI

Even if your design does not match the reference design perfectly, it is absolutely fine. We will use auto layout constraints to lay out the view later.

Now, the app UI is perfect for iPhone 8 or 4.7-inch screen. Let's conduct a quick test to check out the look and feel of the design on different devices.

In Xcode, you have two ways to live preview the app UI:

1. By using device configuration pane
2. By using the Preview Assistant

As you have tried out earlier, you can click *View as* button to switch over to another device. Now try to change the device to iPhone SE/11 to see how it looks. It doesn't look good. The image and the text are both truncated. You can continue to switch over to other devices to preview the UI.

Alternatively, you can use the Preview Assistant, which lets you evaluate the resulting design on different size displays at one time.

To bring up the preview assistant, click the *Adjust Editor Options* button and choose *Preview*.

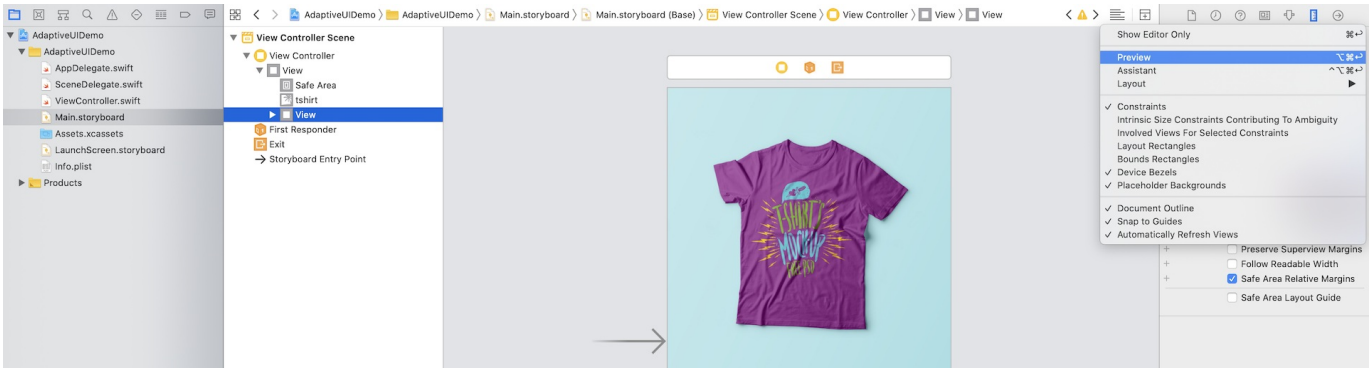


Figure 1.5. Opening the preview assistant

Xcode will then display a preview of the app's UI in the assistant editor. By default, it shows you the preview of your selected iOS device. You can click the + button at the lower-left corner of the assistant editor to get a preview of an iPhone 8 Plus 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 all devices except iPhone 8. So far we haven't defined any auto layout constraints. This is why the view doesn't fit properly on all devices.

Figure 1.6. Demo App UI on different 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 55-60% 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 0.585. (I calculated this value beforehand, but any value between 0.55 and 0.6 should work.)
- 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 *Add new constraints* 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.

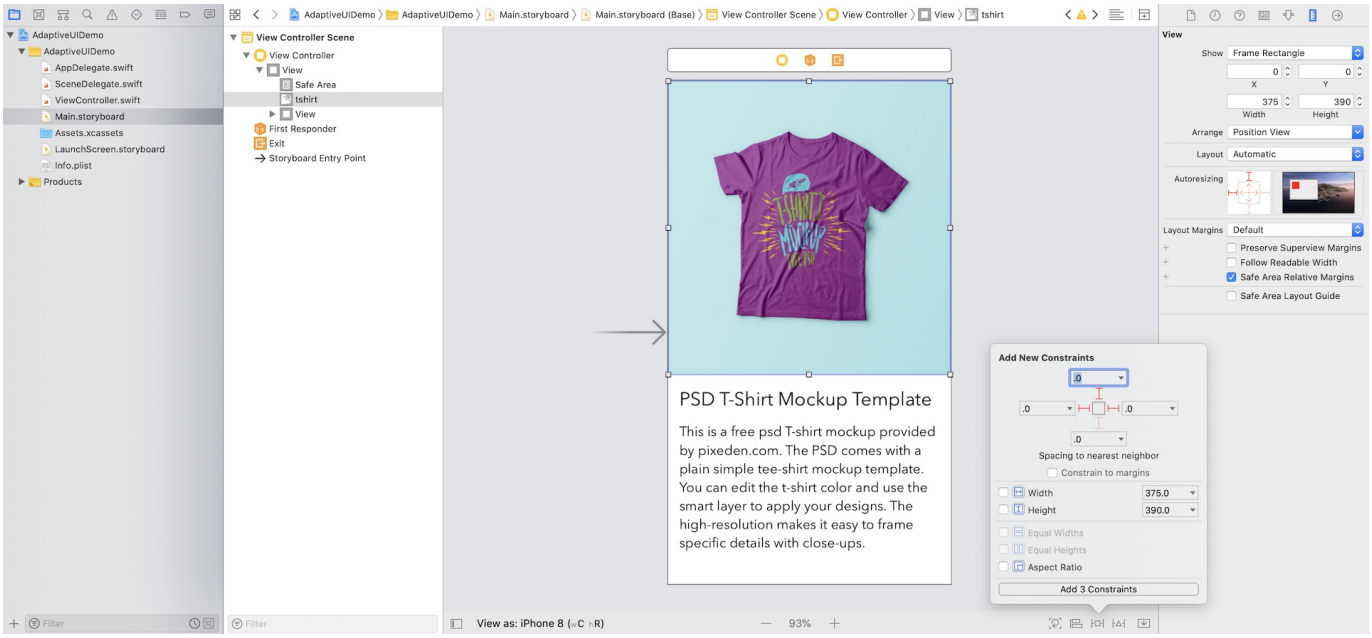


Figure 1.7. Adding spacing constraints for the image view

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.

Figure 1.8. Control-drag from the image view to the main view

Once you added the Equal Heights constraint, the constraint 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.

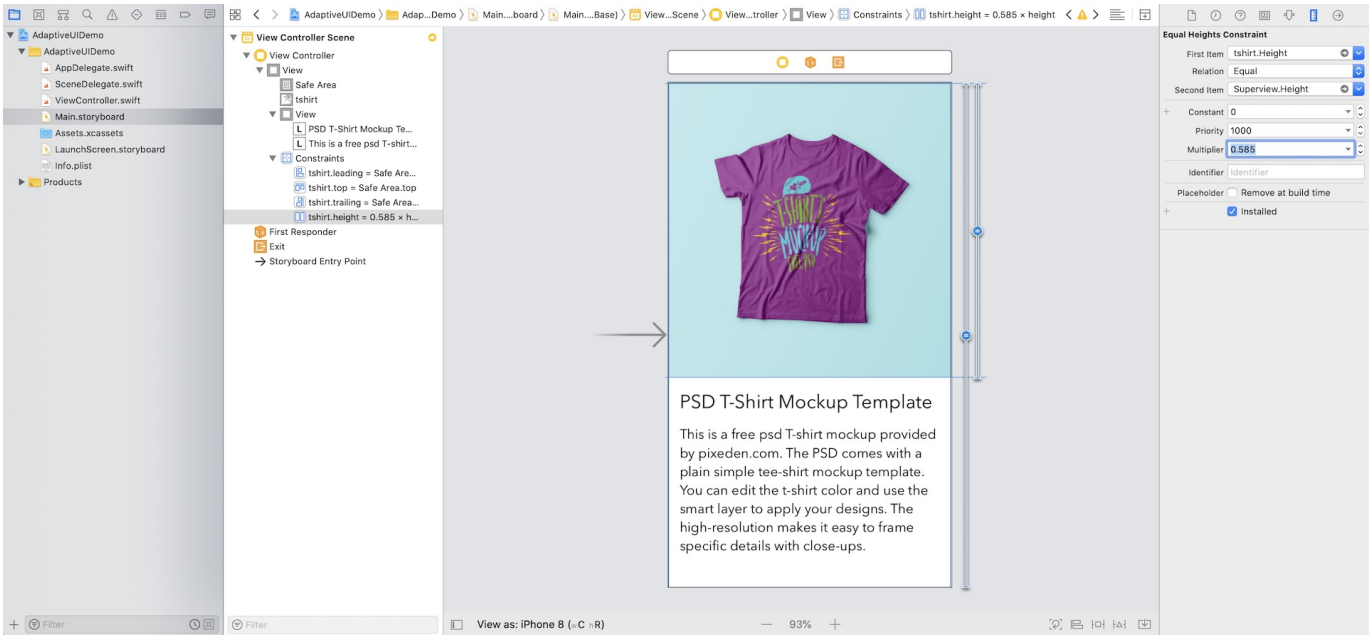


Figure 1.9. Editing the Equal Heights Constraints

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 60% of the main view. So change the multiplier from `1` to `0.585`.

Next, select *Product Info View* and click the *Add new constraints* 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.

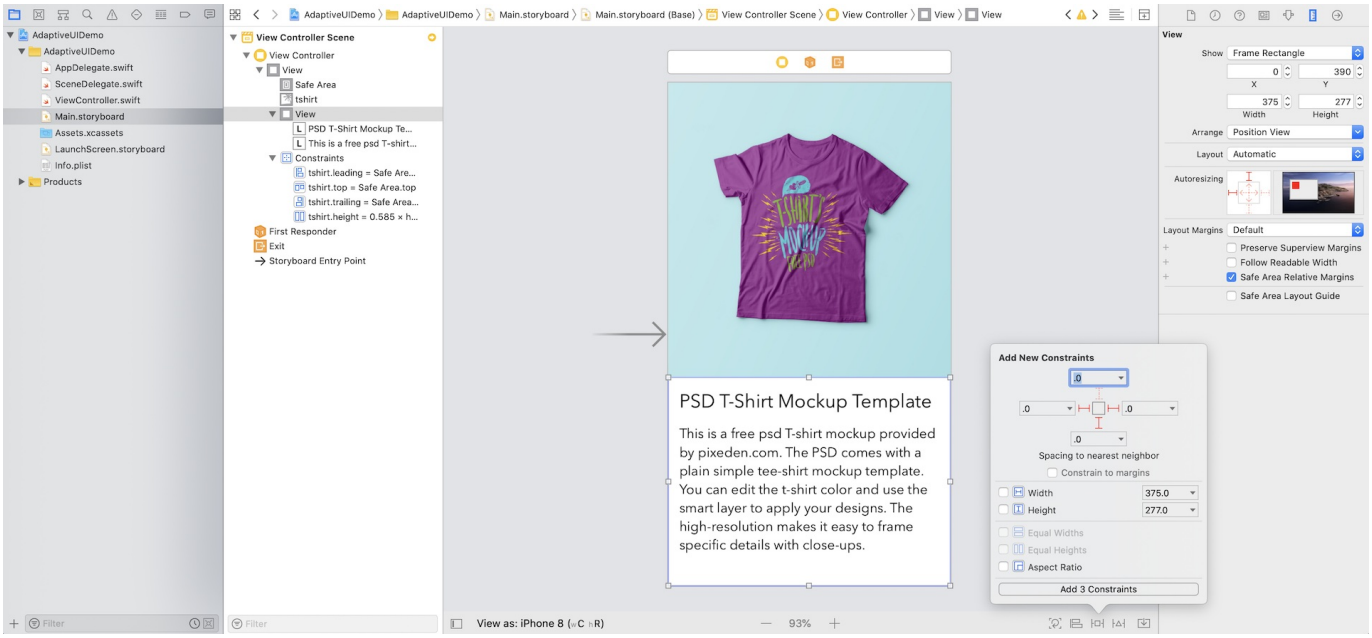


Figure 1.10. Defining the spacing constraints for the product info view

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.

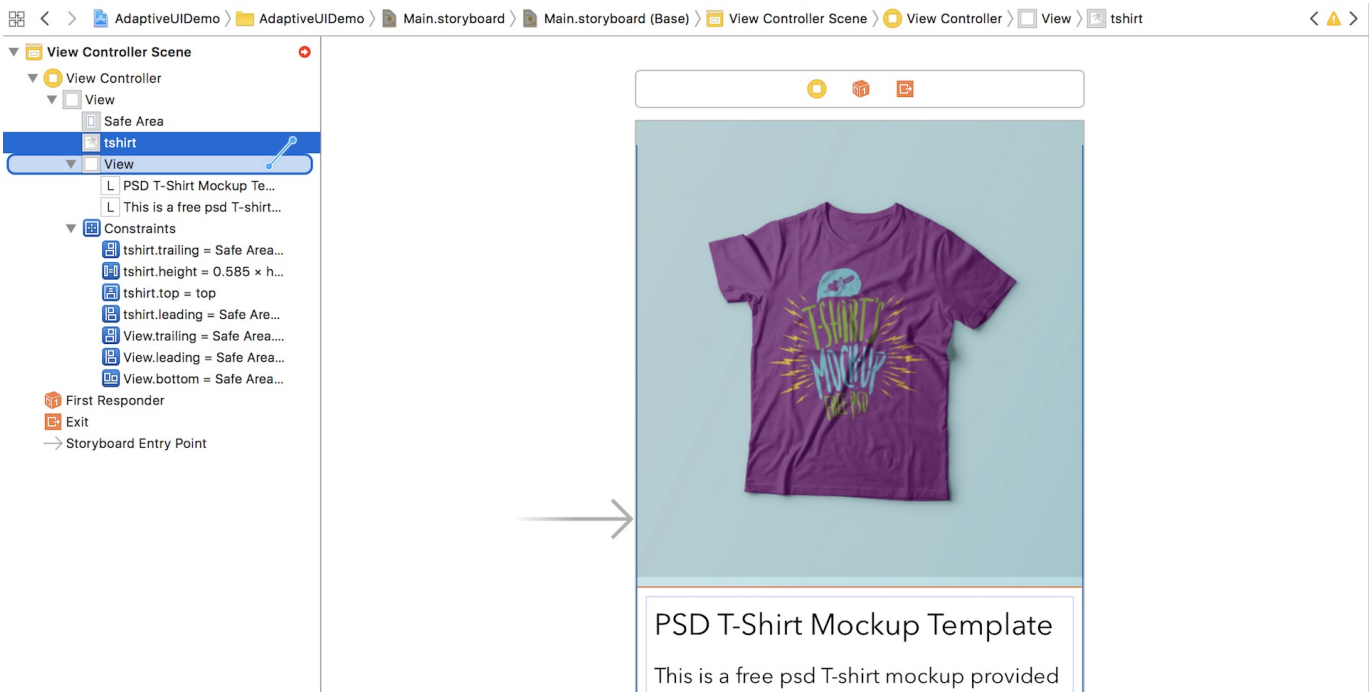


Figure 1.11. Adding a spacing constraint for the image view and main view using control-drag

If you take a look at the views rendered on other devices, the image view should now fit for all devices; however, there is still a lot of work to do for the labels.

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

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

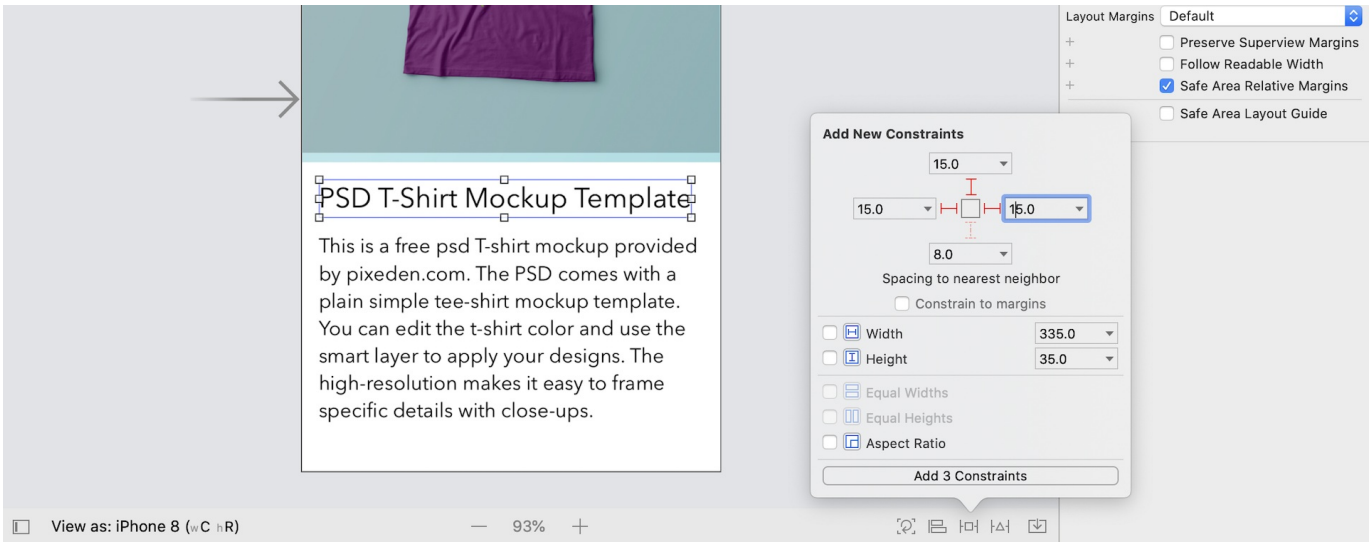


Figure 1.12. Defining the constraints for the title label

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

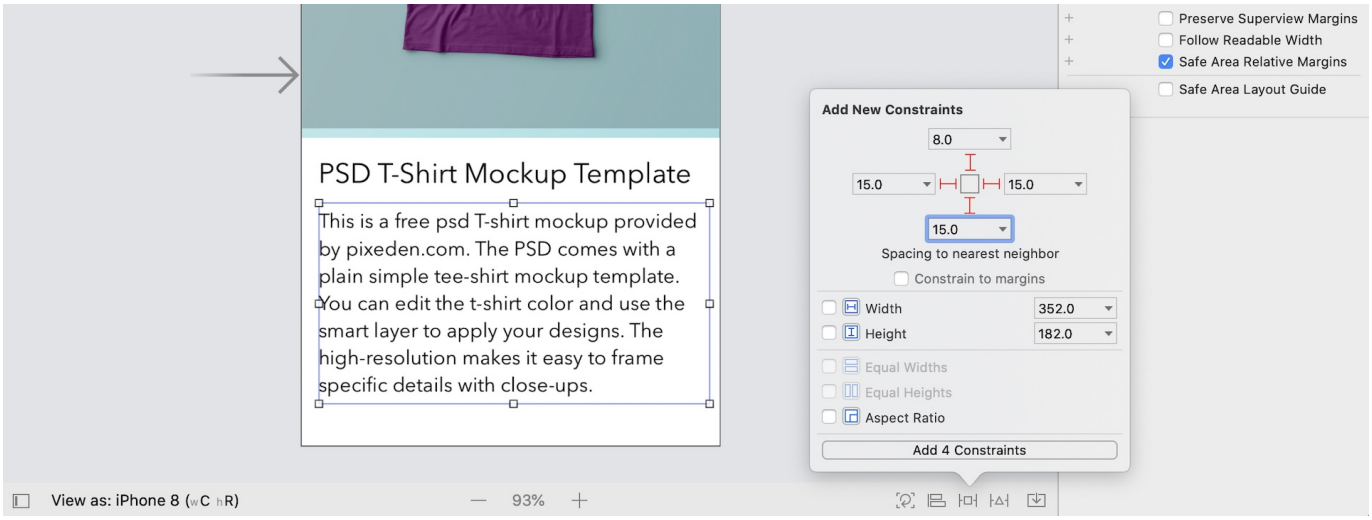


Figure 1.13. Adding spacing constraints for the description label

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.

Figure 1.14. The red disclosure arrow

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

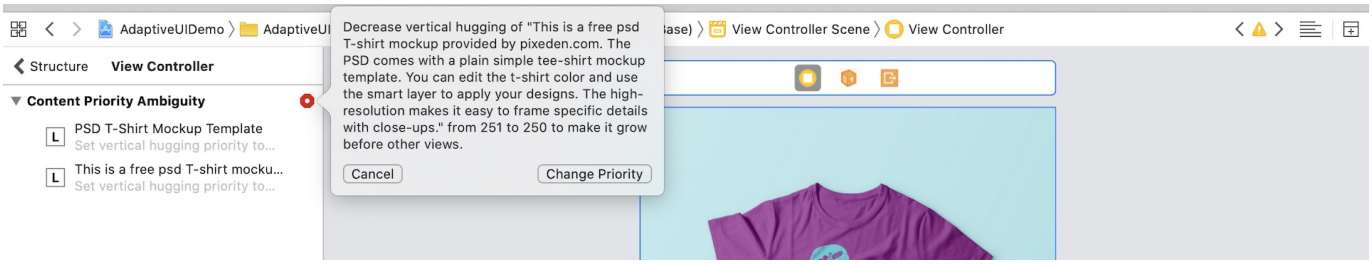
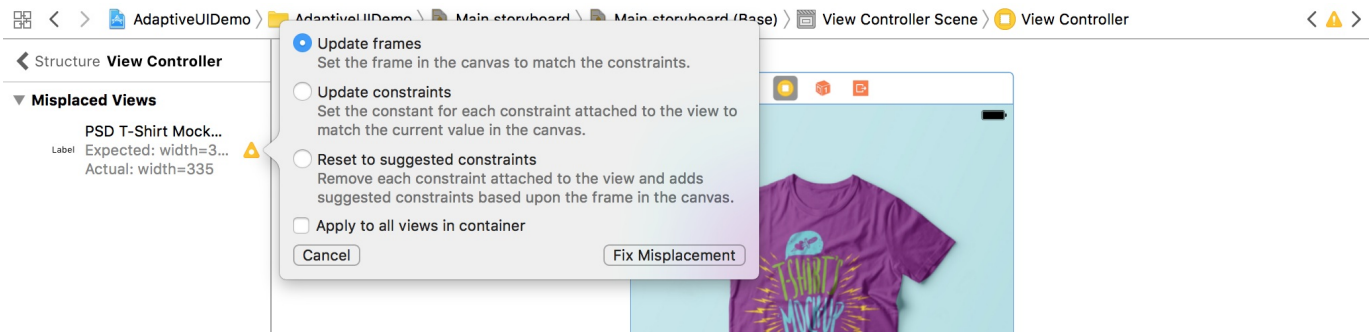


Figure 1.15. Resolving the layout issues

Sometimes, you may see the yellow indicator. This indicates that there are some misplacements of the views. Again, you can let Interface Builder fix the issues for you by updating the frames to match the constraints.



Cool! You've created all the auto layout constraints. You can check out the preview and see how the UI looks on various devices.

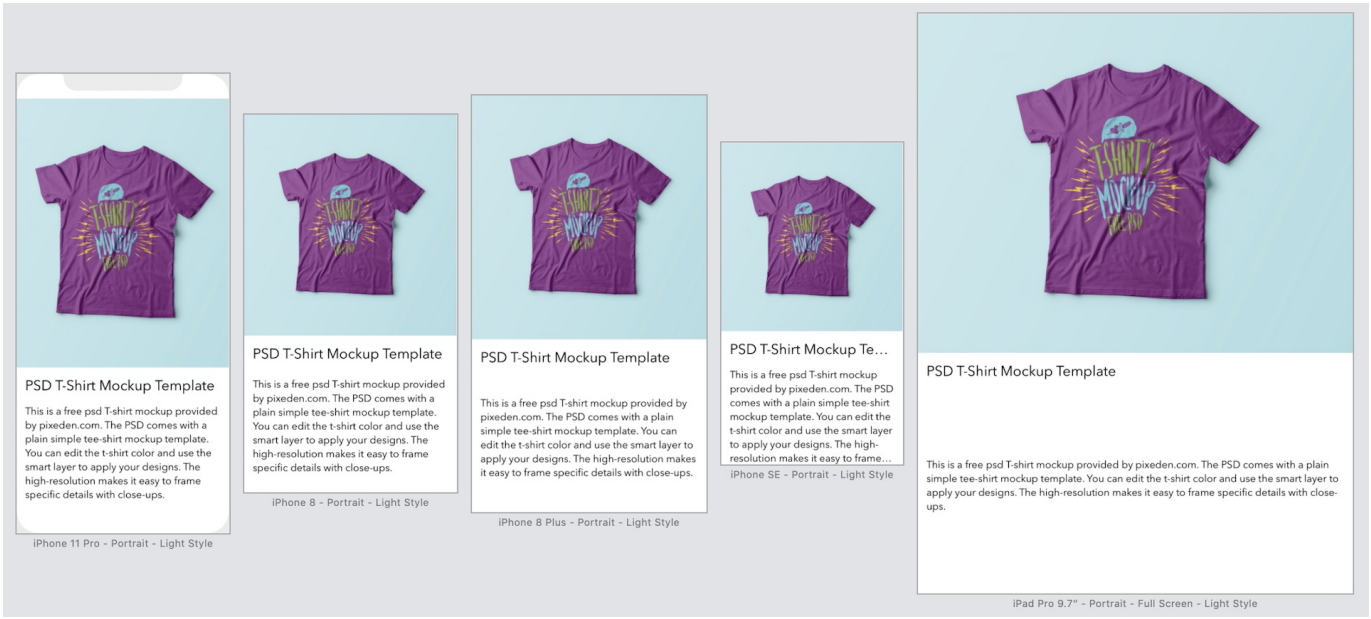


Figure 1.17. The App UI after adding layout constraints

The view looks much better now, with the image view perfectly displayed and aligned. However, there are still a couple of issues:

- The description label is vertically centered on devices with larger screen size. We want to it to be displayed right below the title label.

- The title and description labels are partially displayed for some of the iPhone models.

Let's take a look at the first issue. Do you remember that we defined a couple of vertical space constraints for the description label? The constraint said that the description label should be 8 points away from the title label and 15 points away from the bottom of the super view (refer to figure 1.13). In order to satisfy the constraints, iOS has to expand the description label on larger screen size. Thus, the description stays vertically centered.

Content Hugging Priority

Let me side track a bit and ask you a question. Why doesn't iOS expand the title label instead of the description label? Take a look at figure 1.18. There are actually two options for iOS to render the UI that satisfies the layout constraints. How does iOS come up with that choice?

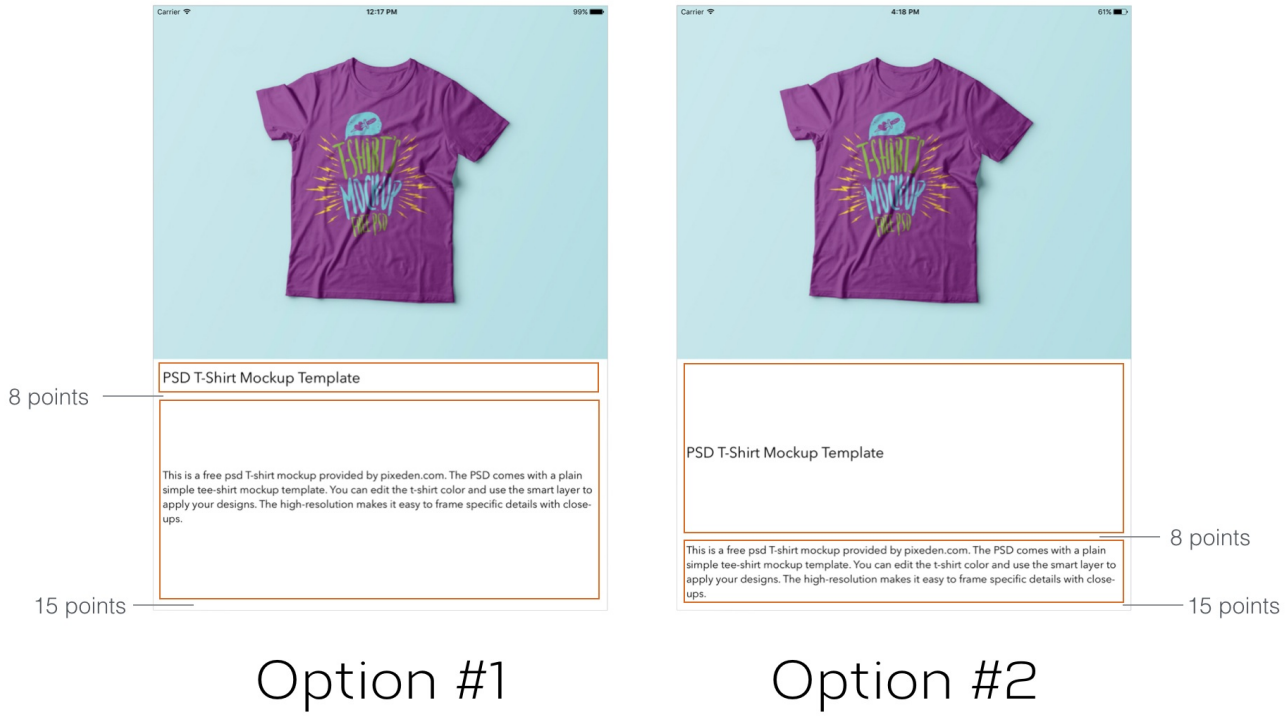


Figure 1.18. Two options for rendering the title and description labels on devices with bigger screen size

If you select the description label and go to the Size inspector, you should notice that the content hugging priority (vertical) is set to `250`. Now select the title label and check out its content hugging priority. You should notice that it is set to `251`. In other words, it has a higher content hugging priority (for the vertical axis) than the description label.

The value of content hugging priority helps iOS determine which view it should enlarge. The view with a higher hugging priority can resist from growing larger than its intrinsic size. Here, the title label has a higher hugging priority. This is why iOS chooses to make the description label larger, while the size of the description label is unchanged.

Editing the Relation of the Constraints

Now that you should have a basic understanding of the content hugging priority, let's continue to fix the first issue we have discovered.

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.

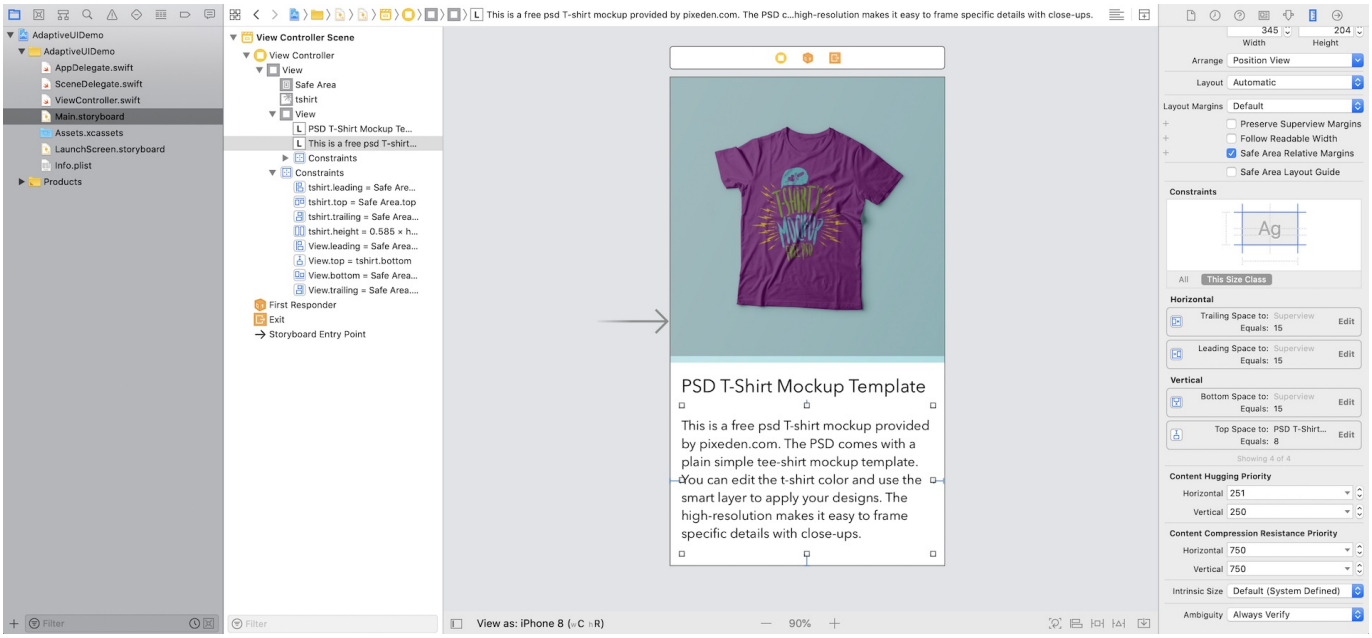


Figure 1.19. Review the layout constraints of the description label in the Size Inspector

We have defined four spacing constraints for the label. If you look into the constraints, each of them has a relation *Equal*. This means each side of the label should have the exact same space as we specify in the constraints, when rendering the description label. The space can't be bigger or smaller.

So how can we modify the constraint so that the description label is placed under the title label, regardless of the screen size?

I guess you may know the answer. Instead of strictly set the constraint relation to *Equal*, the bottom space constraint should have some more flexibility. The space is not required to equal `15 points`. This is just the minimum space we want. The space can actually grow with the screen size.

Now double click the Bottom Space constraint to edit it. Change the relation from `Equal` to `Greater than or Equal`. Once the change is made, the space issue should be fixed.

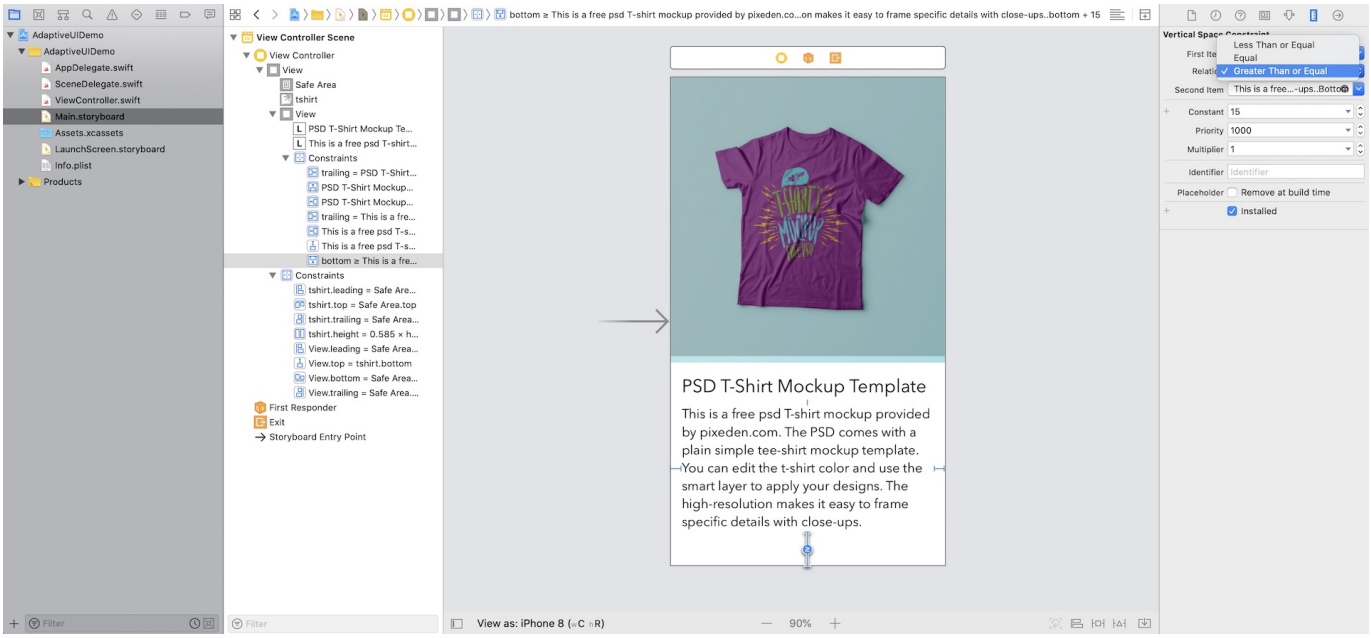


Figure 1.20. Changing the relation of the bottom space constraint

Okay, it's time to look into the second layout issue:

- The title and description labels are partially displayed for some of the iPhone models.

This issue is pretty easy to fix. We can just let iOS auto shrink the font size for devices with smaller screen sizes. Select the title label and go to the Attributes inspector. Set the value of `Autoshrink` option to `Minimium Font Size`. Repeat the same procedures for the description label.

That's it. If you preview the UI on iPhone SE or execute the project on these devices, both labels are displayed properly.

Size Classes

As I mentioned at the very beginning of the chapter, designing adaptive UI is a two-part process. So far we have created 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.

First, what is a size class?

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: *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 results 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.

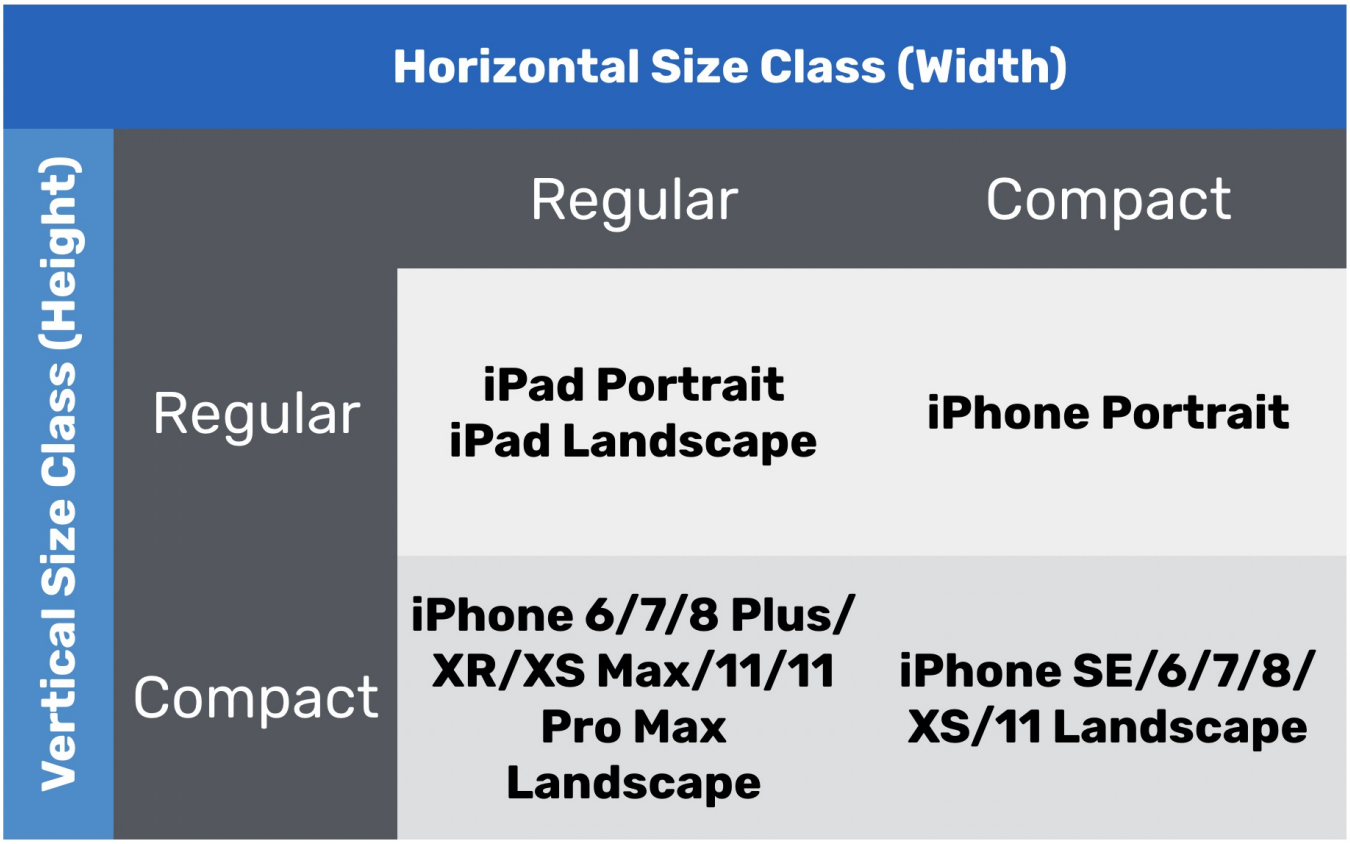


Figure 1.21. Size Classes

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 SE as it does on an iPhone 8.

The iPhone 6/7/8 Plus/11/11 Pro Max, 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 8 (or lower).

Using Size Classes for Font Customization

With some basic understanding of size classes, let's see how we use it to apply layout specialization.

Needless to say, we want to make the title and description labels perfect for iPhones. The current font size is ideal for the iPhones but too small for iPad. What we're going to do is make the font a bit larger for the iPad devices.

With size classes, you can now adjust the font style for a particular screen size. In this case, we want to change the font size for all iPad models. In terms of size classes, the iPad device defaults to *regular* size class for horizontal (width) and *regular* size class for vertical (height).

To set the font size for this particular size class, switch to the iPad device in Interface Builder and select the title label. Under the Attributes inspector, you should see a plus (+) button next to the font field. Click the + button. Make both width and height are set to *Regular*, and then click `Add Variation`.

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 `wR hR` font field to `35` points.

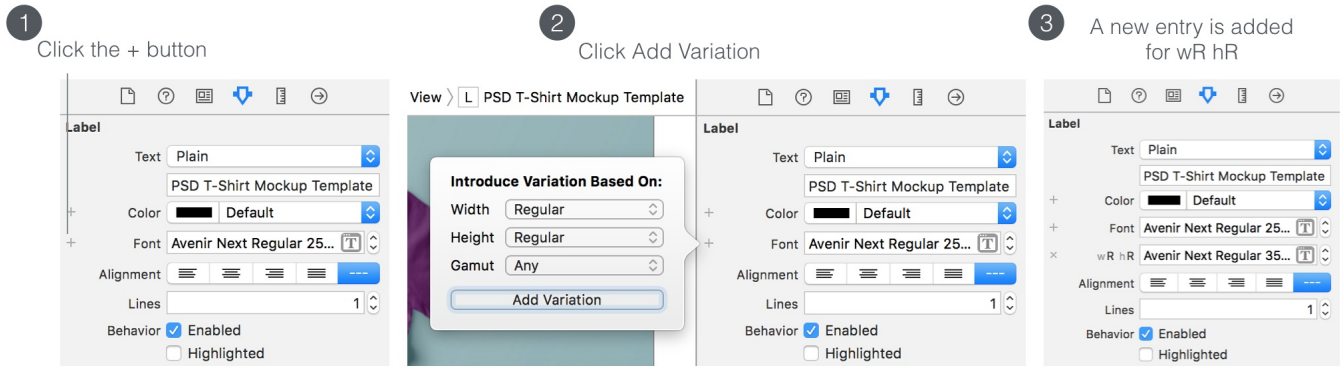


Figure 1.22. Setting the font style for regular-regular size class

This will instruct iOS to use the second font with a larger font size on iPad. For the iPhone devices, 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 click `Add Variation`. Change the size of `wR hR` font field to `25` points. Look at the preview or test the app in simulators. The layout looks perfect on all screen sizes.

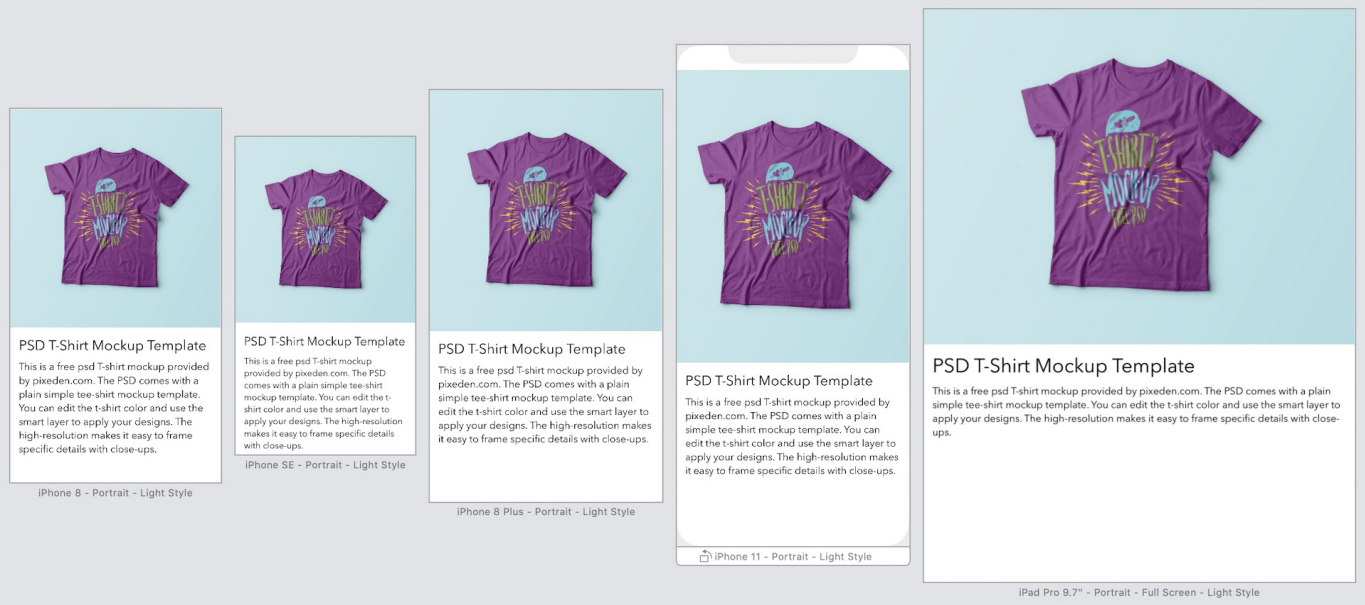


Figure 1.23. Revised App UI after using size classes

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 8). Or you can check out the UI in landscape mode using the simulator. The view looks okay but I think there is a better way to lay out the UI in the landscape orientation.

Figure 1.24. App UI in landscape orientation

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.

Figure 1.25. Redesigned App UI in landscape orientation

So how can we use size classes to create two different UIs? Currently, we only have a single set of auto layout constraints that apply to all size classes. In order to create two different UIs, we will have to use two different sets of layout constraints for each of the UIs:

- For iPad and iPhone (Portrait), we utilize the existing layout and layout constraints.
- For iPhone (Landscape), we re-layout the UI and define a new set of layout constraints.

First, we have to move the existing layout constraints to a size class such that the constraints are activated when the device is an iPad or iPhone in portrait orientation. In the device configuration pane, you should find the `Vary for Traits` button, which is designed for creating user interface variation. When you click the button, a popover appears with two options for you to choose. In this case, select `height` and click anywhere in the Interface Builder. The device configuration pane turns blue and shows

you the affected devices for the size class we just selected. If you click the `Varying 98 Regular Height Devices` option, it will reveal all the affected devices including iPad and iPhone (Portrait).

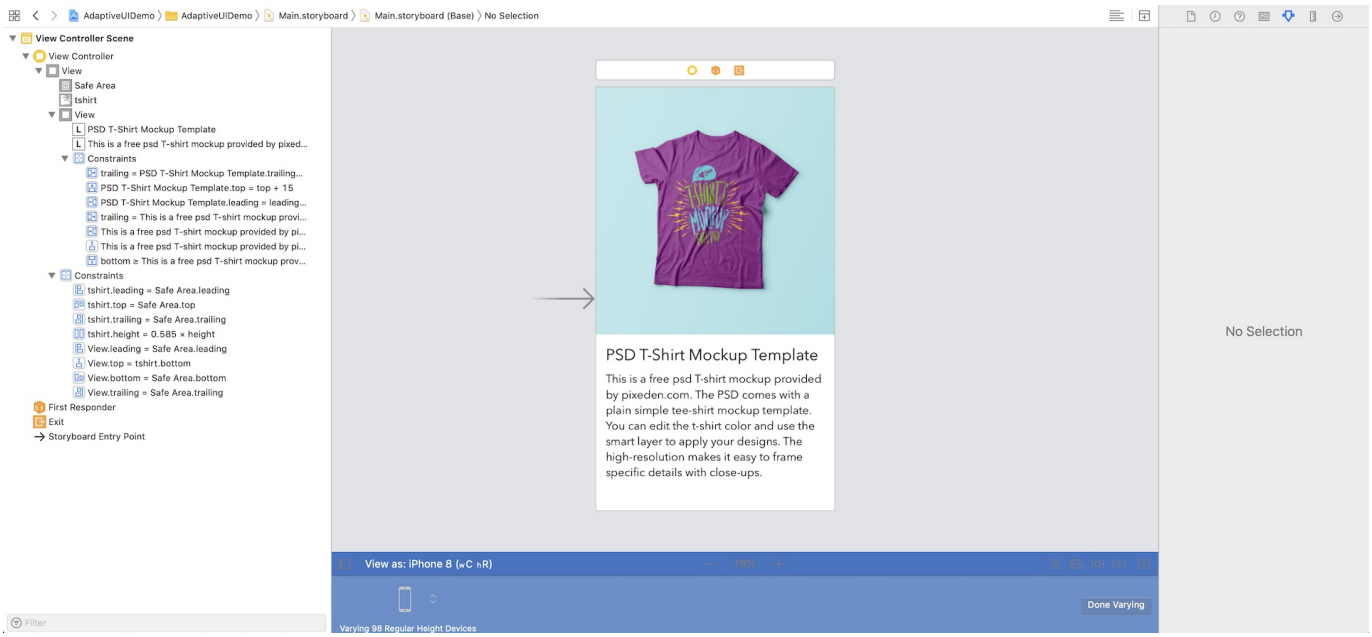


Figure 1.26. Creating user interface variation

While in the variation mode, any changes you made to the canvas will apply to the current variation (or size class) only. As we want to migrate all existing constraints to this variation, select all constraints in the document outline view (hold the *command* key and select each of the constraints). Next, go to the Size inspector and click the + button (next to the installed option) to create a variation.

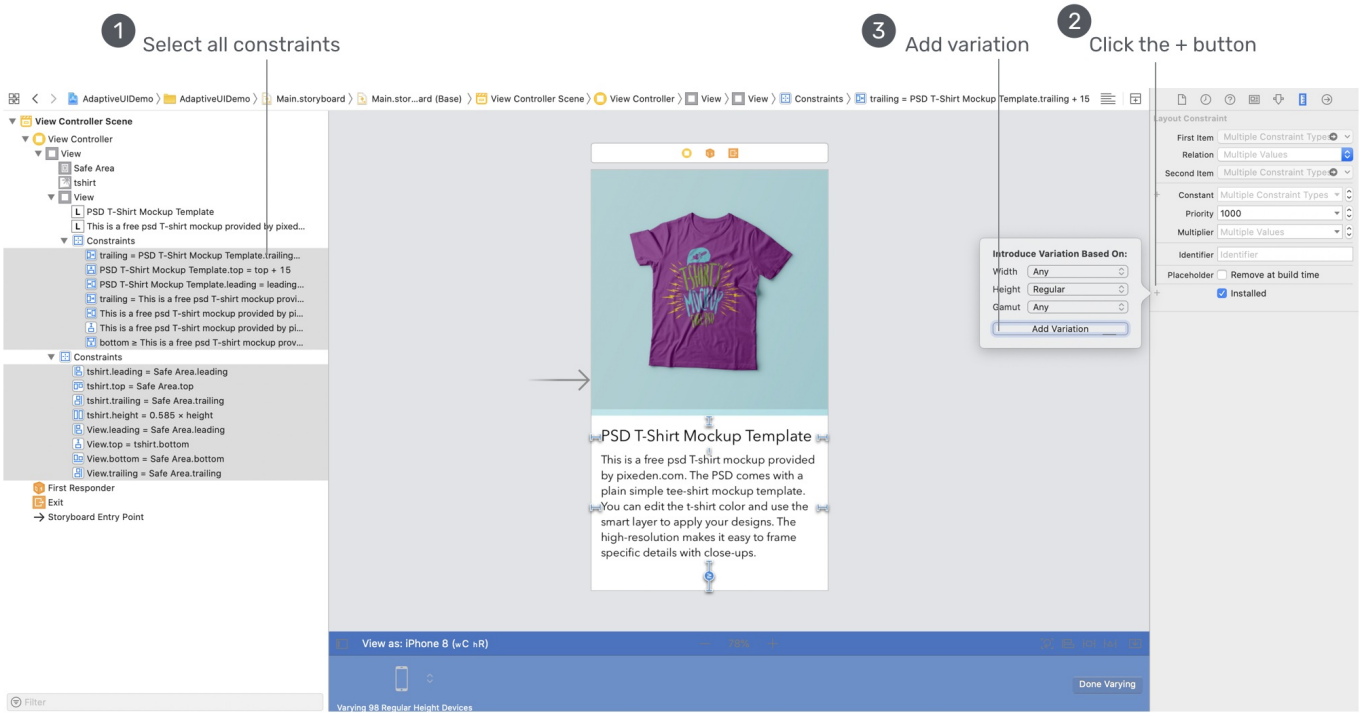


Figure 1.27. Add variation for the selected constraints

Interface Builder then shows you an `Installed` checkbox for the *regular-height* size class. Because all existing constraints should be applied to this size class only. Uncheck the `Installed` checkbox and check the `hR Installed` checkbox. This means all the selected constraints are activated for the iPad and iPhone (Portrait) devices. Lastly, click `Done Varying` to complete the changes.

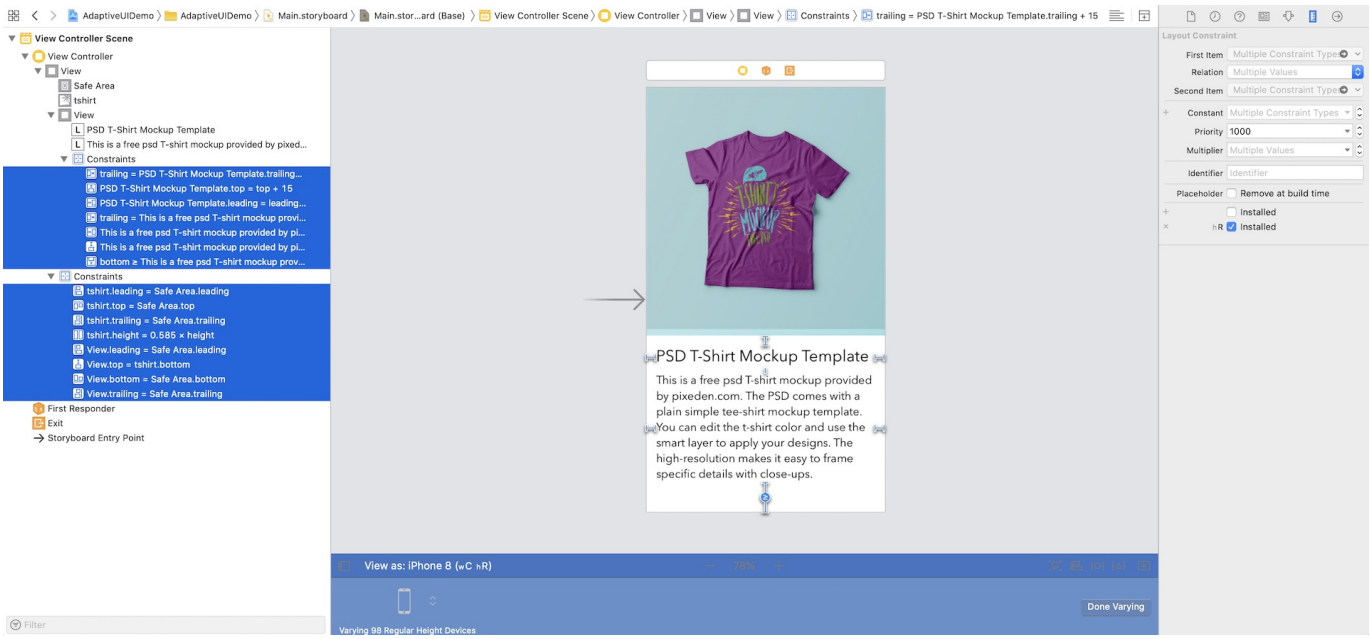


Figure 1.28. Install the selected constraints for the regular-height size class

How do you know if the constraints are applied to the *regular-height* device only? In the device configuration pane, you can change the orientation of the iPhone to landscape. You should find that the UI in landscape is no longer rendered properly. And, all the constraints are grayed out, which means they do not belong to this size class.

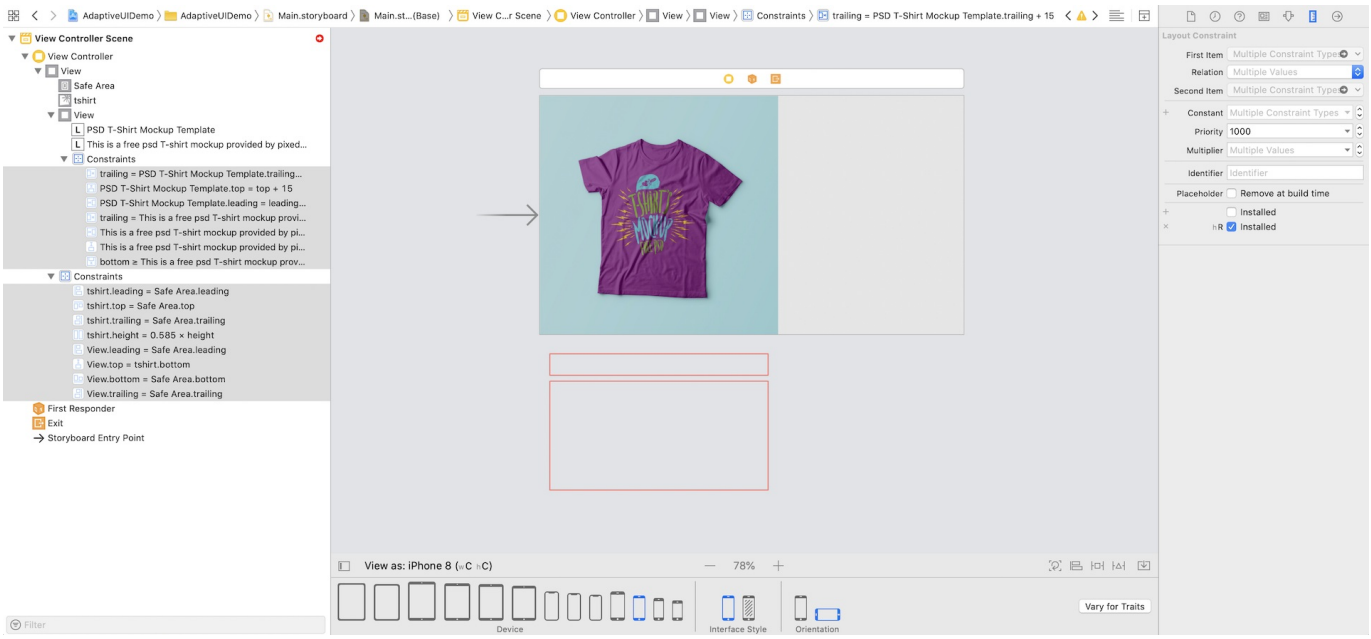


Figure 1.29. Layout constraints are deactivated for iPhone in landscape orientation

Now it's time to redesign the app layout in landscape orientation and define a separate set of layout constraints.

Make sure you select the iPhone 8 device and landscape orientation in the device configuration bar. Again, click the `Vary for Traits` button. In the popover, select Height to create a variation for all iPhone devices in landscape mode.

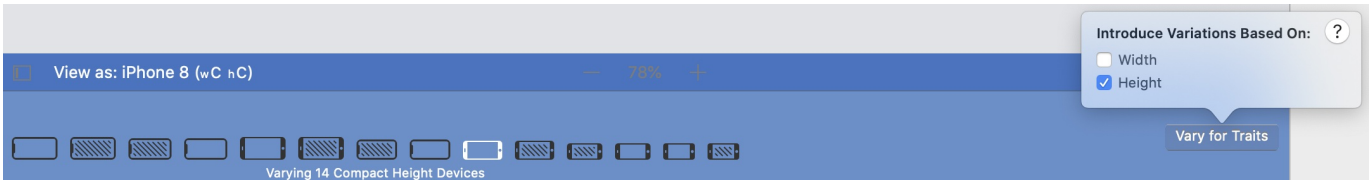


Figure 1.30. Creating a user interface variation

From now on, all the changes you're going to make will apply to the selected size class only (i.e. compact-width and compact height).

First, select the *t-shirt* image view. In the Size inspector, set x to `0`, y to `0`, width to `333` and height to `375`. In the Attributes inspector, make sure the `Clip to Bounds` option is enabled.

Next, select the view that holds the title and description label. Go to the Size inspector, set x to `333`, y to `0`, width to `334` and height to `375`.

Lastly, resize the title and description labels to make them fit. Here I change the width of both labels to `303` points. Your layout should look similar to figure 1.31.

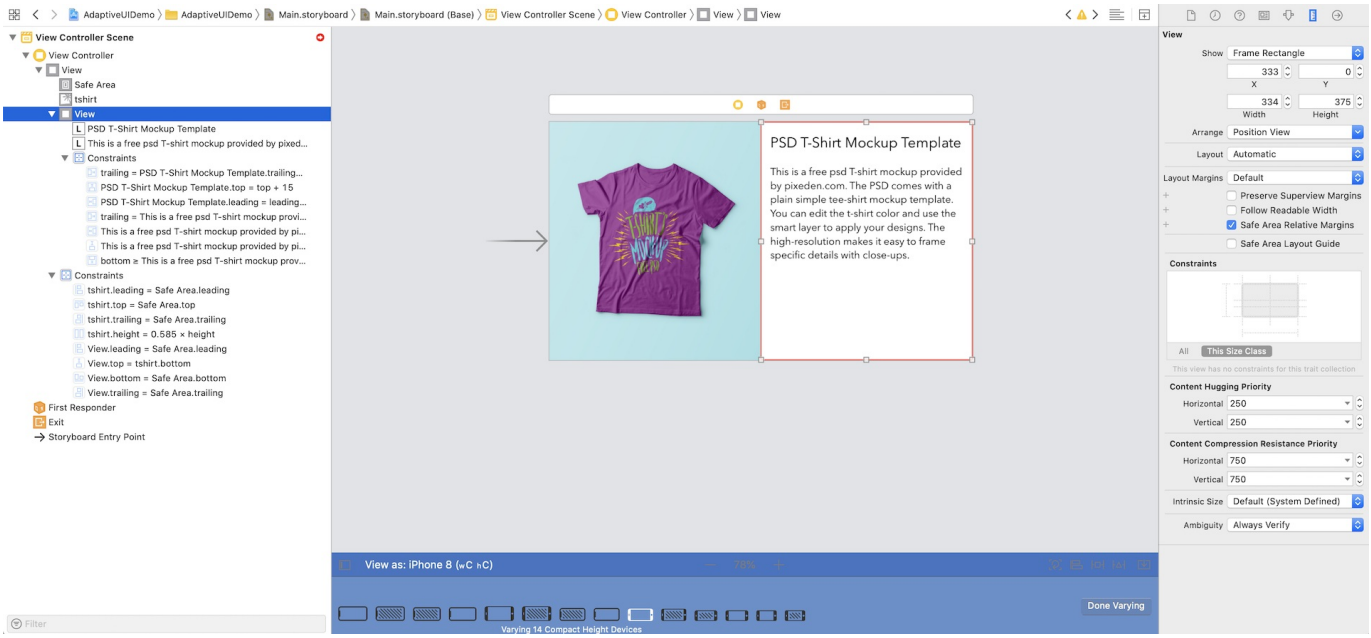


Figure 1.31. Redesigning UI for the iPhone devices in landscape mode

So far we haven't defined any layout constraints for this size class. Now select the *t-shirt* image view and click the *Add new constraints* button. Add three space constraints for the top, bottom, and left sides.

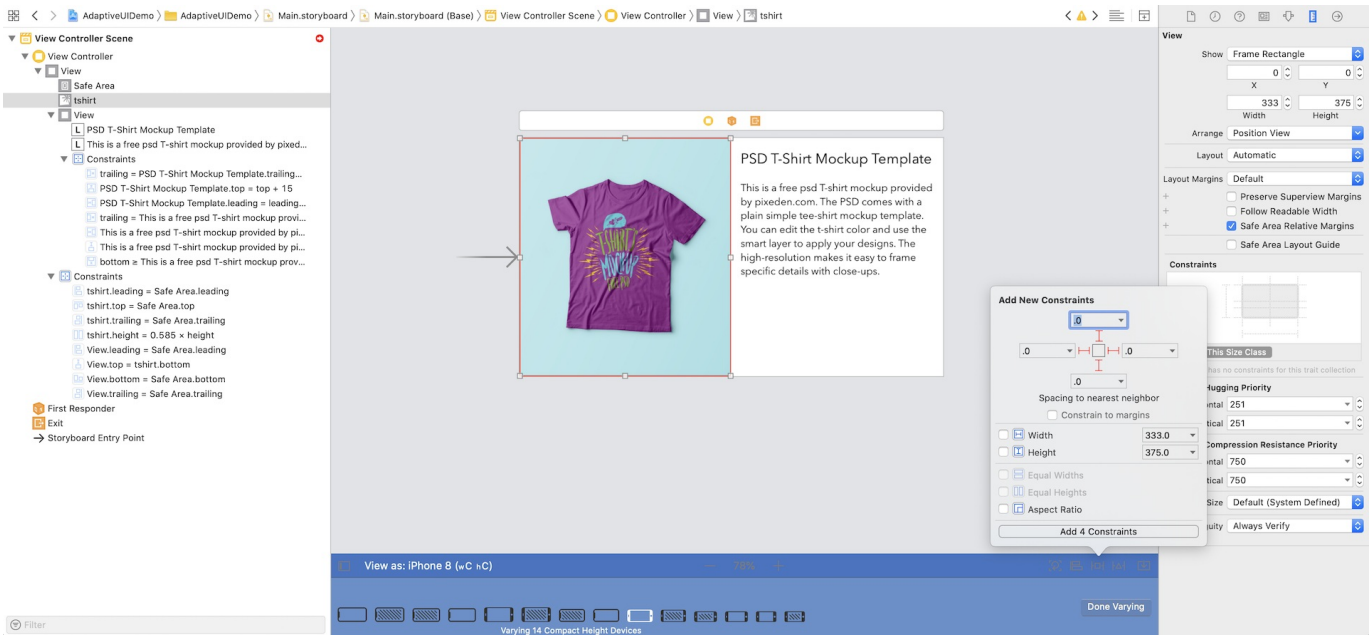


Figure 1.32. Adding layout constraints for the image view

Next, select the view and add three space constraints for the top, left and right sides.

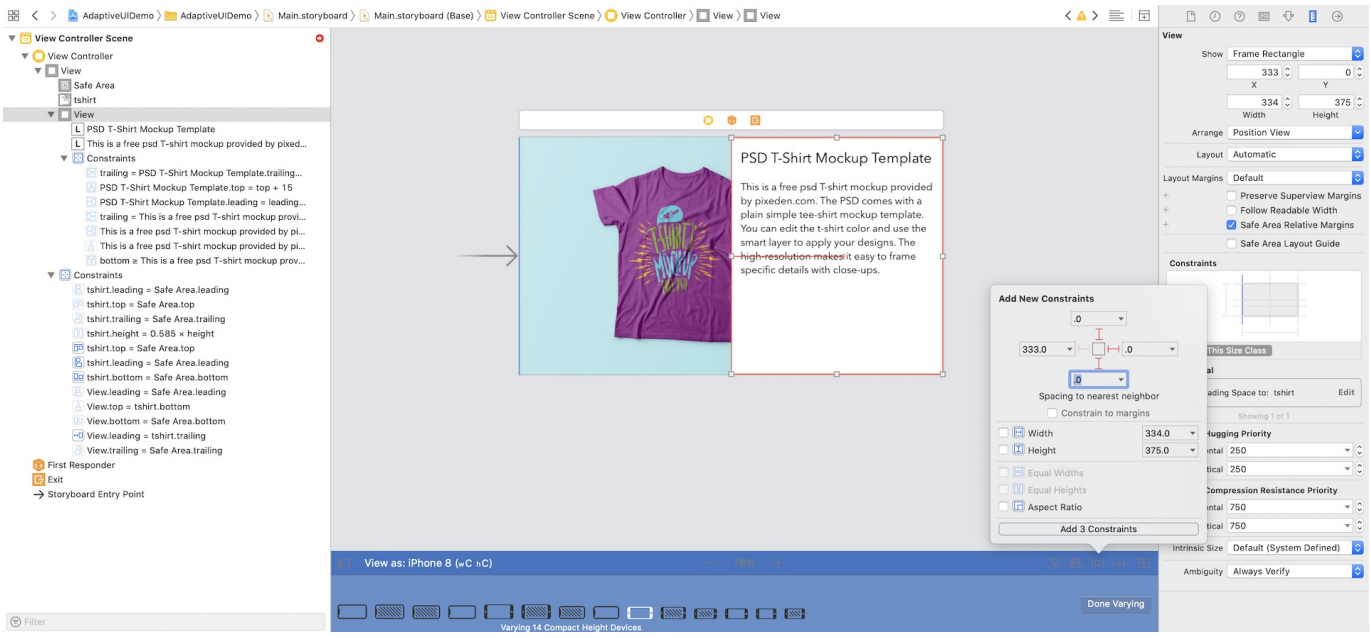


Figure 1.33. Adding layout constraints for the image view

As we want both views to take up 50% of the screen, control-drag from the *t-shirt* image view to the container view. When the popover appears, select `Equal Widths` to add the constraint. Next, select the `tshirt.width = View.width` constraint. Go to the Size inspector and set the multiplier to `1`. This ensures both views to take up 50% of the screen.

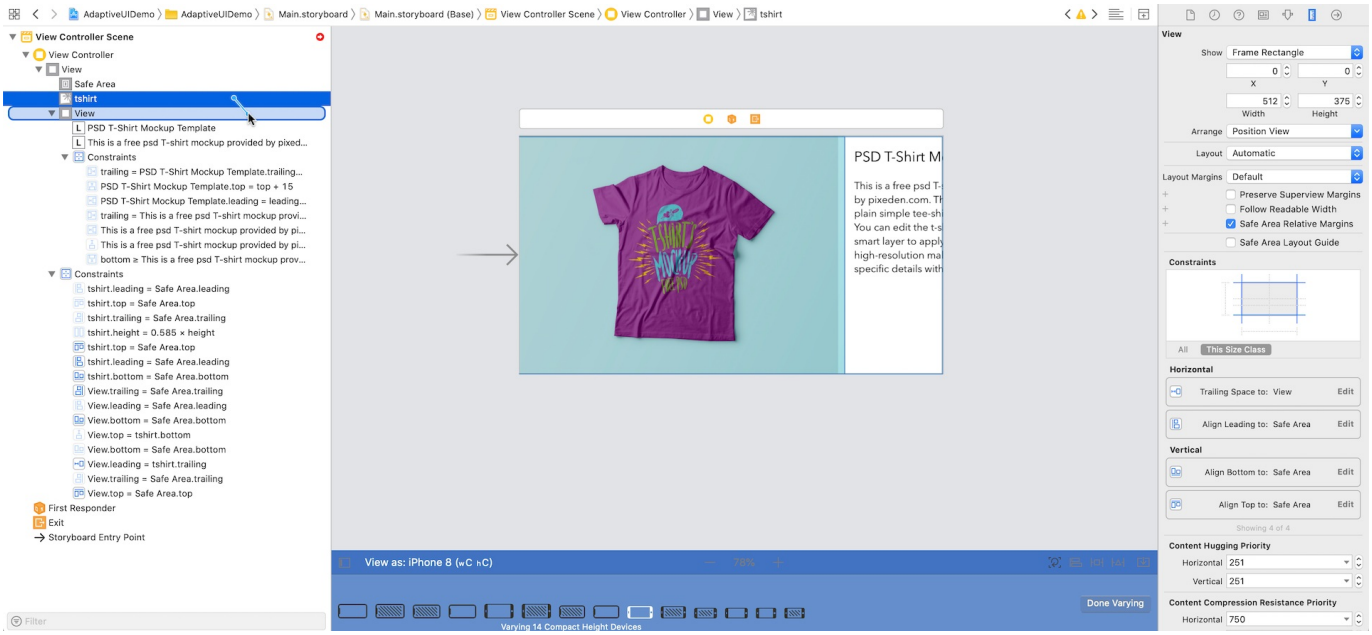


Figure 1.34. Defining an Equal Widths constraint

The rest is to add the layout constraints for the title and description labels. Select the title label and click the Pin button. Add the space constraints for the top, bottom, left and right sides (see figure 1.35). Then, add two space constraints (left and right sides) for the description label.

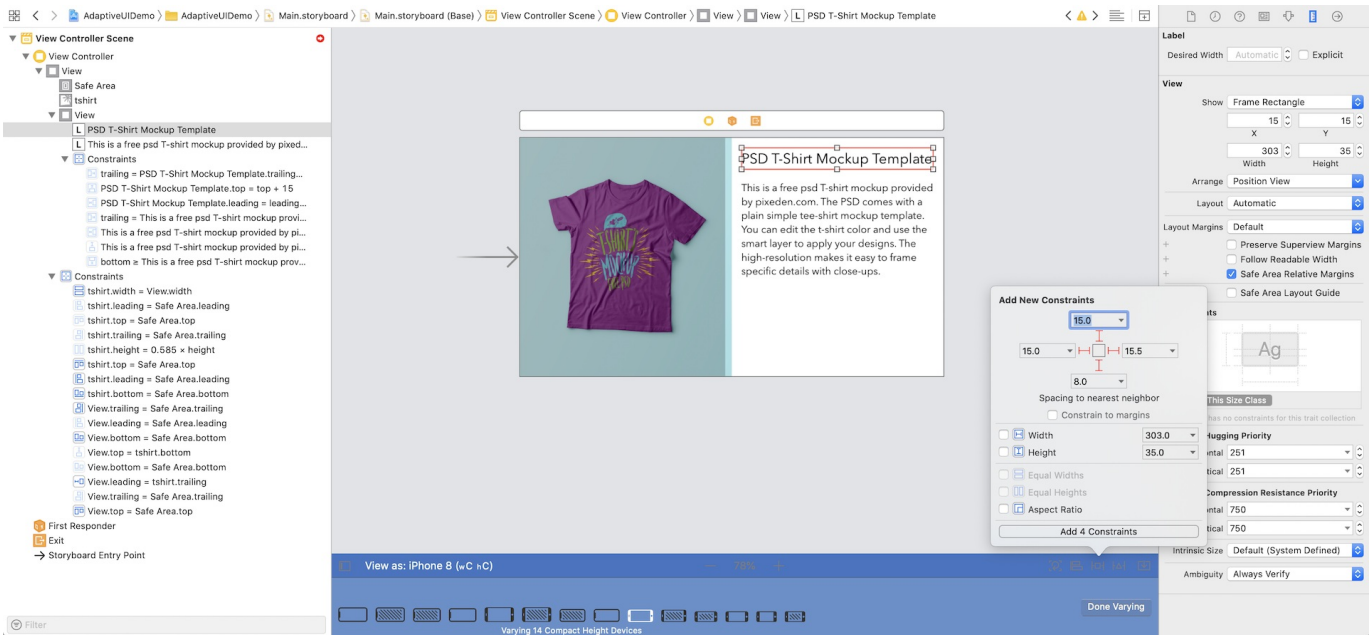


Figure 1.35. Adding space constraints for the title label

Now make sure you click `Done Varying` to save the changes. Your final layout looks similar to figure 1.36.

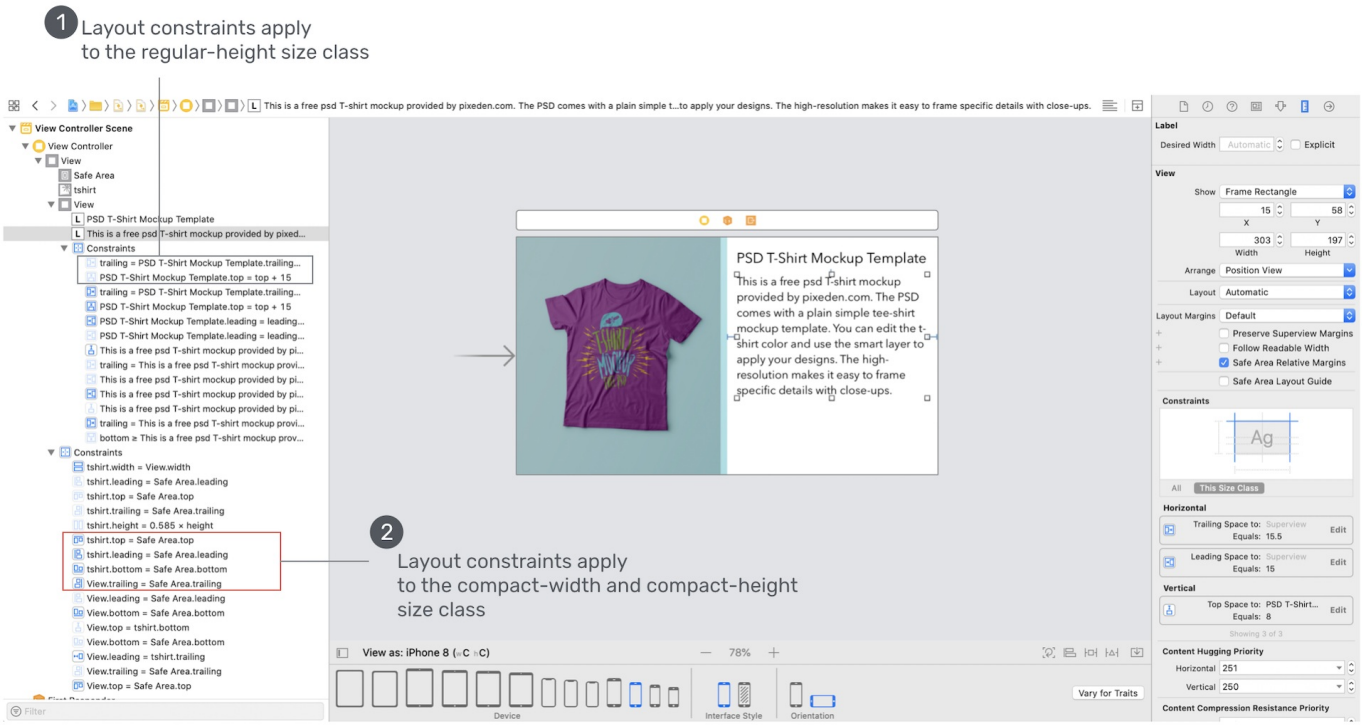


Figure 1.36. Layout constraints for different size classes

If you look closer to the constraints in the document outline view, you should see that some constraints are enabled, while some are grayed out. Those constraints in normal color are applied in the current size class. In this case, it's the *compact-width* and *compact-height* size class. If you switch over to the portrait mode, you will see a different set of constraints enabled.

This is how you use Size Classes to apply different sets of layout constraints, and lay out two different UIs in Interface Builder. If you run the app using any of the iPhone simulators, you will see two distinct UIs for portrait and landscape orientations. Another great thing is that iOS renders a smooth transition when the view is changed from portrait to landscape. It looks pretty awesome, right?

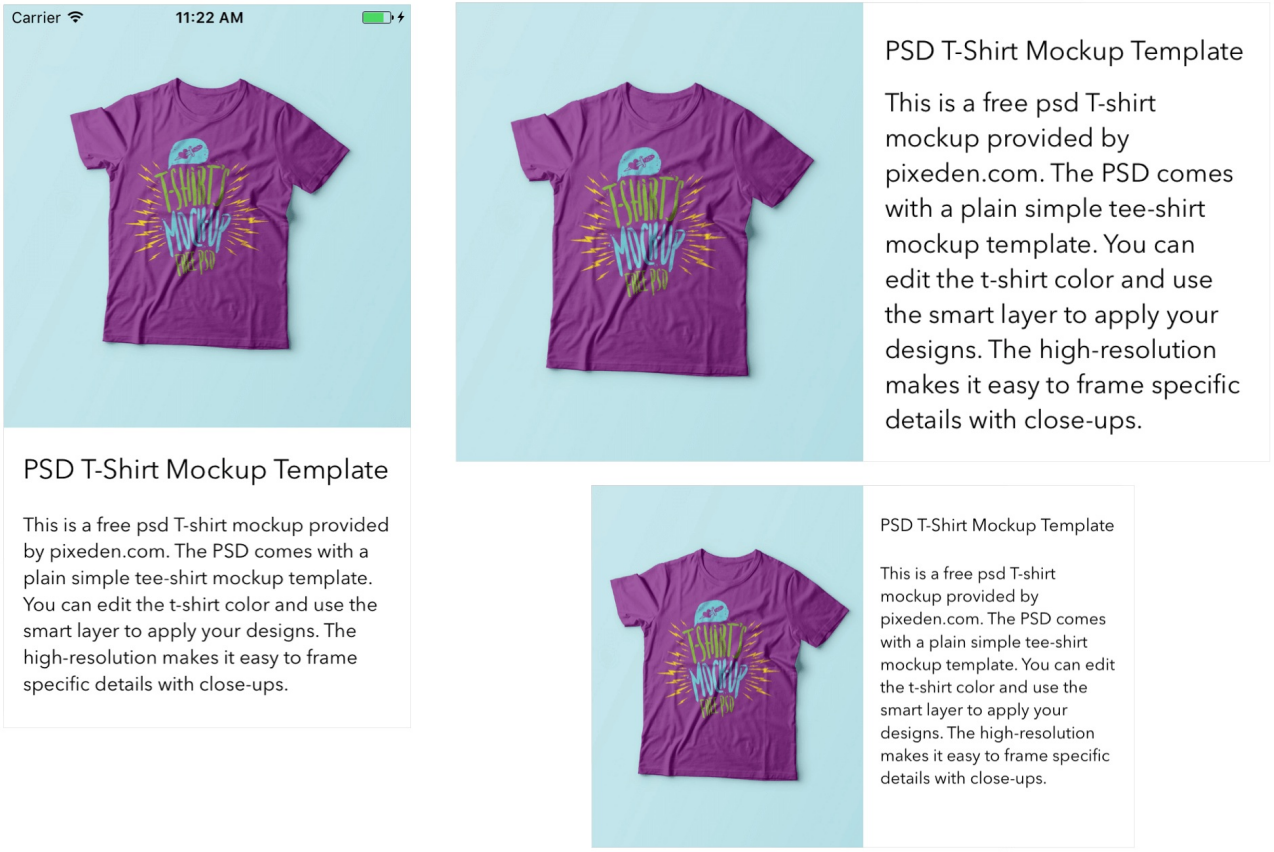


Figure 1.37. App UI in both portrait and landscape orientations

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 8 Plus (landscape) to this view but keep the design intact for other iPhones?

Figure 1.38. New UI Design for iPhone 8 Plus in landscape orientation

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.

First, change the device to iPhone 8 Plus and set the orientation to landscape in the device configuration pane. As we want to apply layout specialization for this device in this particular orientation, click the `Vary for Traits` button, and select both height & width options. Interface Builder should indicate that the change will only apply to the *regular-width* and *compact-height* device. Next, select the vertical space constraint for the top side of the title label. In the Attributes inspector, you should see the constant field. The value defines the vertical space in points. As we want to increase this value for this particular size class, click the + button and confirm to add variation.

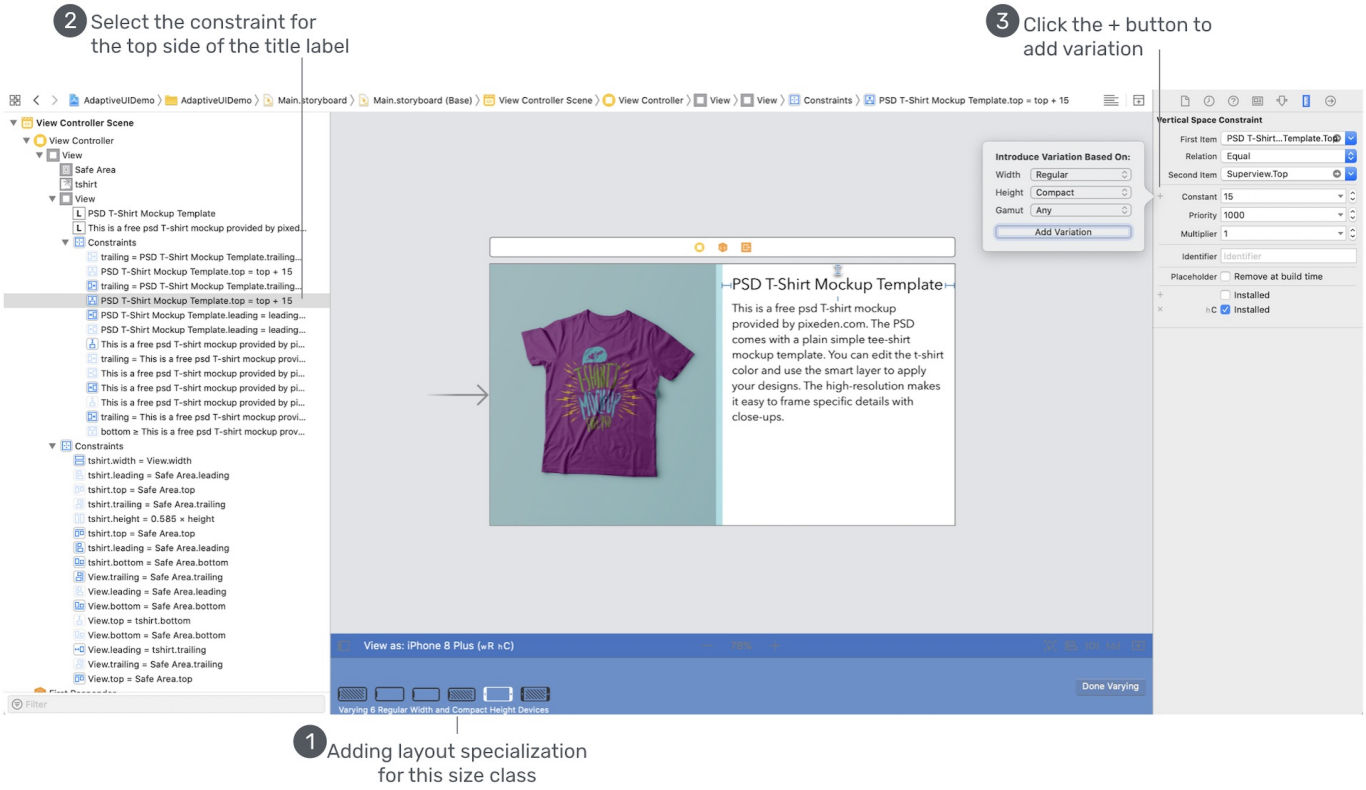


Figure 1.39. Adding user interface variation for iPhone 8 Plus in landscape mode

This will create an additional field for the `wR hC` size class. Set the value to `150` points. That's it. Remember to click the `Done Varying` button to save the changes.

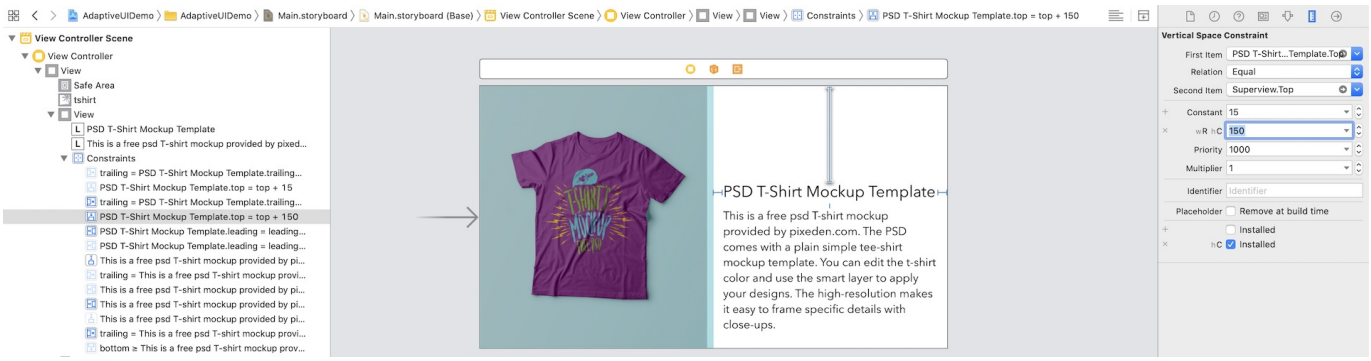


Figure 1.40. Set the vertical space for the regular-width and compact-height size class

You can preview the new UI design in Interface Builder or using the simulator. On iPhone 5.5-inch (or some larger devices), the new UI will appear when the device is in landscape orientation.

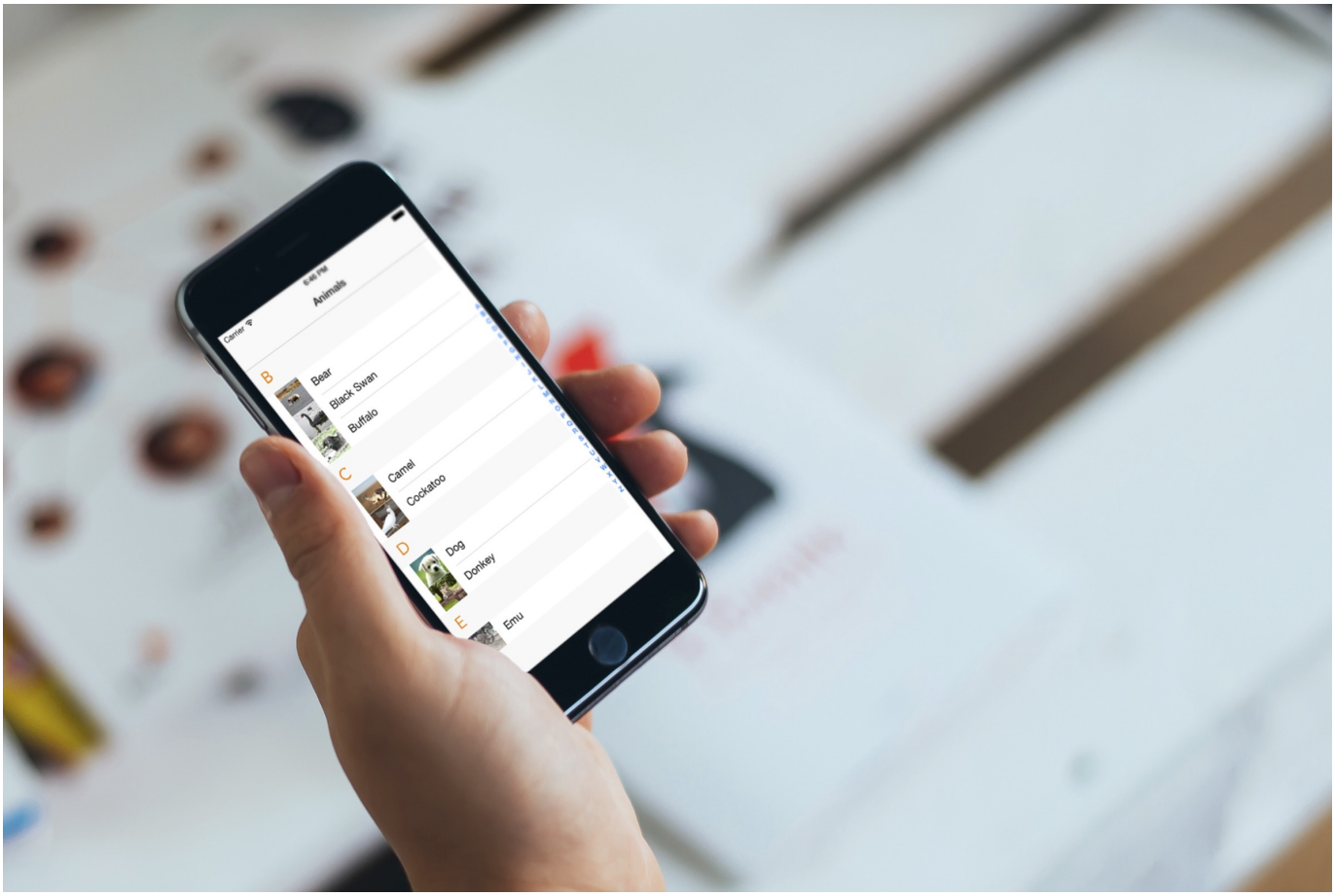
Summary

With Interface Builder, Apple provides developers with a great tool to build adaptive UIs in iOS apps. I hope you already understand the concept of size classes and know how to use them to create adaptive layouts.

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. So get ready for it!

For reference, you can download the Xcode project from http://www.appcoda.com/resources/swift5/AdaptiveUIDemo.zip.

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:

- `numberOfSections(in:)` method – returns the total number of sections in the table view. Usually, we set the number of sections to `1`. If you want to have multiple sections, set this value to a number larger than 1.
- `tableView(_:titleForHeaderInSection:)` method – returns the header titles for different sections. This method is optional if you do not prefer to assign titles to the section.
- `tableView(_:numberOfRowsInSection:)` method – returns the total number of rows in a specific section.
- `tableView(_:cellForRowAt:)` 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.
- `sectionIndexTitles(for:)` 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`.
- `tableView(_:sectionForSectionIndexTitle:at:)` 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.

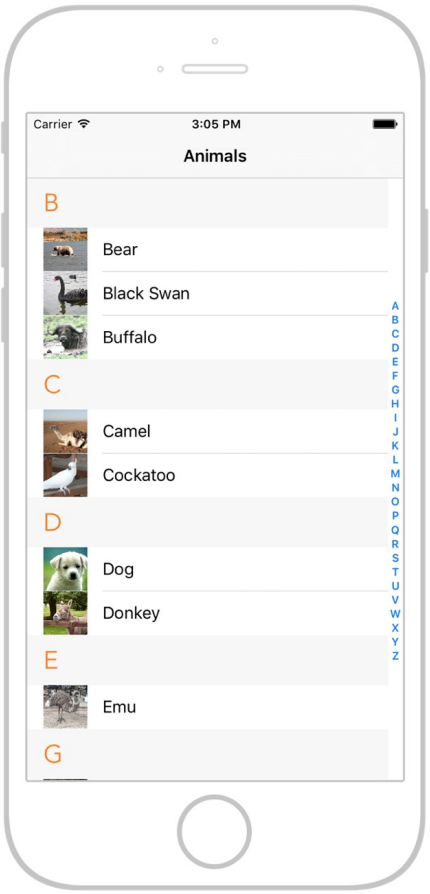


Figure 2.1. Demo app

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 http://www.appcoda.com/resources/swift5/IndexedTableDemoStarter.zip 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.

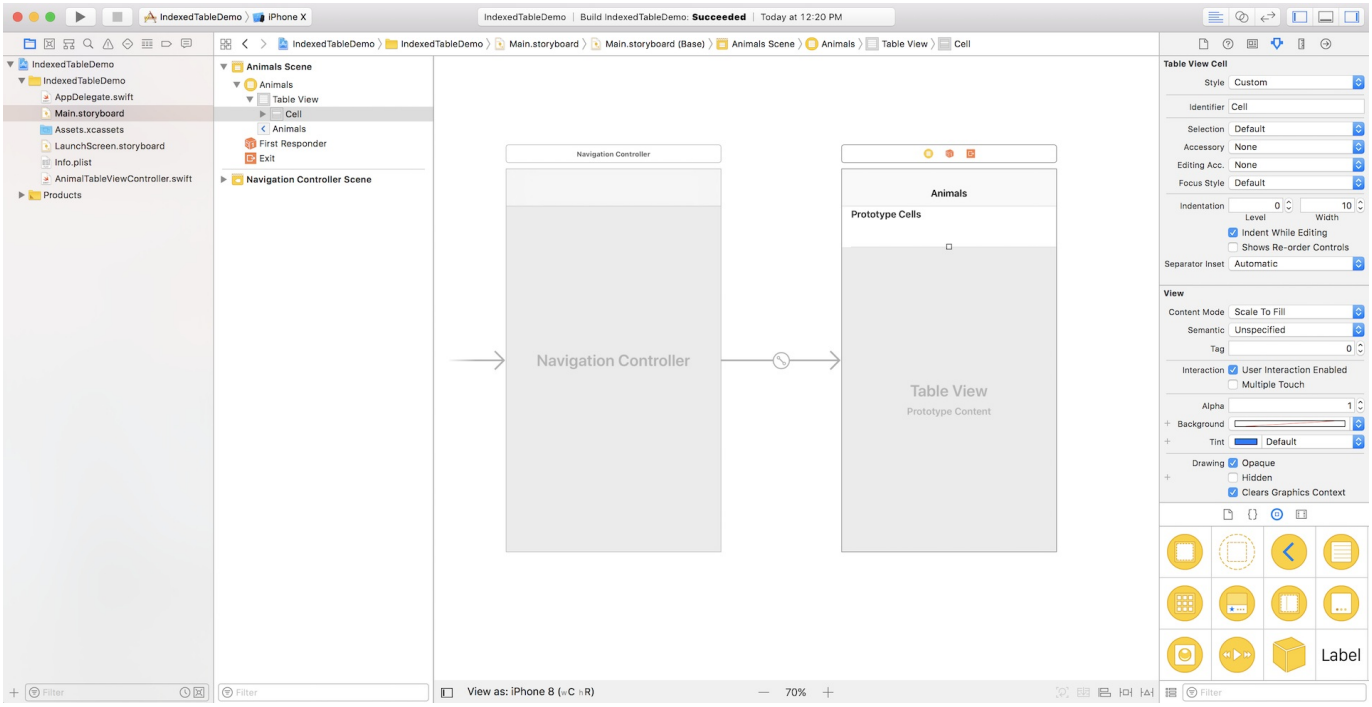


Figure 2.2. Storyboard of the starter project

Displaying Sections in UITableView

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

```
let animals = ["Bear", "Black Swan", "Buffalo", "Camel", "Cockatoo", "Dog", "Donke
y", "Emu", "Giraffe", "Greater Rhea", "Hippopotamus", "Horse", "Koala", "Lion", "L
lama", "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 code above, 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 can manually create the dictionary, but wouldn't it be great if we could create the indexes from the `animals` array programmatically? 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 firstLetterIndex = animal.index(animal.startIndex, offsetBy: 1)
        let animalKey = String(animal[..<firstLetterIndex])

        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 ascendin
g order
    animalSectionTitles = [String](animalsDict.keys)
    animalSectionTitles = animalSectionTitles.sorted(by: { $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. To obtain an index for a specific position (i.e. `String.Index`), you have to ask the string itself for the `startIndex` and then call the `index` method to get the desired position. In this case, the target position is `1`, since we are only interested in the first character.

In older version of Swift, you use `substring(to:)` method of a string to get a new string containing the characters up to a given index. Now, the method has been deprecated. Instead, you slice a string into a substring using subscripting like this:

```
let animalKey = String(animal[..<firstLetterIndex])
```

`animal[..<firstLetterIndex]` slices the `animal` string up to the specified index. In the above case, it means to extract the first character. You may wonder why we need to wrap the returned substring with a String initialization. In Swift 4, when you slice a string into

a substring, you will get a `Substring` instance. It is a temporary object, sharing its storage with the original string. In order to convert a `Substring` instance to a `String` instance, you will need to wrap it with `String()`.

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. Therefore, 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 `sorted`, 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.sorted( by: { (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.sorted(by: { $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.

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

```
animalSectionTitles.sort(by: { $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 `numberOfSections(in:)` method and return the total number of sections:

```
override func numberOfSections(in tableView: UITableView) -> Int {
    // Return the number of sections.
    return animalSectionTitles.count
}
```

To display a header title in each section, we need to implement the `tableView(_:titleForHeaderInSection:)` method. This method is called every time a new section is displayed. Based on the given section index, we 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 `tableView(_:numberOfRowsInSection:)` method in `AnimalTableViewController.swift` like this:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: I
nt) -> Int {
    // Return the number of rows in the section.
    let animalKey = animalSectionTitles[section]
    guard let animalValues = animalsDict[animalKey] else {
        return 0
    }

    return animalValues.count
}
```

When the app starts to render the data in the table view, the `tableView(_: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. Then we return the total number of animal names for that section. In the above code, we use the `guard` keyword to determine if the dictionary returns a valid array for the specific `animalKey`. If not, we just return `0`.

The `guard` keyword is particularly useful in this situation. We want to ensure that `animalValues` contains some values before continuing the execution. And, it makes the code clearer and more readable.

Lastly, modify the `tableView(_:cellForRowAt:)` method as follows:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPat
h) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPat
h)

    // 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].lowercased().replacingOccu
rrences(of: " ", with: "_")
        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

So 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 `sectionIndexTitles(for:)` method and return an array of section indexes. Here we will use the section titles as the indexes.

```
override func sectionIndexTitles(for 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.

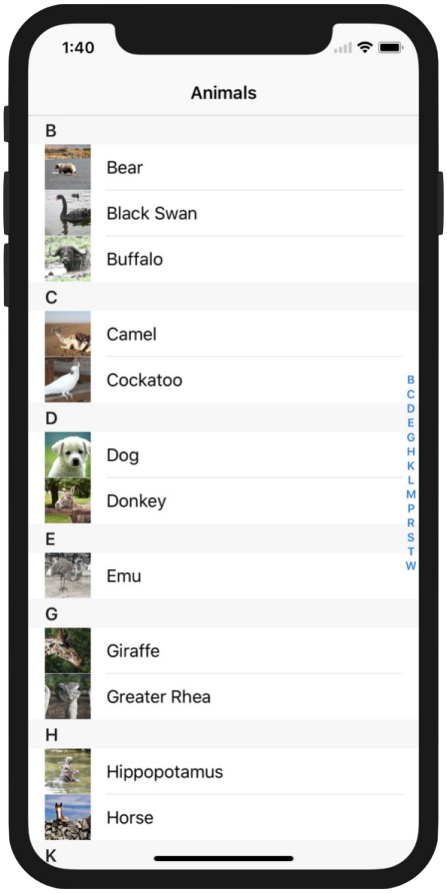


Figure 2.3. Adding an index list to the table view

Adding An A-Z Index List

Looks like we've done everything. But why did we mention the `tableView(_:sectionForSectionIndexTitle:at:)` 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 `AnimalTableViewController.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 `sectionIndexTitles(for:)` method and return the `animalIndexTitles` array instead of the `animalSectionTitles` array.

```
override func sectionIndexTitles(for tableView: UITableView) -> [String]? {
    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.

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 `tableView(_:sectionForSectionIndexTitle:at:)` 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 titl
e: String, at index: Int) -> Int {

    guard let index = animalSectionTitles.firstIndex(of: 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 `firstIndex(of:)` to find the index of a particular item in the array.

The whole point of the implementation is to verify if the given title can be found in the `animalSectionTitles` array and return the corresponding index. Then the table view moves to the corresponding section. For instance, if the title is `B`, we check that `B` is a valid section title and return the index `1`. In case the title is not found (e.g. `A`), we return `-1`.

Compile and run the app again. The index list should now work!

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 a couple of simple changes:

- Alter the height of the section header
- Change the font and background color of the section header

To alter the height of the section header, you can simply override the `tableView(_: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 `tableView(_:willDisplayHeaderView:forSection:)` 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, font, and background color.

```
override func tableView(_ tableView: UITableView, willDisplayHeaderView view: UIVi
ew, forSection section: Int) {
    let headerView = view as! UITableViewHeaderFooterView
    headerView.backgroundView?.backgroundColor = UIColor(red: 236.0/255.0, green:
240.0/255.0, blue: 241.0/255.0, alpha: 1.0)
    headerView.textLabel?.textColor = UIColor(red: 231.0/255.0, green: 76.0/255.0,
 blue: 60.0/255.0, alpha: 1.0)

    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.

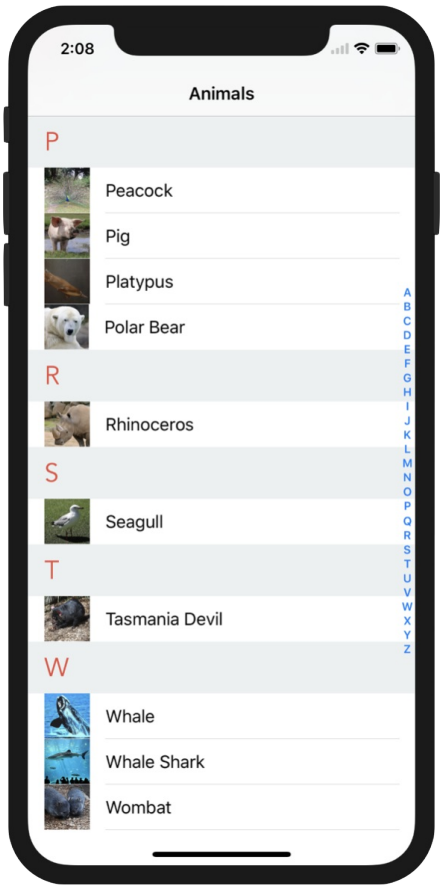


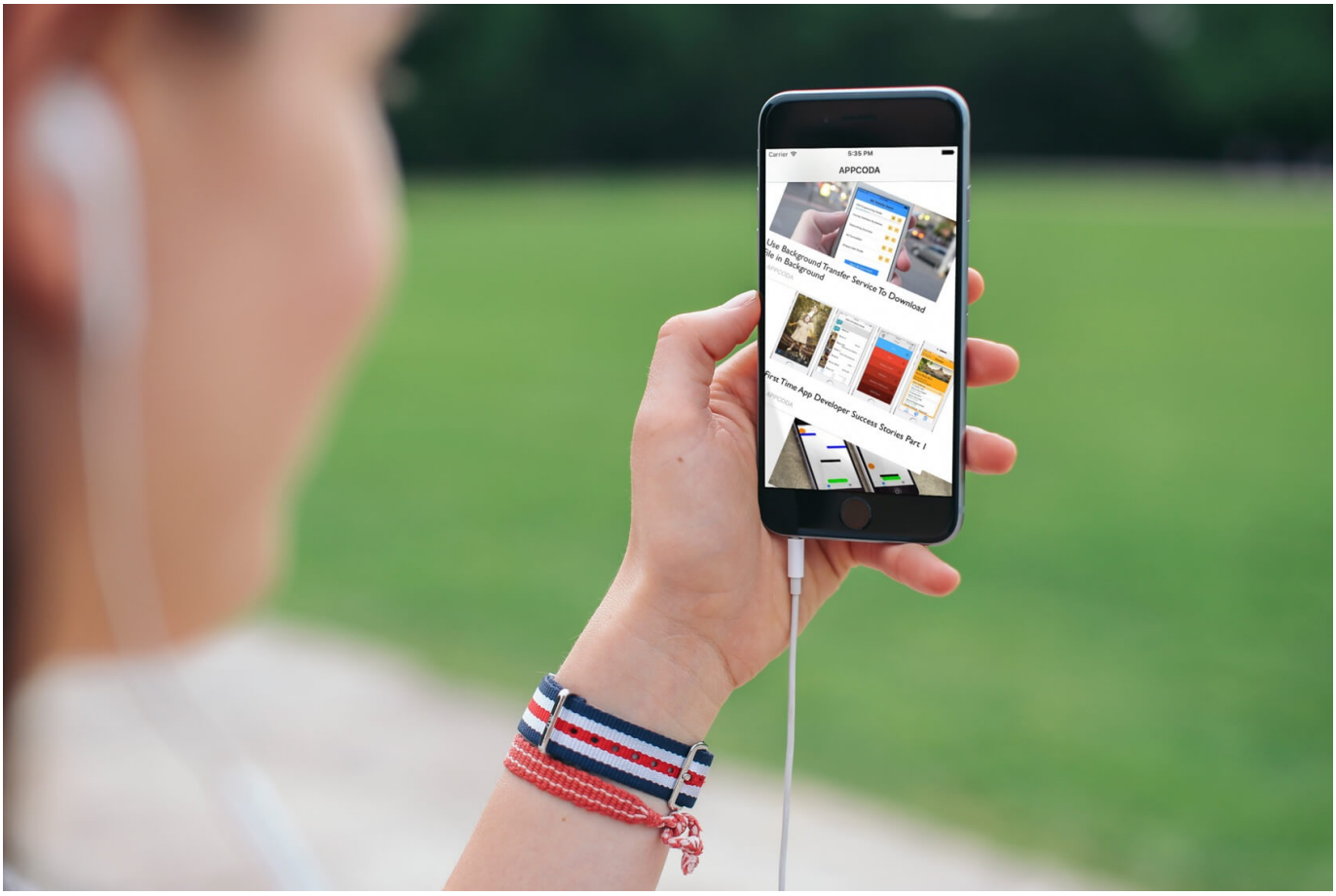
Figure 2.5. Demo app with customized section headers

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 reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/IndexedTableDemo.zip.

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 13 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, I will show you a powerful technique to liven up your app by adding subtle animation. Thanks to the built-in APIs. It is super 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 http://www.appcoda.com/resources/swift5/TableCellAnimationStarter.zip. 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.

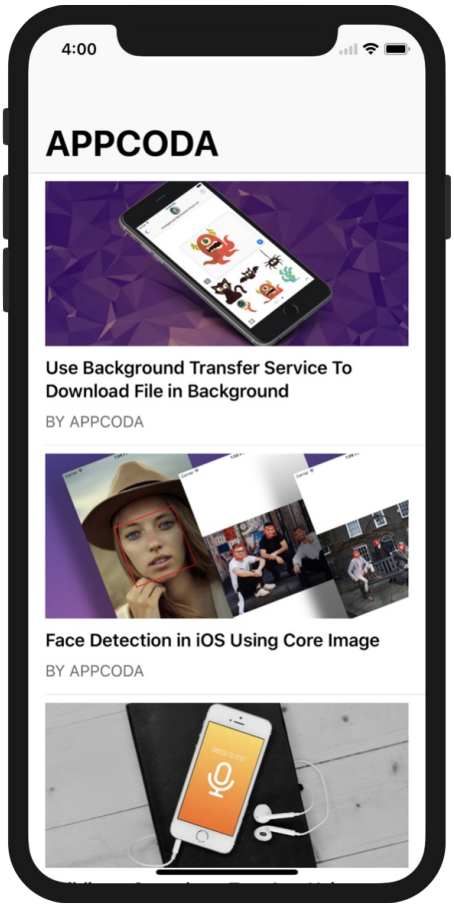


Figure 3.1. The demo app

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(_:willDisplay:forRowAt:)`:

```
optional func tableView(_ tableView: UITableView,
            willDisplay cell: UITableViewCell,
               forRowAt 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, willDisplay cell: UITableViewCel
l, forRowAt indexPath: IndexPath) {

    // Define the initial state (Before the animation)
    cell.alpha = 0

    // Define the final state (After the animation)
    UIView.animate(withDuration: 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 code above, 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(_:willDisplay:forRowAt:)` method is the key to table view cell animation. You can implement whichever type of animation in the method.

The fade-in animation is an example. 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, willDisplay cell: UITableViewCel
l, forRowAt indexPath: IndexPath) {

    // Define the initial state (Before the animation)
    let rotationAngleInRadians = 90.0 * CGFloat(Double.pi/180.0)
    let rotationTransform = CATransform3DMakeRotation(rotationAngleInRadians, 0, 0
, 1)
    cell.layer.transform = rotationTransform

    // Define the final state (After the animation)
    UIView.animate(withDuration: 1.0, animations: { cell.layer.transform = CATrans
form3DIdentity })
}
```

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 to 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!

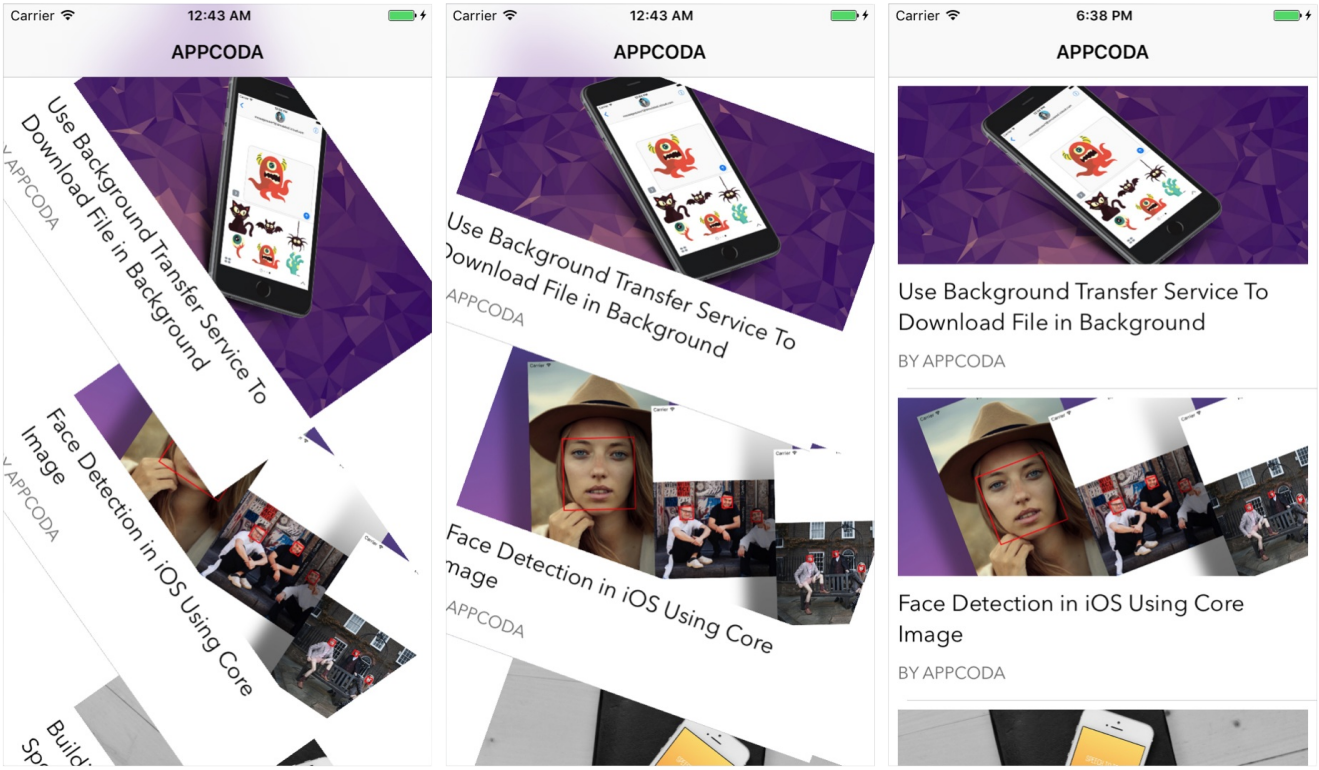


Figure 3.2. Rotating table cells

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.opengl-tutorial.org/beginners-tutorials/tutorial-3-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(_:willDisplay:forRowAt:)` 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.

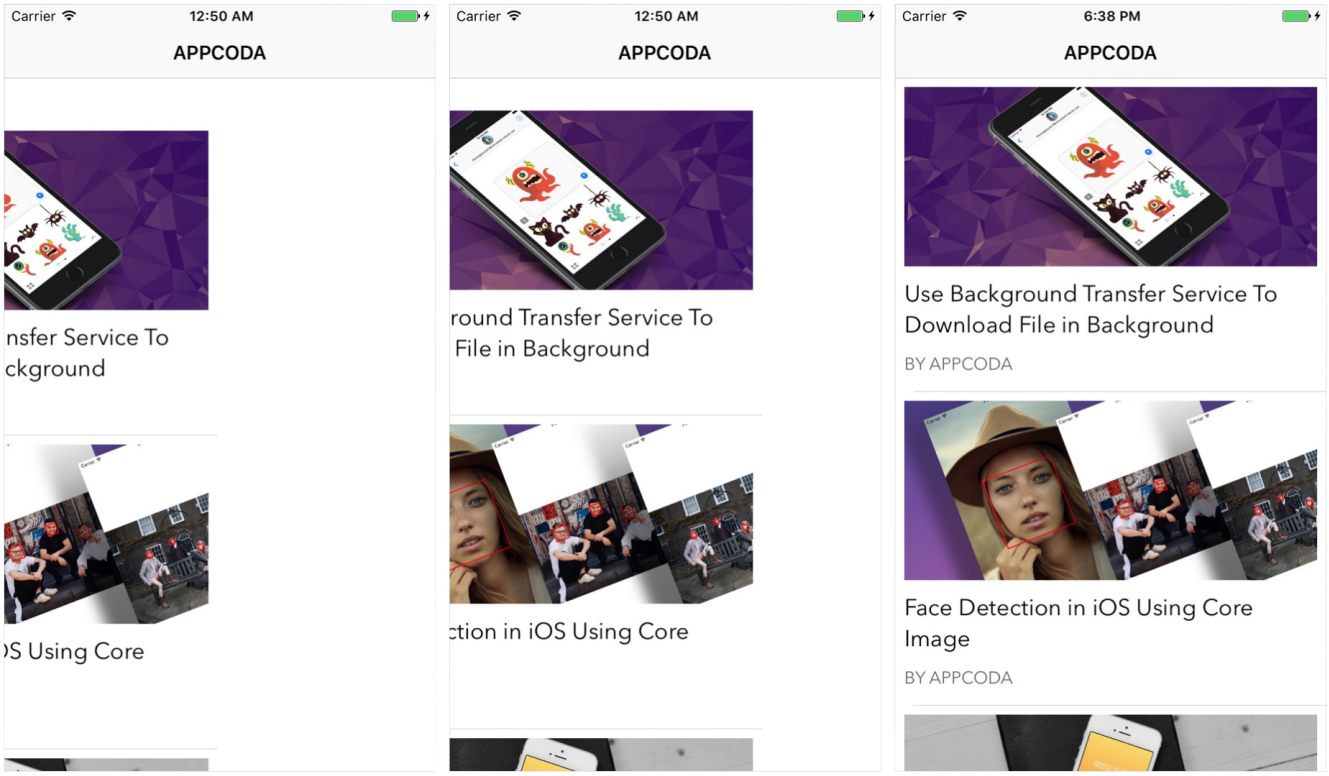


Figure 3.3. Fly-in effect

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 http://www.appcoda.com/resources/swift5/TableCellAnimation.zip. The solution of the exercise is included in the project.

Chapter 4
Working with JSON and Codable

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 every day 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/.

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.

Since the release of Swift 4, Apple introduced the `Codable` protocol to simplify the whole JSON archival and serialization process. We will also look into this new feature and see how we can apply it in JSON parsing.

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 feature called App Transport Security (ATS) 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. Optionally, you can add a key named `NSAllowsArbitraryLoads` in the Info.plist and set the value to YES to disable ATS, so that you can connect to web APIs over HTTP.

However, you'll have to take note if you use `NSAllowsArbitraryLoads` in your apps. In iOS 10, Apple further enforces ATS for all iOS apps. By January 2017, all iOS apps should be ATS-compliant. In other words, if your app connects to external any web services, the connection must be over HTTPS. If your app can't fulfil this requirement, Apple will not allow it to be released on the App Store.

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" = "2016-08-05T09:20:02Z";
        "posted_date" = "2016-07-06T09:20:02Z";
        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 http://www.appcoda.com/resources/swift5/KivaLoanStarter.zip. 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.

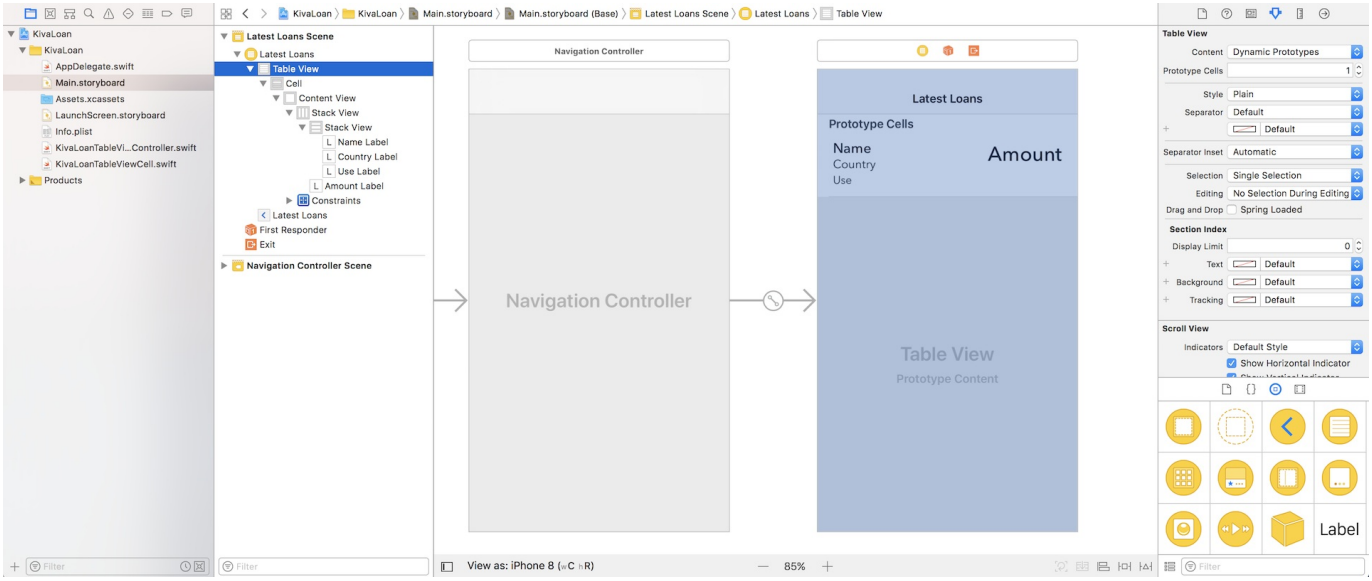


Figure 4.1. Kiva Loan Project Template - Storyboard

Creating JSON Data Model

We will first create a class to model a loan. It's not required for loading JSON but the best practice is 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` structure like this:

```
struct 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:

```
private let kivaLoanURL = "https://api.kivaws.org/v1/loans/newest.json"
private 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() {
    guard let loanUrl = URL(string: kivaLoanURL) else {
        return
    }

    let request = URLRequest(url: loanUrl)
    let task = URLSession.shared.dataTask(with: request, completionHandler: { (dat
a, response, error) -> Void in

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

```
        }

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

            // Reload table view
            OperationQueue.main.addOperation({
                self.tableView.reloadData()
            })
        }
    })

    task.resume()
}

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

    var loans = [Loan]()

    do {
        let jsonResult = try JSONSerialization.jsonObject(with: data, options: JSO
NSerialization.ReadingOptions.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 instantiate the `URL` structure with the URL of the Kiva Loan API. The initialization returns us an optional. This is why we use the `guard` keyword to see if the optional has a value. If not, we simply return and skip all the code in the method.

Next, we create a `URLSession` with the load URL. The `URLSession` class provides APIs for dealing with online content over HTTP and HTTPS. The shared session is good enough for making simple HTTP/HTTPS requests. In case you have to support your own networking protocol, `URLSession` also provides you an option to create a custom session.

One great thing of `URLSession` is that you can add a series of session tasks to 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* (`URLSessionDataTask`) for retrieving data to memory, *download tasks* (`URLSessionDownloadTask`) for downloading a file to disk, and *upload tasks* (`URLSessionUploadTask`) 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 `dataTask` method with the specific URL request. After you add the task, the session will not take any action. You have to call the resume method (i.e. `task.resume()`) to initiate the data task.

Like most networking APIs, the `URLSession` API is asynchronous. Once the request completes, it returns the data (as well as errors) by calling the completion handler.

In the completion handler, immediately after the data is returned, we check for an error. If no error is found, we 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 `JSONSerialization` class, which is capable of

converting JSON to Foundation objects and converting Foundation objects to JSON. In the code snippet, we call the `jsonObject` 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 from 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": "2015-11-20T08:50:02Z",
      "planned_expiration_date": "2016-01-04T08:50:02Z",
      "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": "2016-11-20T08:50:02Z",
      "planned_expiration_date": "2016-01-04T08:50:02Z",
      "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 `JSONSerialization` 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 {
    var 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 `OperationQueue.main.addOperation` 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 call the `reloadData` method in the 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 `OperationQueue.main.addOperation` method and request to run the `reloadData` method in the main queue.

```
OperationQueue.main.addOperation({
    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 `OperationQueue` 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 numberOfSections(in tableView: UITableView) -> Int {
    // Return the number of sections
    return 1
}

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

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPat
h) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPat
h) 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 `tableView(_:cellForRowAt:)` 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)"
```

In some cases, you may want to 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 the App

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.

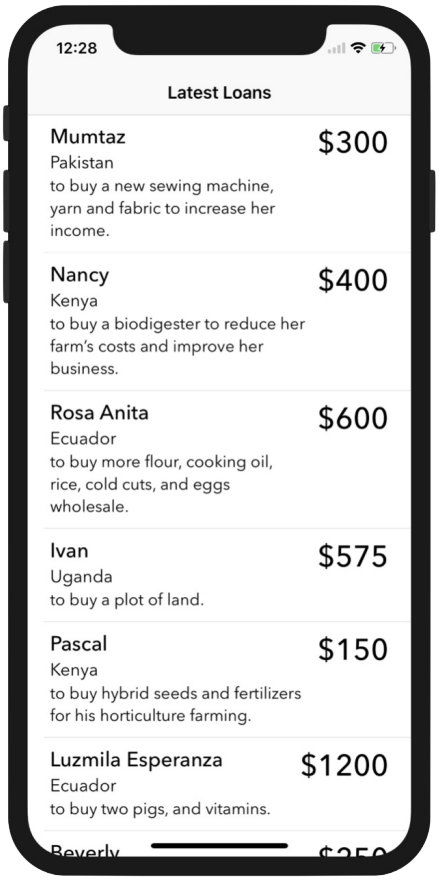


Figure 4.2. The demo app retrieves and displays the latest loan from kiva.org

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/KivaLoan.zip.

Introducing Codable

Since the release of Swift 4, Apple introduced a new way to encode and decode JSON data using `Codable`. We will rewrite the JSON decoding part of the demo app using this new approach.

Before we jump right into the modification, let me give you a basic walkthrough of `Codable`. If you look into the documentation of `Codable`, it is just a type alias of a protocol composition:

```
typealias Codable = Decodable & Encodable
```

`Decodable` and `Encodable` are the two actual protocols you need to work with. However, for convenience's sake, we usually refer to this type alias for handling JSON encoding and decoding.

First, what's the advantage of using `Codable` over the traditional approach for encoding/decoding JSON? If you go back to the previous section and read the code again, you will notice that we had to manually parse the JSON data, convert it into dictionaries and create the `Loan` objects.

`Codable` simplifies the whole process by offering developers a different way to decode (or encode) JSON. As long as your type conforms to the `Codable` protocol, together with the new `JSONDecoder`, you will be able to decode the JSON data into your specified instances. Figure 4.3 illustrates the decoding of a sample loan data into an instance of `Loan` using `JSONDecoder`.

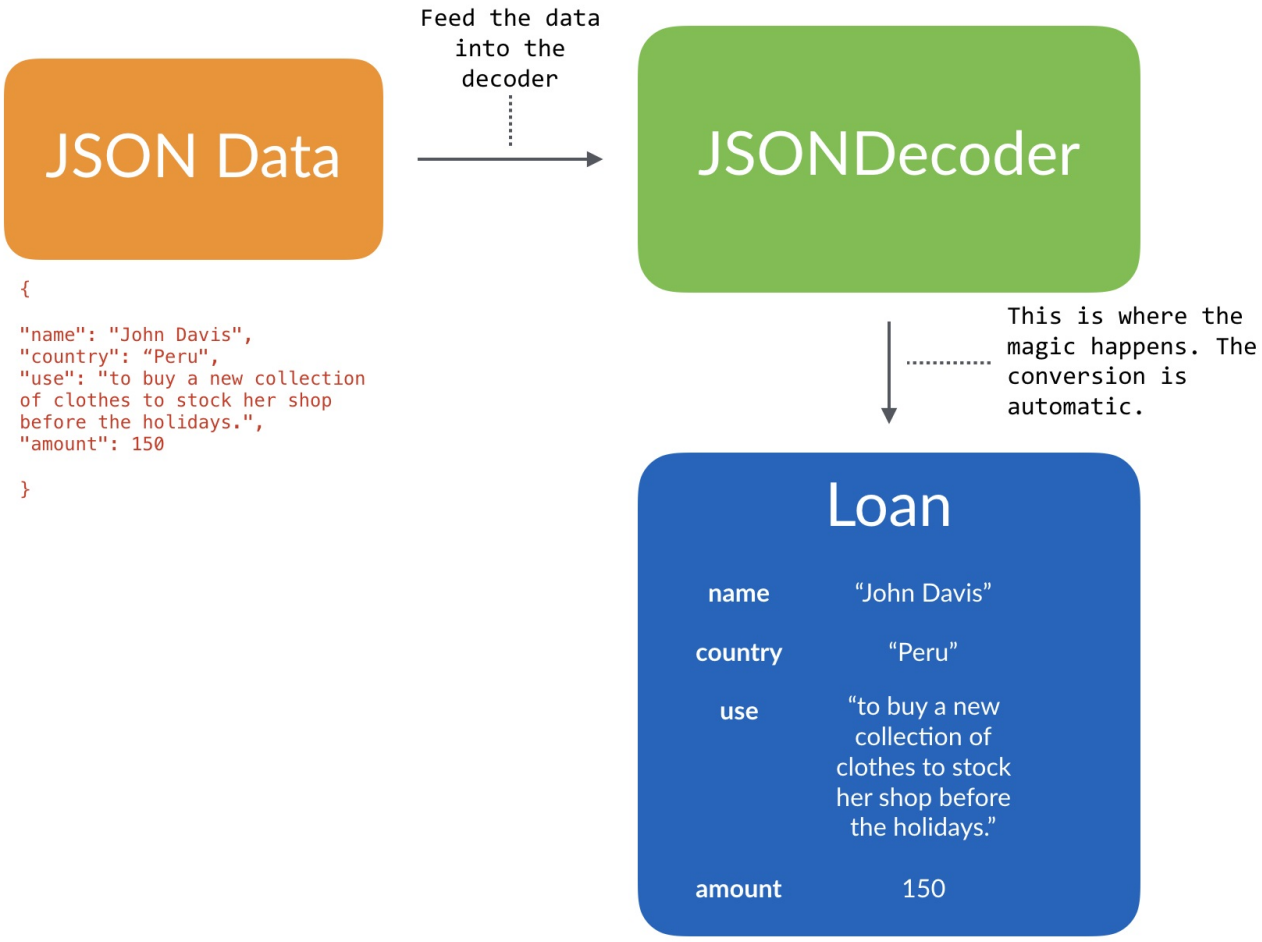


Figure 4.3. JSONDecoder decodes JSON data and convert it into an instance of Loan

Decoding JSON

To give you a better idea about how `Codable` works, let's start a Playground project and write some code. Once you have created your Playground project, declare the following `json` variable:

```
let json = """
{

"name": "John Davis",
"country": "Peru",
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"amount": 150

}
"""
```

We will first start with the basics. Here we define a very simple JSON data with 4 items. The value of the first three items are of the type `String` and the last one is of the type `Int`. As a side note, if this is the first time you see the pair of triple quotes (`"""`), this syntax was introduced in Swift 4 for declaring strings with multi-lines.

Next, declare the `Loan` structure like this:

```
struct Loan: Codable {
    var name: String
    var country: String
    var use: String
    var amount: Int
}
```

This `Loan` structure is very similar to the one we defined in the previous section, except that it adopts the `Codable` protocol. You should also note that the property names match those of the JSON data.

Now, let's see the magic!

Continue to insert the following code in your Playground file:

```
let decoder = JSONDecoder()

if let jsonData = json.data(using: .utf8) {

    do {
        let loan = try decoder.decode(Loan.self, from: jsonData)
        print(loan)

    } catch {
        print(error)
    }
}
```

In the code above, we instantiate an instance of `JSONDecoder` and then convert the JSON string we defined earlier into `Data`. The magic happens in this line of code:

```
let loan = try decoder.decode(Loan.self, from: jsonData)
```

You just need to call the `decode` method of the decoder with the JSON data and specify the type of the value to decode (i.e. `Loan.self`). The decoder will automatically parse the JSON data and convert them into a `Loan` object.

If you've done it correctly, you should see this line in the console:

```
Loan(name: "John Davis", country: "Peru", use: "to buy a new collection of clothes
 to stock her shop before the holidays.", amount: 150)
```

Cool, right?

`JSONDecoder` automatically decodes the JSON data and stores the decoded value in the corresponding property of the specified type (here, it is `Loan`).

Working with Custom Property Names

Earlier, I showed you the simplest example of JSON decoding. However, the decoding process is not always so straightforward. Now, let's take a look another example.

Sometimes, the property name of your type and the key of the JSON data are not exactly matched. How can you perform the decoding?

Let's say, we define the `json` variable like this:

```
let json = """
{

"name": "John Davis",
"country": "Peru",
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"loan_amount": 150

}
"""
```

In the JSON data, the key of the loan amount is changed from `amount` to `loan_amount`. How can we decode the data without changing the property name `amount` of `Loan`?

Now, update the `Loan` structure like this:

```
struct Loan: Codable {
    var name: String
    var country: String
    var use: String
    var amount: Int

    enum CodingKeys: String, CodingKey {
        case name
        case country
        case use
        case amount = "loan_amount"
    }
}
```

To define the mapping between the key and the property name, you are required to declare an enum called `CodingKeys` that has a rawValue of type `String` and conforms to the `CodingKey` protocol. In the enum, you define all the property names of your model and their corresponding key in the JSON data. Say, the case `amount` is defined to map to the key `loan_amount`. If both the property name and the key of the JSON data are the same, you can omit the assignment.

If you've changed the code correctly, you should be able to decode the updated JSON data with the following message found in the console:

```
Loan(name: "John Davis", country: "Peru", use: "to buy a new collection of clothes
 to stock her shop before the holidays.", amount: 150)
```

Working with Nested JSON Objects

The JSON data that we have worked on so far has only one level. In reality, the JSON data is usually more complex with multiple levels. Now let's see how we can decode nested JSON objects.

First, update the `json` variable like this:

```
let json = """
{

"name": "John Davis",
"location": {
"country": "Peru",
},
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"loan_amount": 150

}
"""
```

We've made a minor change to the data by introducing the `location` key that has a nested JSON object with the nested key `country`. How can we decode this type of JSON data and retrieve the value of `country` from the nested object?

Now, modify the `Loan` structure like this:

```
struct Loan: Codable {
    var name: String
    var country: String
    var use: String
    var amount: Int

    enum CodingKeys: String, CodingKey {
        case name
        case country = "location"
        case use
        case amount = "loan_amount"
    }

    enum LocationKeys: String, CodingKey {
        case country
    }

    init(from decoder: Decoder) throws {
        let values = try decoder.container(keyedBy: CodingKeys.self)

        name = try values.decode(String.self, forKey: .name)

        let location = try values.nestedContainer(keyedBy: LocationKeys.self, forK
ey: .country)
        country = try location.decode(String.self, forKey: .country)

        use = try values.decode(String.self, forKey: .use)
        amount = try values.decode(Int.self, forKey: .amount)

    }
}
```

Similar to what we have done earlier, we have to define an enum `CodingKeys`. For the case `country`, we specify to map to the key `location`. To handle the nested JSON object, we need to define an additional enumeration. In the code above, we name it `LocationKeys` and declare the case `country` that matches the key `country` of the nested object.

Since it is not a direct mapping, we need to implement the initializer of the `Decodable` protocol to handle the decoding of all properties. In the `init` method, we first invoke the `container` method of the decoder with `CodingKeys.self` to retrieve the data related to the specified coding keys, which are `name`, `location`, `use` and `amount`.

To decode a specific value, we call the `decode` method with the specific key (e.g. `.name`) and the associated type (e.g. `String.self`). The decoding of the `name`, `use` and `amount` is pretty straightforward. For the `country` property, the decoding is a little bit tricky. We have to call the `nestedContainer` method with `LocationKeys.self` to retrieve the nested JSON object. From the values returned, we further decode the value of `country`.

That is how you decode JSON data with nested objects. If you've followed me correctly, you should see the following message in the console:

```
Loan(name: "John Davis", country: "Peru", use: "to buy a new collection of clothes
 to stock her shop before the holidays.", amount: 150)
```

Working with Arrays

In the JSON data returned from Kiva API, it usually comes with more than one loan. Multiple loans are structured in the form of an array. Now, let's see how to decode an array of JSON objects using Codable.

First, modify the `json` variable like this:

```
let json = """

[{
"name": "John Davis",
"location": {
"country": "Paraguay",
},
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"loan_amount": 150
},
{
"name": "Las Margaritas Group",
"location": {
"country": "Colombia",
},
"use": "to purchase coal in large quantities for resale.",
"loan_amount": 200
}]

"""
```

To decode the above array into an array of `Loan` object, all you need to use is to modify the following line of code from:

```
let loan = try decoder.decode(Loan.self, from: jsonData)
```

to:

```
let loans = try decoder.decode([Loan].self, from: jsonData)
```

As you can see, you just need to specify `[Loan].self` when decoding the JSON data.

Now that the JSON data is fully utilized, but sometimes you may want to ignore some key/value pairs. Let's say, we update the `json` variable like this:

```
let json = """
{
"paging": {
"page": 1,
"total": 6083,
"page_size": 20,
"pages": 305
},
"loans":
[{
"name": "John Davis",
"location": {
"country": "Paraguay",
},
"use": "to buy a new collection of clothes to stock her shop before the holidays."
,
"loan_amount": 150
},
{
"name": "Las Margaritas Group",
"location": {
"country": "Colombia",
},
"use": "to purchase coal in large quantities for resale.",
"loan_amount": 200
}]
}
"""
```

This JSON data comes with two top-level objects : *paging* and *loans*. Apparently, we are only interested in the data related to loans. In this case, how can you decode the array of loans?

To do that, declare another struct named `LoanDataStore` that also adopts `Codable` :

```
struct LoanDataStore: Codable {
    var loans: [Loan]
}
```

This `LoanDataStore` only has a `loans` property that matches the key `loans` of the JSON data.

Now modify the following line of code from:

```
let loans = try decoder.decode([Loan].self, from: jsonData)
```

to:

```
let loanDataStore = try decoder.decode(LoanDataStore.self, from: jsonData)
```

The decoder will automatically decode the `loans` JSON objects and store them into the `loans` array of `LoanDataStore`. You can add the following lines of code to verify the content of the array:

```
for loan in loanDataStore.loans {
    print(loan)
}
```

The console should have an output like this:

```
Loan(name: "John Davis", country: "Paraguay", use: "to buy a new collection of clo
thes to stock her shop before the holidays.", amount: 150)
Loan(name: "Las Margaritas Group", country: "Colombia", use: "to purchase coal in
large quantities for resale.", amount: 200)
```

Using Codable in the KivaLoan App

Now that I believe you have some ideas about how to decode JSON using Codable, let's go back to the KivaLoan project and modify it to use Codable.

Open `Loan.swift` and replace it with the following code:

```
import Foundation

struct Loan: Codable {

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

    enum CodingKeys: String, CodingKey {
        case name
        case country = "location"
        case use
        case amount = "loan_amount"
    }

    enum LocationKeys: String, CodingKey {
        case country
    }

    init(from decoder: Decoder) throws {
        let values = try decoder.container(keyedBy: CodingKeys.self)

        name = try values.decode(String.self, forKey: .name)

        let location = try values.nestedContainer(keyedBy: LocationKeys.self, forK
ey: .country)
        country = try location.decode(String.self, forKey: .country)

        use = try values.decode(String.self, forKey: .use)
        amount = try values.decode(Int.self, forKey: .amount)

    }
}


struct LoanDataStore: Codable {
    var loans: [Loan]
}
```

The code above is exactly the same as the one we developed earlier. The `LoanDataStore` is designed to store an array of loans.

Next, replace the `parseJsonData` method of the `KivaLoanTableViewController` class with the following code:

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

    var loans = [Loan]()

    let decoder = JSONDecoder()

    do {
        let loanDataStore = try decoder.decode(LoanDataStore.self, from: data)
        loans = loanDataStore.loans

    } catch {
        print(error)
    }

    return loans
}
```

Here, we just use the `JSONDecoder` to decode the JSON data instead of `JSONSerialization`. I will not go into the code because it is the same as we have just worked on in the Playground project.

Now you're ready to hit the Run button and test the app in the simulator. Everything should be the same as before. Under the hood, the app now makes use of `Codable` in Swift 5 to decode JSON.

For reference, you can download the Xcode project from http://www.appcoda.com/resources/swift5/KivaLoanCodable.zip.

Chapter 5
How to Integrate the Twitter and Facebook SDK for Social Sharing

With the advent of social networks, I believe you want to provide social sharing in your apps. This is one of the many ways to increase user engagement. In the past, Apple provided a framework known as *Social Framework* that lets developers integrate their apps with some common social networking services such as Facebook and Twitter. 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 or publish Facebook posts within your app.

However, the Social framework no longer supports Facebook and Twitter in iOS 11 (or up). In other words, if you want to provide social sharing feature in your app, you have to integrate with the SDKs provided by these two companies.

In this chapter, I will walk you through the installation procedures and usage of the APIs. Again, we will work on a simple demo app.

Create the Demo Project and Design the Interface

To begin, you can download the starter project from http://www.appcoda.com/resources/swift5/SocialSharingStarter.zip. This simple app just displays a list of restaurants on 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 and Twitter.

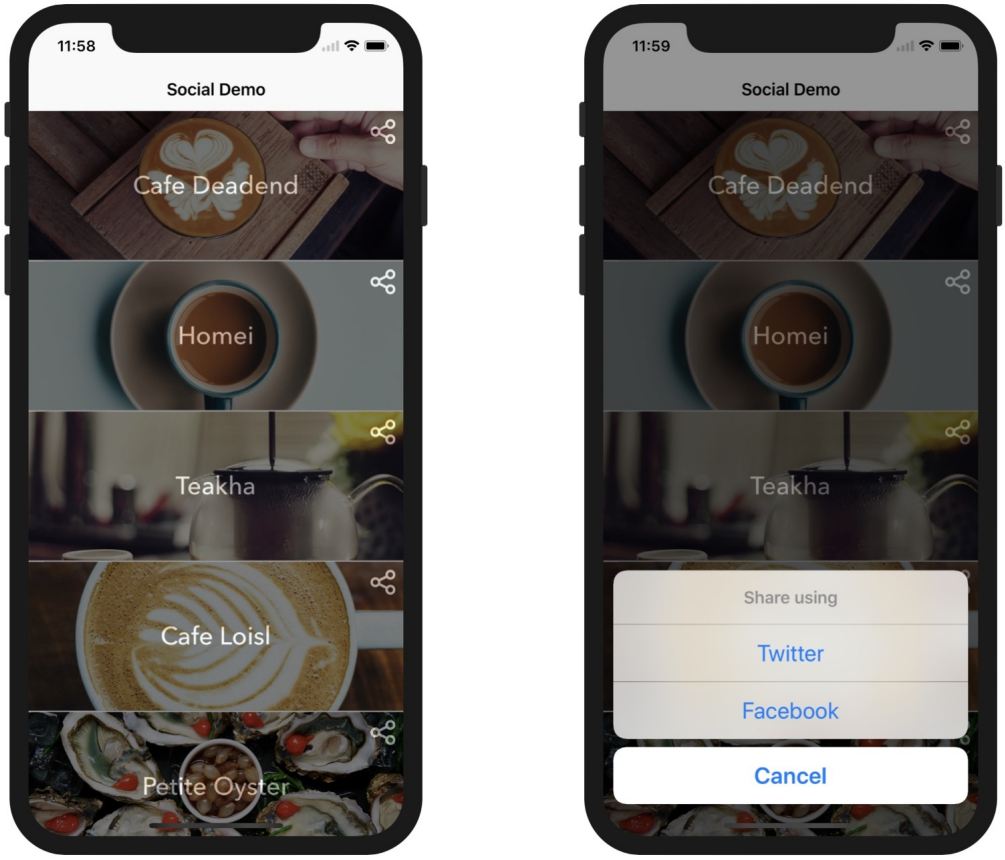


Figure 5.1. Social Sharing Demo App

I have already written some of the code for you, so that we can focus on understanding the Social framework. But it deserves a mention for the following lines of code in the `share` method of the `SocialTableViewController` class:

```
// Get the selected row
let buttonPosition = sender.convert(CGPoint.zero, to: tableView)
guard let indexPath = tableView.indexPathForRow(at: buttonPosition) else {
    return
}
```

If you refer to figure 5.1, each of the cells has a share button. When any of the buttons are tapped, it invokes the `share` action method. One common question is: *how do you know at which row the share button has been tapped?*

There are multiple solutions for this problem. One way to do it is use the `indexPathForRow(at:)` method to determine the index path at a given point. This is how we did it in the starter project. We first convert the coordinate of the button position to that of the table view. Then we get the index path of the cell by calling the `indexPathForRow(at:)` method.

Okay, I am a bit off the topic here. Let's go back to the implementation of the social sharing feature.

The sharing feature has not been implemented in the starter project. This is what we're going to work on.

> **Assumption:** I assume that you understand how `UIAlertController` works. If not, you can refer to our beginner book or the official documentation.

Implementing the Facebook Sharing

Let's begin with the social sharing feature for Facebook. Facebook provides the Facebook SDK for Swift so that iOS developers can integrate with their services like Facebook Login and Share dialogs.

Installing the Facebook SDK

Before we dive into the implementation part, we have to first install the Facebook SDK. The easiest way to do it is by using CocoaPods. If you haven't installed CocoaPods on your Mac, please read chapter 33 first. It will give you an overview of CocoaPods and teach you how it works.

Assuming you have CocoaPods installed, open Terminal and change to your starter project folder. Type the following command to create the `Podfile`:

```
pod init
```

Then edit the `Podfile` like this:

```
target 'SocialSharingDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for SocialSharingDemo
  pod 'FacebookCore'
  pod 'FacebookLogin'
  pod 'FacebookShare'

end
```

In the configuration, we specify to use *FacebookCore*, *FacebookLogin* and *FacebookShare* pods. Now save the configuration file and run the following command in Terminal:

```
pod install
```

CocodPods will then download the required libraries for you and integrate them with the Xcode project. When finish, please make sure to open the `SocialSharingDemo.xcworkspace` file in Xcode.

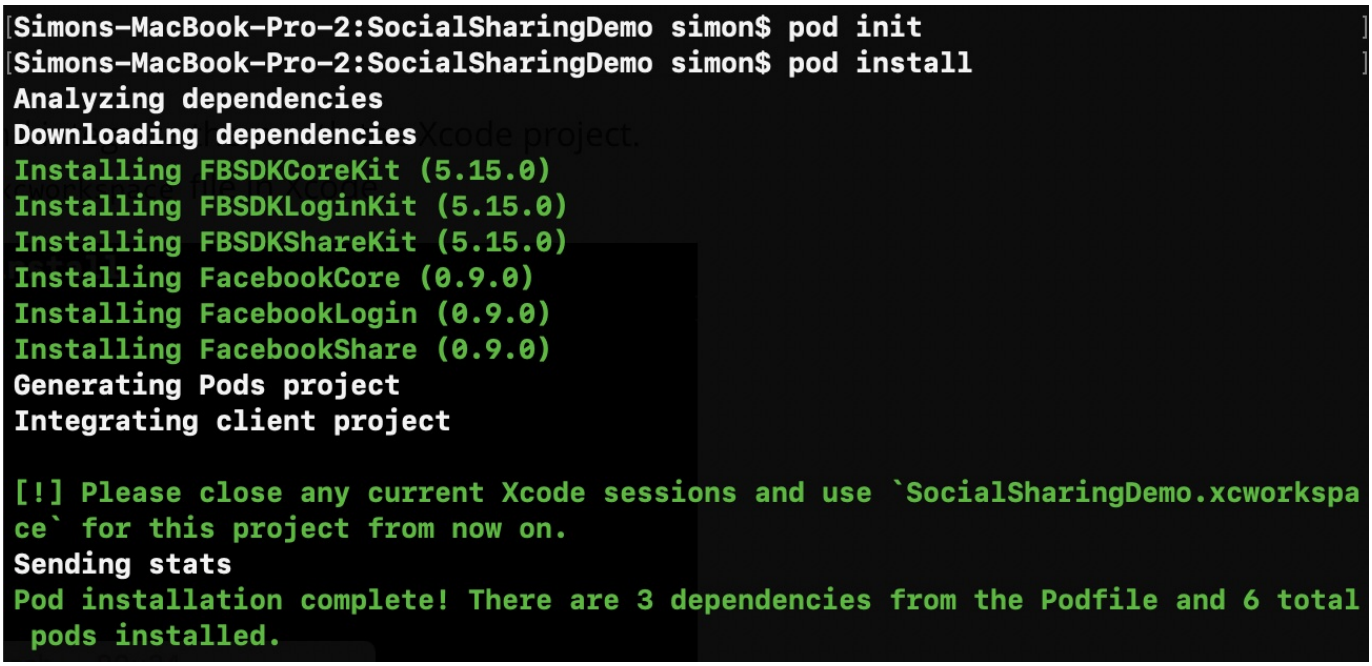


Figure 5.2. Installing the Facebook SDK using CocoaPods

Setting a New Facebook App

Before you write code to integrate with the Facebook platform, it is required to set up a new Facebook app. Go to Facebook for Developers site (https://developers.facebook.com/), log in with your Facebook account. In the dropdown menu of My Apps, choose *Create app*. You will be prompted to give your app a display name and contact email. I set the name to `SocialDemoApp` and click the *Create App ID*.

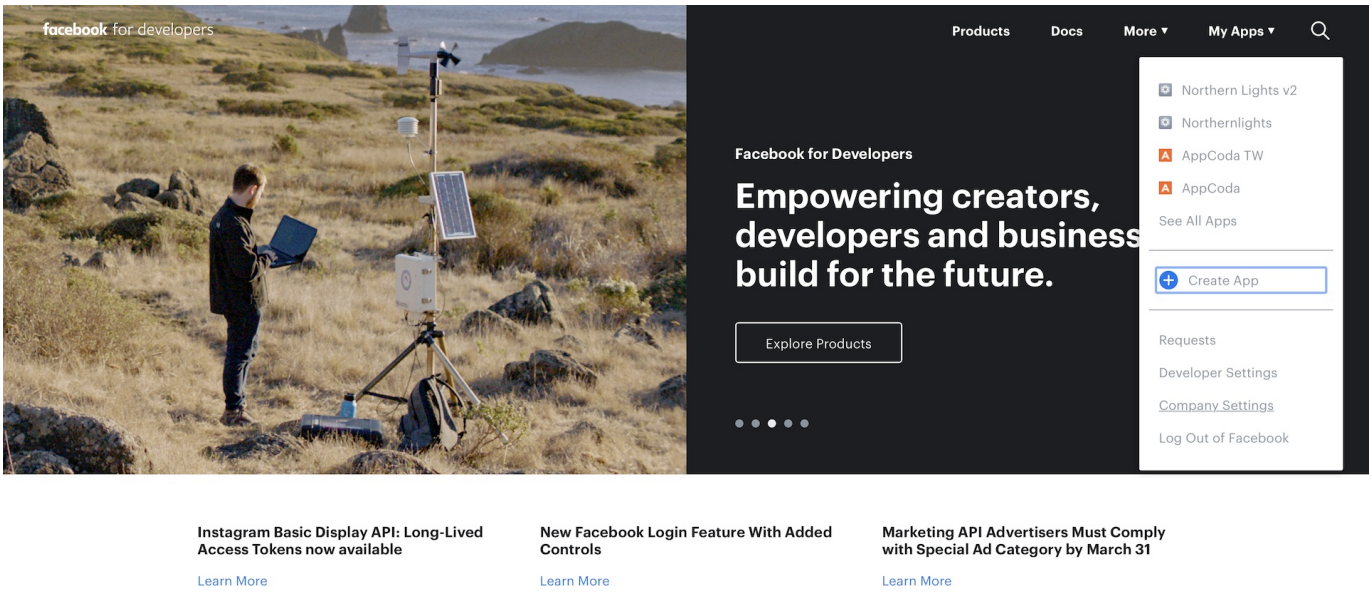


Figure 5.3. Adding a new Facebook app

Next, click `Create App ID` to proceed. Afterwards, choose *Settings* in the side menu and then click *Add Platform*.

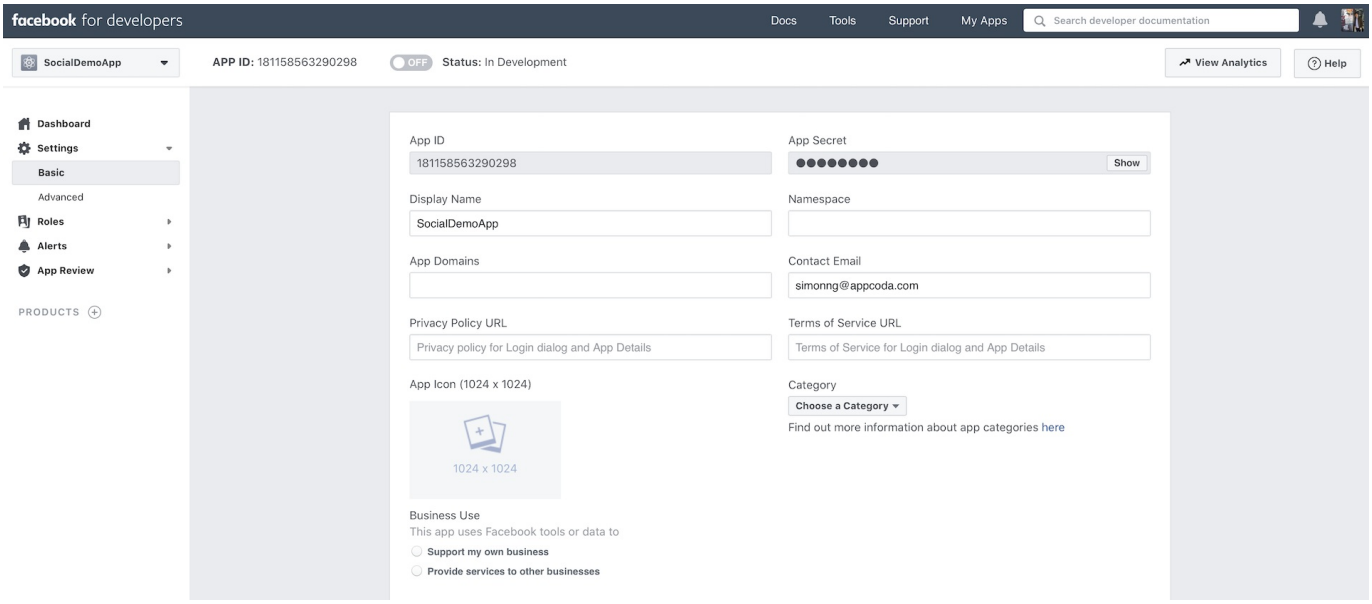


Figure 5.4. Adding a new platform in Settings

In the popover dialog, choose *iOS*. Fill in the bundle ID of your Xcode project. Please note that you shouldn't copy my bundle ID. Use your own bundle ID instead. Remember to hit the *Save Changes* button to save the setting.

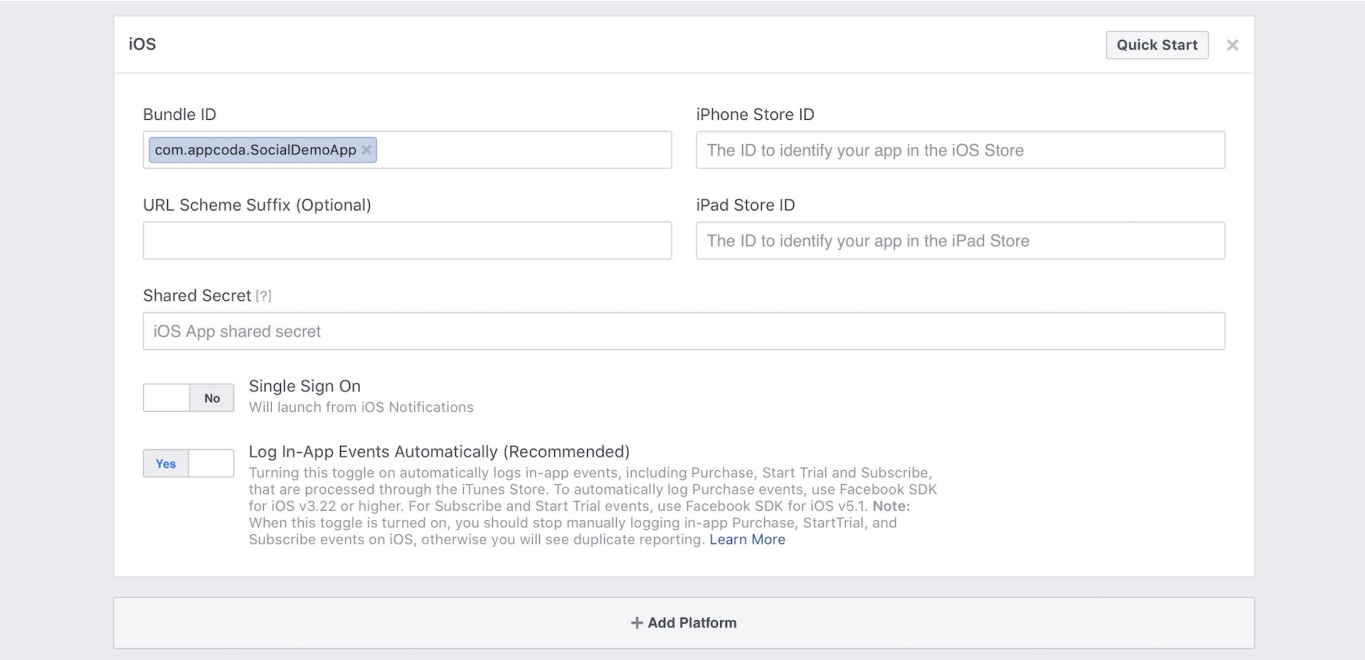


Figure 5.5. Setting the bundle ID

By default, the app is in development mode. In order to test the app using a real Facebook account, go to *App Review* and flip the switch to *YES* to make your app public. Please note that you need to provide a link to your privacy policy in order to take your app live.

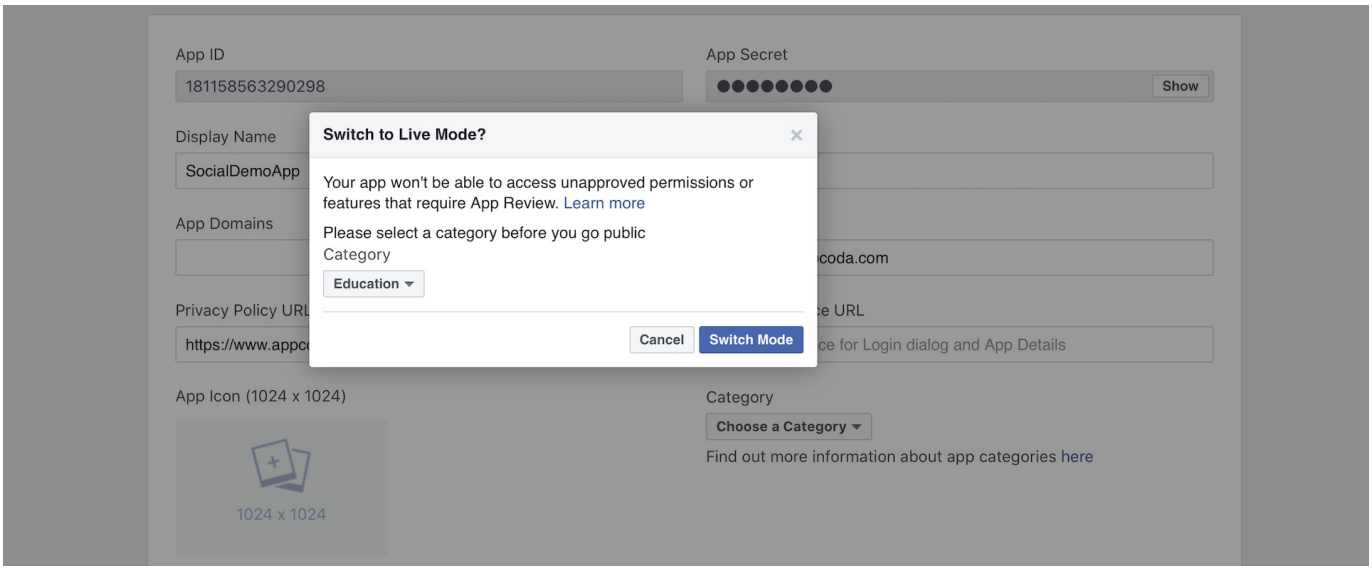


Figure 5.6. Make the app public

Configuring the Xcode Project

Now that you've configured your Facebook app for iOS, it is time to move on to the actual implementation. Go back to your Xcode project. First, please review your bundle ID. Make sure it is set to the same value as you defined in your Facebook app earlier.

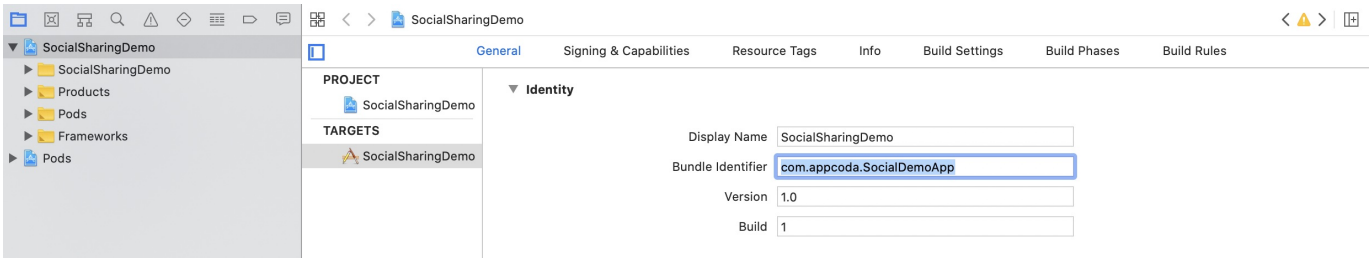


Figure 5.7. Bundle ID defined in your Xcode project

There is one more configuration before we dive into the Swift code. Open `SocialSharingDemo.xcworkspace`. In project navigator, right click the `Info.plist` file and choose *Open as* > *Source code*. This will open the file, which is actually an XML file, in text format.

Insert the following XML snippet before the `</dict>` tag:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>fb137896523578923</string>
        </array>
    </dict>
</array>
<key>FacebookAppID</key>
<string>137896523578923</string>
<key>FacebookDisplayName</key>
<string>Social Sharing Demo</string>
<key>LSApplicationQueriesSchemes</key>
<array>
    <string>fbapi</string>
    <string>fb-messenger-api</string>
    <string>fbauth2</string>
    <string>fbshareextension</string>
</array>
```

The snippet above is my own configuration. Yours should be different from mine, so please make the following changes:

- Change the App ID (`137896523578923`) to your own ID. You can reveal this ID in the dashboard of your Facebook app.
- Change `fb137896523578923` to your own URL scheme. Replace it with `fb{your app ID}` .
- Optionally, you can change the display name of the app (i.e. Social Sharing Demo) to your own name.

The Facebook APIs read the configuration specified in `Info.plist` for connecting your Facebook app. You have to ensure the App ID matches the one you created in the earlier section.

The `LSApplicationQueriesSchemes` key specifies the URL schemes your app can use with the `canOpenURL:` method of the `UIApplication` class. If the user has the official Facebook app installed, it may switch to the app for login purpose. In such case, it is required to declare the required URL schemes in this key, so that Facebook can properly perform the app switch.

Using the Facebook API

Now that we have completed the configuration, it is time to dive into the code. Open `SocialTableViewController.swift` and look into the `share` action method. You should find the code snippet shown below that instantiates the `UIAlertAction` instances of Twitter and Facebook actions.

```
// Display the share menu
let shareMenu = UIAlertController(title: nil, message: "Share using", preferredSty
le: .actionSheet)
let twitterAction = UIAlertAction(title: "Twitter", style: .default, handler: nil)
let facebookAction = UIAlertAction(title: "Facebook", style: .default, handler: nil
)
let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, handler: nil)

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

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

For all the `UIAlertAction` instances, the handler is set to `nil`. Now we will first implement the `facebookAction` for users to share a photo.

Because we are going to use the Facebook Share framework, the first thing you have to do is import the `FacebookShare` framework. Place the following statement at the very beginning of the `SocialTableViewController` class:

```
import FacebookShare
```

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

```
let facebookAction = UIAlertAction(title: "Facebook", style: .default) { (action)
in

    let selectedImageName = self.restaurantImages[indexPath.row]

    guard let selectedImage = UIImage(named: selectedImageName) else {
        return
    }

    let photo = SharePhoto(image: selectedImage, userGenerated: false)
    let content = SharePhotoContent()
    content.photos = [ photo ]

    let shareDialog = ShareDialog(fromViewController: self, content: content, dele
gate: nil)
    shareDialog.show()

}
```

Before testing the app, let's go through the above code line by line. First, we find out the selected image and use a guard statement to validate if we can load the image. To share a photo using the Facebook framework, you have to create a `SharePhoto` object with the selected image and then instantiate a `SharePhotoContent` object for the photo

The `ShareDialog` class is a very handy class for creating a share dialog with the specified content. Once you call its `show()` method, it will automatically show the appropriate share dialog depending on the type of content and the device's application. For example, if the device has the native Facebook app installed, the `ShareDialog` class will direct the user to the Facebook app for sharing.

Now you are ready to test the app. In order to share photos, it is required for the device to have the native Facebook app installed. Therefore, remember to deploy the app to a real device with the Facebook app installed and test out the share feature.

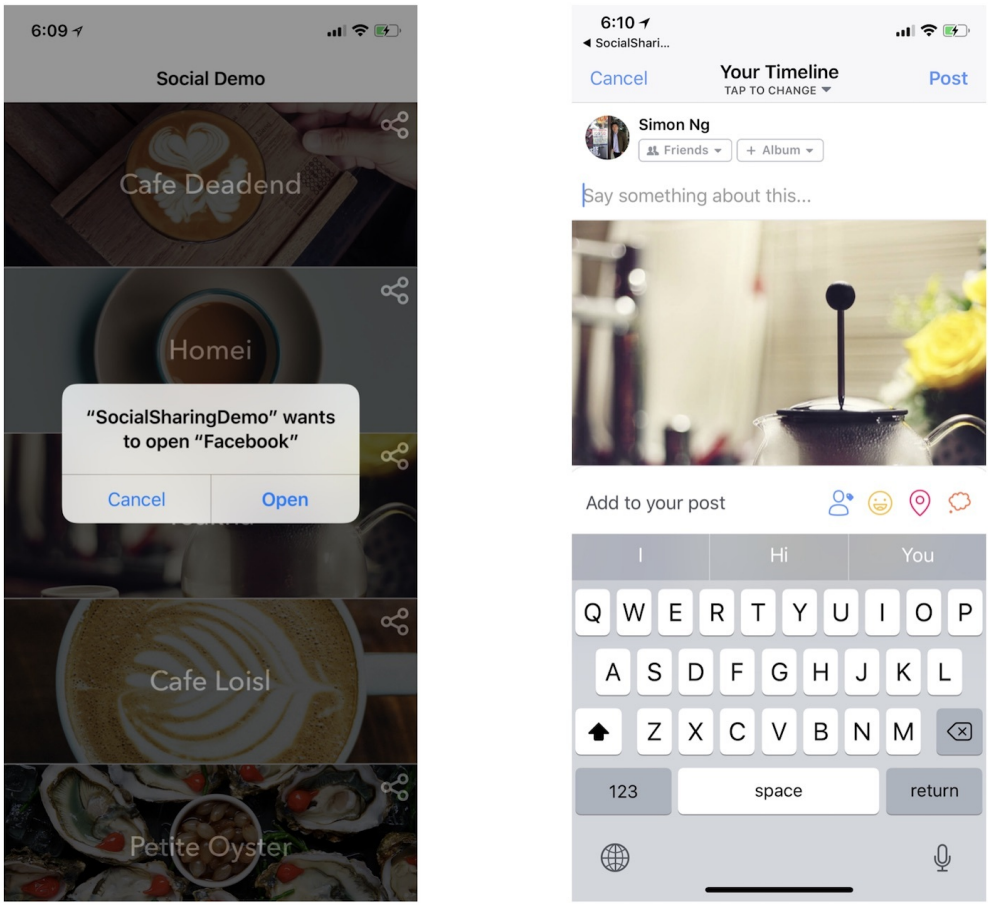


Figure 5.8. When you choose to share the photo on Facebook, the app will automatically switch over to the Facebook app and create a post with your selected photo

While this demo shows you how to initiate a photo share, the Facebook Share framework supports other content types such as links and videos. Say, if you want to share a link, you can replace the `content` variable like this:

```
let content = ShareLinkContent()
content.contentURL = URL(string: "https://www.appcoda.com")!
```

You use the `LinkShareContent` class to create a link share. For details of other content types, you can refer to the official documentation (https://developers.facebook.com/docs/swift/sharing/content-types).

Implementing the Twitter Sharing

Now it's time to implement the Twitter button. Twitter also provides the SDK for composing tweets on iOS, which is known for Twitter Kit for iOS. Not only can you use the kit for creating tweets, you are allowed to display tweets and integrate with Twitter Login. In this chapter, we will focus on the tweet composer.

Installing the Twitter Kit

Twitter requires that all API requests be authenticated with tokens. Thus, you have to apply an API key for your app on Twitter's developer dashboard. Visit https://apps.twitter.com/ and click the `Create New App` button.

Please note that you need to apply for a Twitter Developer account before you're allowed to create an app.

Assuming you're able to create a new app, fill in the app details to create the app including the app name, app description, and other required fields.

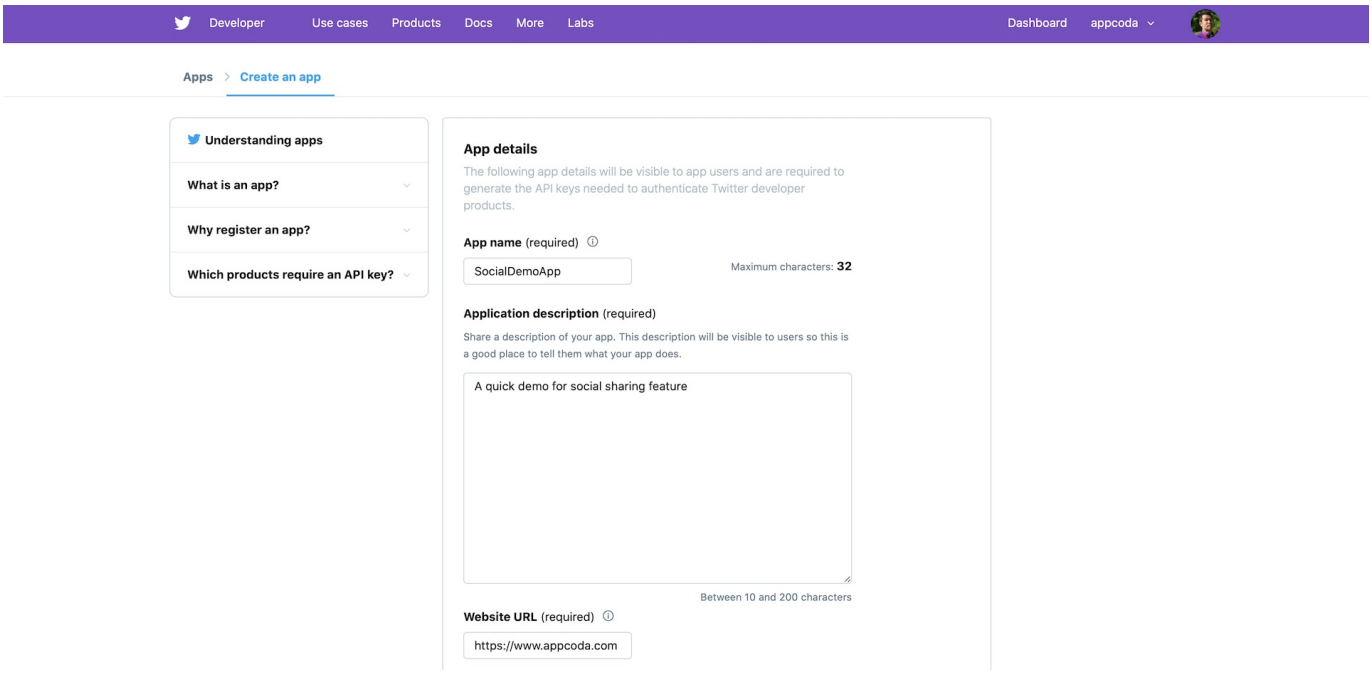


Figure 5.9. Fill in the app information

Once the application is created, choose *Keys and Access Tokens*. You will find your application's API key and secret. Later, you will need these keys in the project configuration.

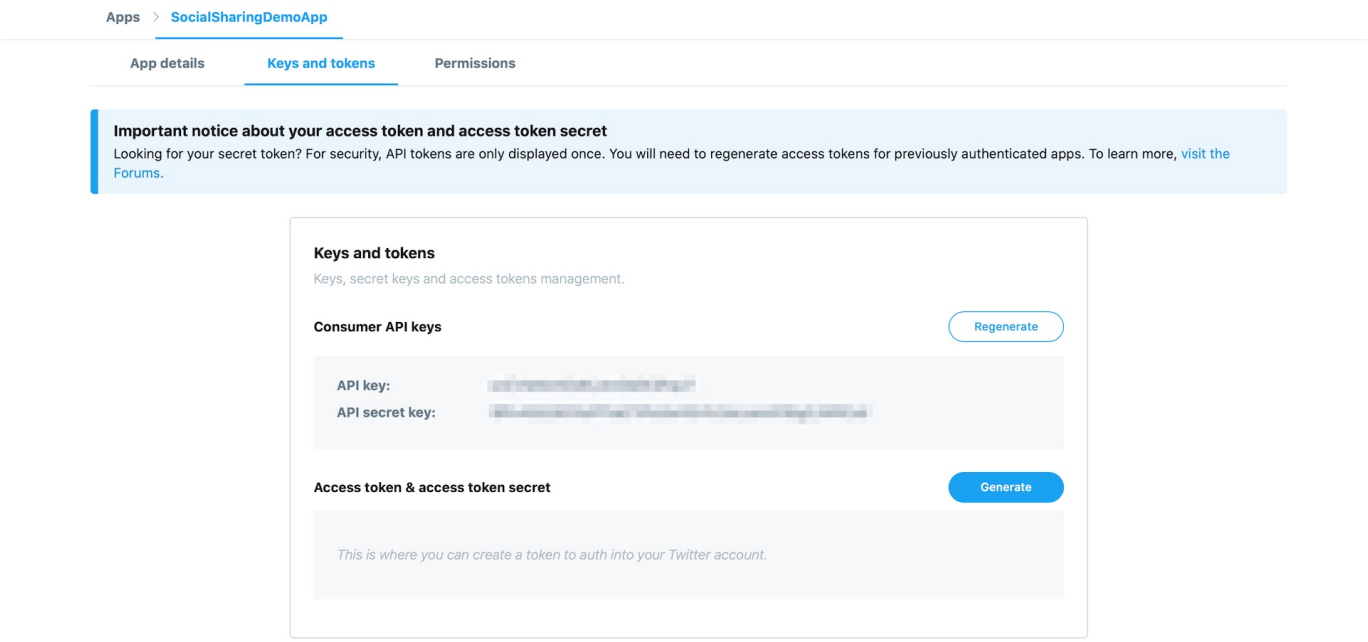


Figure 5.10. Keys and access tokens

Before we move to the coding part, go back to App Details and set the *Callback URL* to `twitterkit-<consumer-key>` where is the API key that can be found in the *Keys and tokens* tab.

Installing Twitter Kit Using CocoaPods

Similar to the installation of the Facebook SDK, the easiest way to install the Twitter Kit is through CocoaPods. Assuming you have CocoaPods installed on your Mac, open Terminal and go to your Xcode project folder. Edit the `Podfile` like this:

```
target 'SocialSharingDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for SocialSharingDemo
  pod 'FacebookCore'
  pod 'FacebookLogin'
  pod 'FacebookShare'

  # Pods for Twitter
  pod 'TwitterKit'

end
```

Here we just insert the line `pod 'TwitterKit'` in the file. Save the changes and then type the following command to install the Twitter Kit:

```
pod install
```

Now you are ready to code. Remember to open the `SocialSharingDemo.xcworkspace` file using Xcode.

Configuring the Info.plist

Twitter Kit looks for a URL scheme in the format of `twitterkit-<consumerKey>`, where `consumerKey` is your application's API key. To define this URL scheme, open `Info.plist` and insert the following configure before the `</dict>` tag:

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>twitterkit-vm4dasvZYI2nDorQC9ziNOEXv</string>
        </array>
    </dict>
</array>
```

Please make sure you replace my API key with yours. If you forget your API key, you can refer to the Twitter developer dashboard to find it.

Also, please update the key *LSApplicationQueriesSchemes* with the following values:

```
<key>LSApplicationQueriesSchemes</key>
<array>
    <string>fbapi</string>
    <string>fb-messenger-api</string>
    <string>fbauth2</string>
    <string>fbshareextension</string>
    <string>twitter</string>
    <string>twitterauth</string>
</array>
```

We just added two new entries for Twitter to *LSApplicationQueriesSchemes*. If you do not add these entries, your app will not be able to open the Twitter app for authentication.

Using the Twitter Kit

Now that you have everything configured, let's dive into the coding part. Open the `AppDelegate.swift` file and insert the import statement to import the Twitter Kit framework:

```
import TwitterKit
```

Update the `application(_:willFinishLaunchingWithOptions:)` method like this:

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

    TWTRTwitter.sharedInstance().start(withConsumerKey:"vm4dasvZYI2nDorQC9ziNOEXv"
, consumerSecret:"8QJVWWl4HuWK1MDfdvUjC6M5JuaXxv6FqPLqRfe3y9O2FoZOsE")

    return true
}
```

Again, please make sure you replace the consumer key and secret with yours.

In the same class, insert the following method:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.Open
URLOptionsKey : Any] = [:]) -> Bool {
    return TWTRTwitter.sharedInstance().application(app, open: url, options: optio
ns)
}
```

If the user hasn't logged on Twitter with his/her account, the Twitter Kit will automatically bring up a web interface to prompt for the login. Alternatively, if the user has the Twitter app installed on the device, it will switch over to the Twitter app to ask for permission. Once the login completes, the method is called to register the callback URL. The line of the code simply passes along the redirect URL to Twitter Kit.

Now open the `SocialTableViewController.swift` file and import the Twitter Kit:

```
import TwitterKit
```

In the `share` method, replace `twitterAction` with the following code:

```
let twitterAction = UIAlertAction(title: "Twitter", style: .default) { (action) in

    let selectedImageName = self.restaurantImages[indexPath.row]

    guard let selectedImage = UIImage(named: selectedImageName) else {
        return
    }

    let composer = TWTRComposer()

    composer.setText("Love this restaurant!")
    composer.setImage(selectedImage)

    composer.show(from: self, completion: { (result) in
        if (result == .done) {
            print("Successfully composed Tweet")
        } else {
            print("Cancelled composing")
        }
    })

}
```

To let users compose a tweet, you just need to create an instance of `TWTRComposer`. Optionally, you can set the initial text and image. In the code above, we set the initial image to the image of the selected restaurant. Lastly, we call the `show` method to bring up the composer. That's all you need to do. The Twitter Kit will automatically check if the user has logged in to Twitter. If not, it will ask the user for username and password. The composer interface will only be displayed when the user has successfully logged into Twitter.

That's it! You can now test the app using the built-in simulator or deploy it to your device.

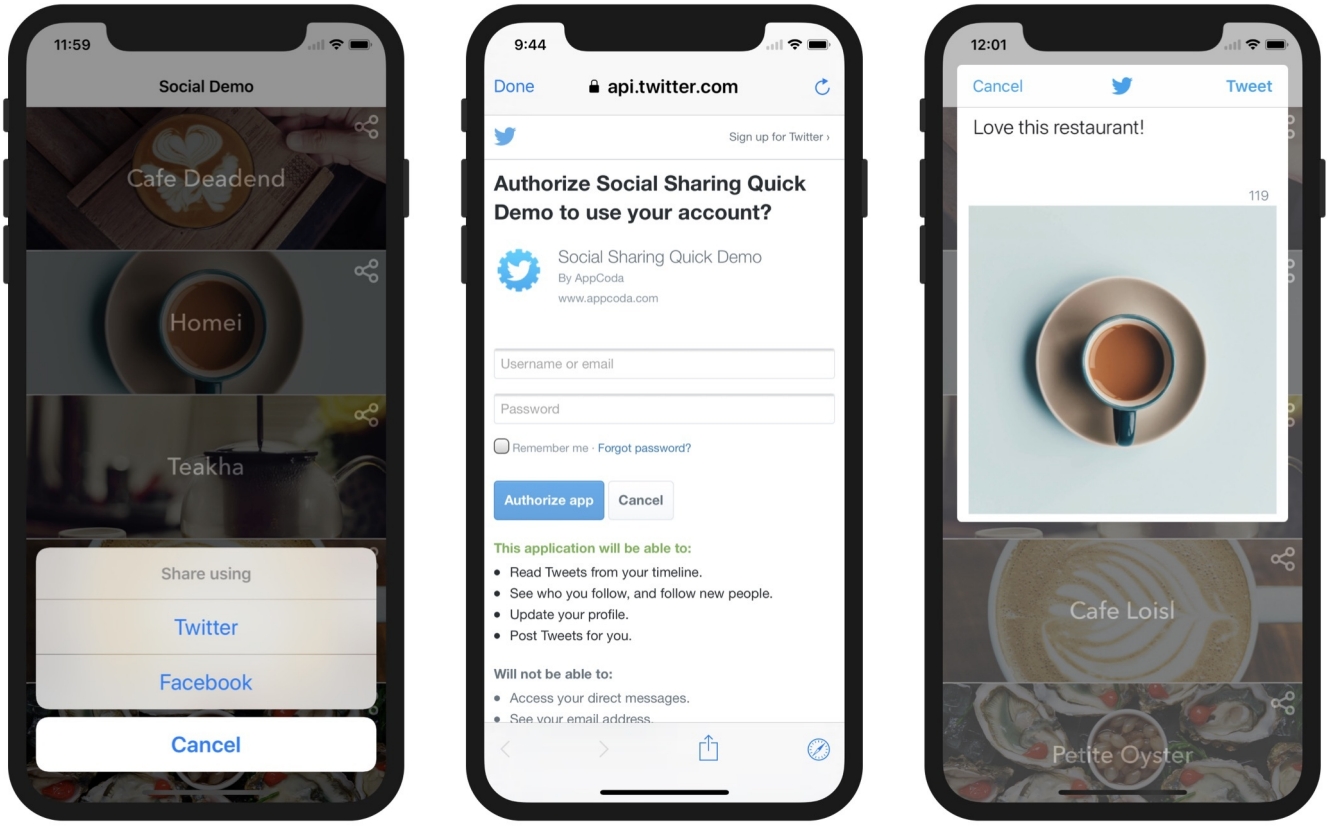


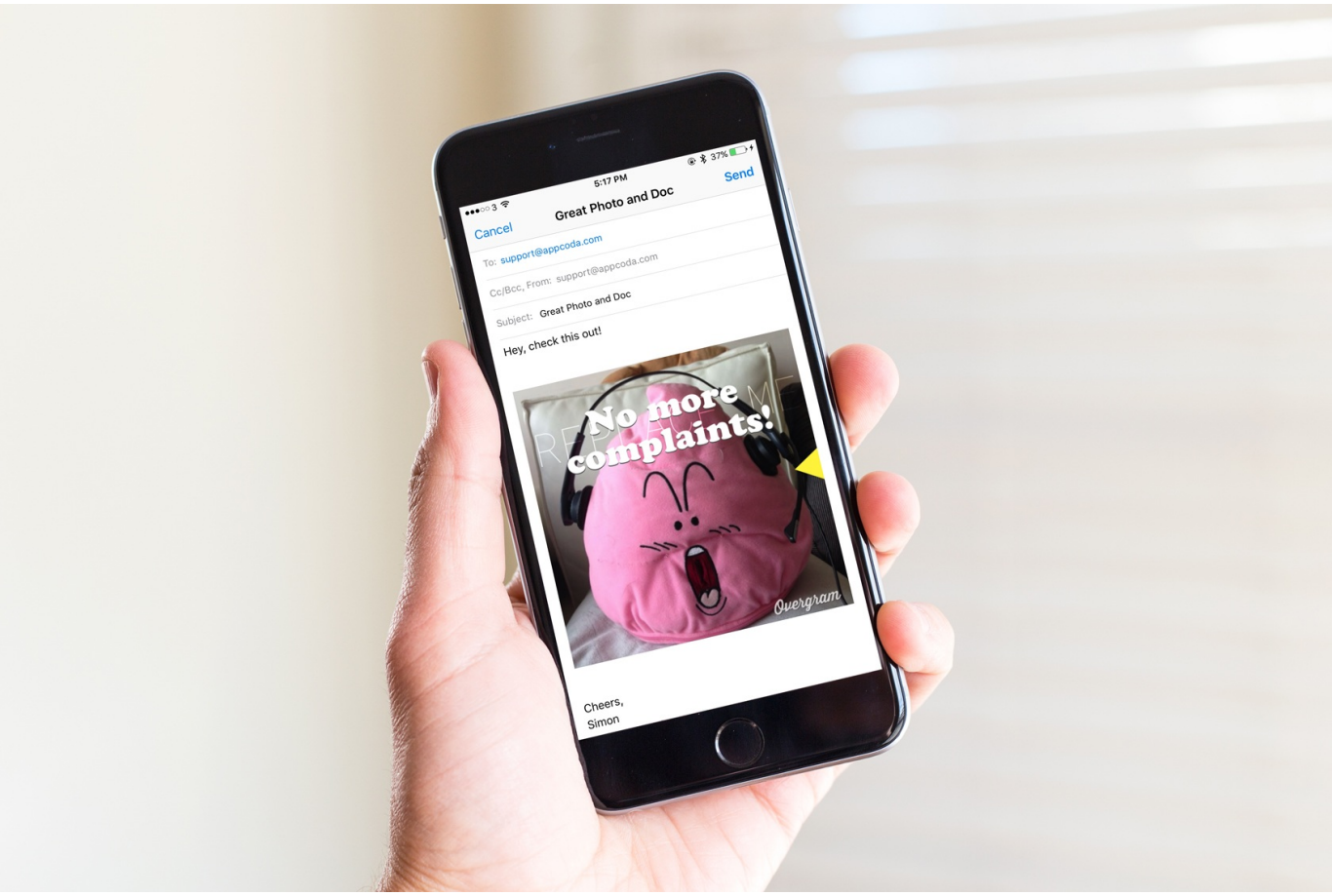
Figure 5.11. Fill in the app information

Summary

With the demise of the Twitter and Facebook integration from the Social framework, it takes you extra to integrate with these social network services. However, as you can see from this chapter, the procedures and APIs are not complicated. If you're building your app, there is no reason why you shouldn't incorporate these social features.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/SocialSharingDemo.zip.

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 emails 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 http://www.appcoda.com/resources/swift5/EmailAttachmentStarter.zip 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 on the main screen. Now, we'll continue to implement the email feature.

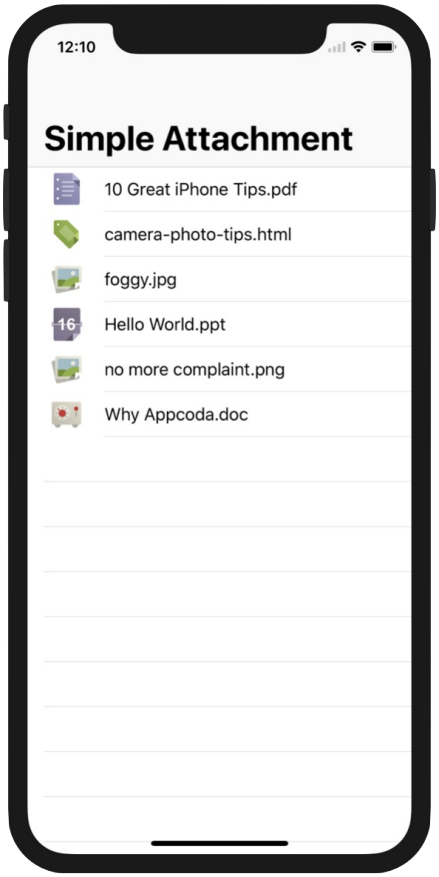


Figure 6.1. The demo app showing a list of attachments

Creating Email Using the MessageUI Framework

To present a standard composition interface for email in your app, you will need to use the APIs provided by the Message UI framework. Here is how you use the framework to let you compose an email without leaving your app:

- Create an instance of `MFMailComposeViewController` and display it inside your app. The `MFMailComposeViewController` class provides a standard user interface for managing and sending an email. The UI is exactly as the one in the stock Mail app.
- Implement the `MFMailComposeViewControllerDelegate` protocol to manage the mail composition. Say, what happens if the user fails to send the email? Also, it is your responsibility to dismiss the mail composer interface.

Okay, now let's go into the implementation.

First, import the `MessageUI` framework at the very beginning of `AttachmentTableViewController.swift` :

```
import MessageUI
```

Next, we will use an extension to adopt and implement the `MFMailComposeViewControllerDelegate` protocol. Your code will be like this:

```
extension AttachmentTableViewController: MFMailComposeViewControllerDelegate {
    func mailComposeController(_ controller: MFMailComposeViewController, didFinis
hWith result: MFMailComposeResult, error: Error?) {

        switch result {
        case MFMailComposeResult.cancelled:
            print("Mail cancelled")
        case MFMailComposeResult.saved:
            print("Mail saved")
        case MFMailComposeResult.sent:
            print("Mail sent")
        case MFMailComposeResult.failed:
            print("Failed to send: \(error?.localizedDescription ?? "")")
        }

        dismiss(animated: true, completion: nil)
    }
}
```

The `mailComposeController(_:didFinishWith:error:)` method is called when a user dismisses the mail composer. There are a few scenarios it may happen when the mail interface is dismissed:

- The operation is cancelled.
- The email is saved as a draft.
- The email has been sent.
- The email is failed to send.

The `result` variable stores one of the possible scenarios at the time the mail composer is dismissed. We implement the method to handle the result. For demo purposes, however, 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.

Next, declare an enumeration for the MIME types in the `AttachmentTableViewController` 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.lowercased() {
        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. graphics) 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. Therefore, 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 (e.g. jpg) and look up for the corresponding case. Later, we will use this enumeration when creating the `MFMailComposeViewController` object.

Now, create the methods for displaying the mail composer. Insert the following code in the same class:

```
func showEmail(attachment: String) {

    // Check if the device is capable to send email
    guard MFMailComposeViewController.canSendMail() else {
        print("This device doesn't allow you to send mail.")
        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 = attachment.components(separatedBy: ".")
    let filename = fileparts[0]
    let fileExtension = fileparts[1]

    // Get the resource path and read the file using NSData
    guard let filePath = Bundle.main.path(forResource: filename, ofType: fileExten
sion) else {
        return
```

```
    }

    // Get the file data and MIME type
    if let fileData = try? Data(contentsOf: URL(fileURLWithPath: filePath)),
        let mimeType = MIMEType(type: fileExtension) {

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

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

The `showEmail` method takes an attachment, which is the file name of the 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.rawValue, fileName: fi
lename)
```

The method accepts three parameters:

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

First, we determine the path of the given attachment by using the `path(forResource:ofType:)` method of `Bundle`. In iOS, a `Bundle` object represents a location in the file system of a resource group. In general, you use the main bundle to locate the directory of the current application executable. Since our resource files are embedded in the app, we retrieve the main bundle object, and then call the `path(forResource:ofType:)` 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 = try? Data(contentsOf: URL(fileURLWithPath: filePath)),
    let mimeType = MIMEType(type: fileExtension) {

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

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

Base on the file path, we instantiate a `Data` object. The initialization of `Data` may throw an exception. Here we use the `try?` keyword to handle an error by converting it to an optional value. In other words, if there are any problems loading the file, a `nil` value will be returned.

Once we initialized the `Data` object, we determine the MIME type of the given file with respect to its file extension. As you can see from the above code, you are allowed to combine multiple `if let` statements into one. Multiple optional bindings are separated by commas.

```
if let fileData = try? Data(contentsOf: URL(fileURLWithPath: filePath)),
    let mimeType = MIMEType(type: fileExtension)  {

}
```

Once we successfully initialized the file data and MIME type, we call the `addAttachmentData(_:mimeType:fileName:)` method to attach the file and then present the mail composer.

We're almost ready to test the app. The app should bring up the mail interface when any of the files are selected. Thus, the last thing is to add the `tableView(_:didSelectRowAt:)` method:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexP
ath) {
    let selectedFile = filenames[indexPath.row]
    showEmail(attachment: selectedFile)
}
```

You're good to go. Compile and run the app on a real iOS device (NOT the simulators). Tap a file and the app should display the mail interface with your selected file attached.

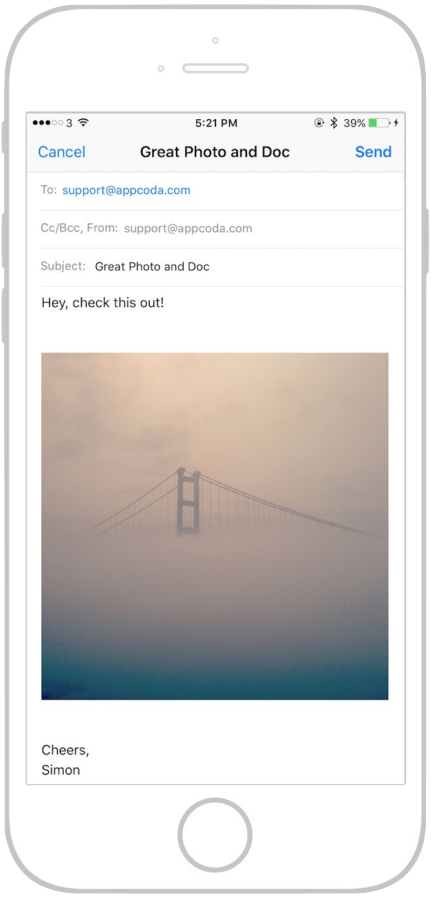
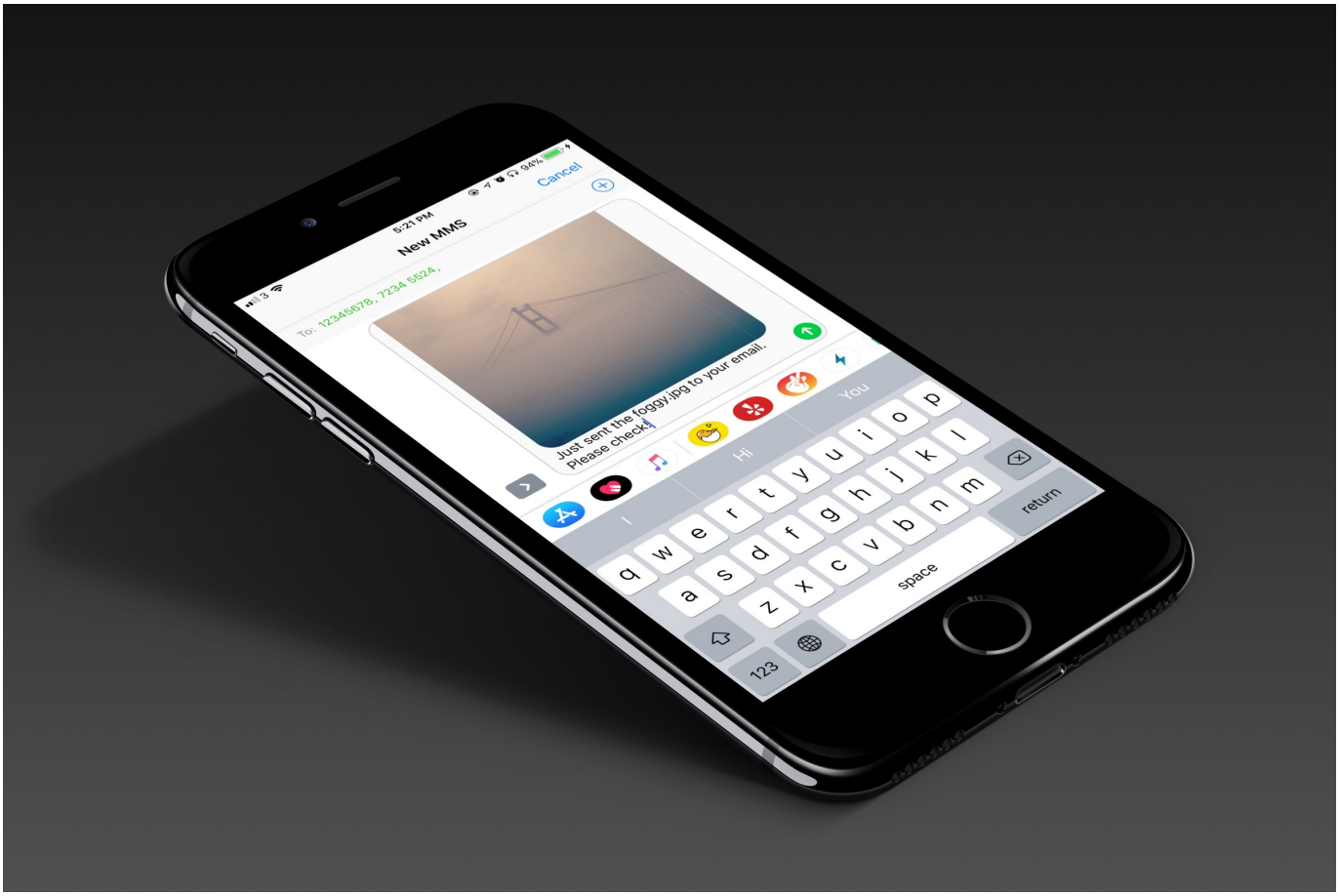


Figure 6.2. Displaying the mail interface inside the demo app

For reference, you can download the full source code from http://www.appcoda.com/resources/swift5/EmailAttachment.zip.

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.

> **Note:** If you haven't read the previous chapter, I highly recommend you to take a look.

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.

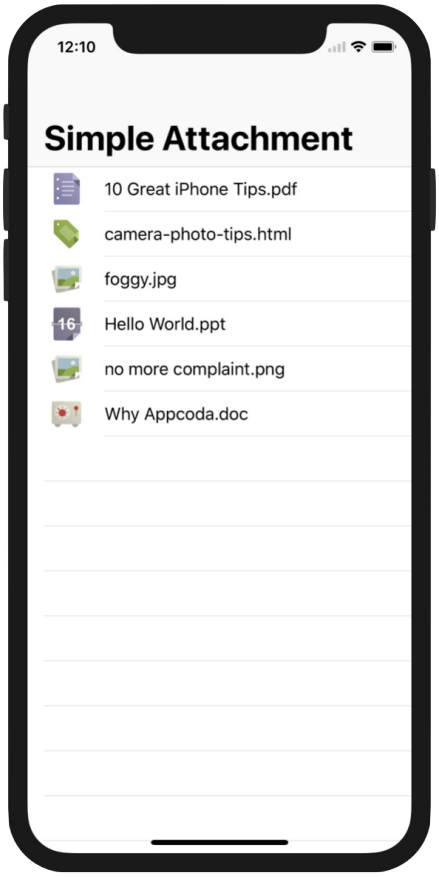


Figure 7.1. The demo app

Getting Started

To save you time from creating the Xcode project from scratch, you can download the project template from http://www.appcoda.com/resources/swift5/SMSDemoStarter.zip to begin with. I have pre-built the storyboard and already loaded the table data for you.

Implementing the Delegate

Open the `AttachmentTableViewController.swift` file. Add the following code to import the `MessageUI` framework:

```
import MessageUI
```

Similar to what we have done in the previous chapter, in order to use the message composer, we have to adopt the `MFMessageComposeViewControllerDelegate` protocol and implement the following method:

```
func messageComposeViewController(_ controller: MFMessageComposeViewController, di
dFinishWith result: MessageComposeResult)
```

The `MFMessageComposeViewControllerDelegate` protocol defines the 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.

Again, we will implement the protocol by using an extension. Insert the following code in `AttachmentTableViewController.swift`:

```
extension AttachmentTableViewController: MFMessageComposeViewControllerDelegate {
    func messageComposeViewController(_ controller: MFMessageComposeViewController
, didFinishWith result: MessageComposeResult) {

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

        case MessageComposeResult.failed:
            let alertMessage = UIAlertController(title: "Failure", message: "Faile
d to send the message.", preferredStyle: .alert)
            alertMessage.addAction(UIAlertAction(title: "OK", style: .default, han
dler: nil))
            present(alertMessage, animated: true, completion: nil)

        case MessageComposeResult.sent:
            print("SMS sent")

        }

        dismiss(animated: 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 `tableView(_:didSelectRowAtIndexPath:)` method in the `AttachmentTableViewController` class:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexP
ath) {
    let selectedFile = filenames[indexPath.row]
    sendSMS(attachment: 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: "Y
our device is not capable of sending SMS.", preferredStyle: .alert)
        alertMessage.addAction(UIAlertAction(title: "OK", style: .default, handler
: nil))
        present(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 ch
eck!"

    // Present message view controller on screen
    present(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 or you're testing the app on the simulator? 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 the phone number of a couple of recipients in the text message and set the message body.

With the content ready, you can invoke `present(_:animated:completion:)` to bring up the message composer. That's it! Simple and easy.

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.

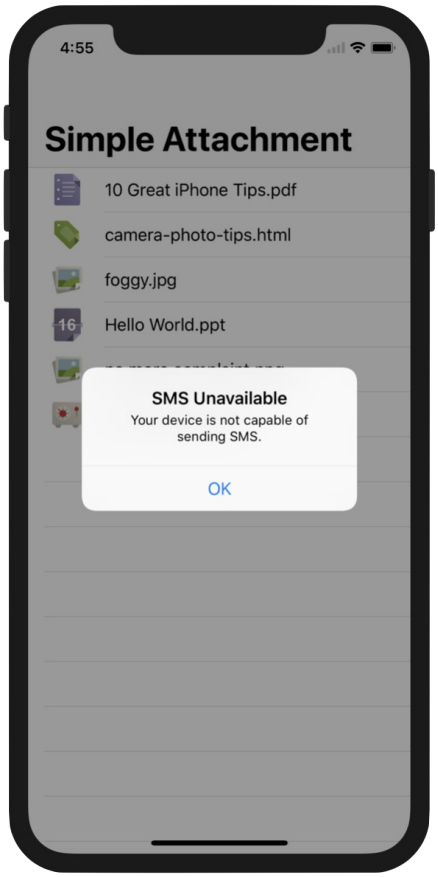


Figure 7.2. Running the app on the built-in simulator

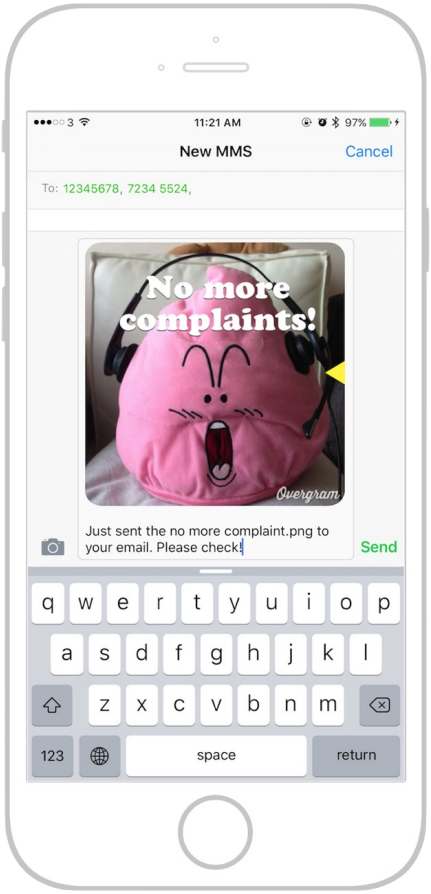
Sending MMS

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.components(separatedBy: ".")
let filename = fileparts[0]
let fileExtension = fileparts[1]
let filePath = Bundle.main.path(forResource: filename, ofType: fileExtension)
let fileUrl = NSURL.fileURL(withPath: filePath!)
messageController.addAttachmentURL(fileUrl, withAlternateFilename: nil)
```

Just add the lines above in the `sendSMS` method and insert them before calling the `present` method. The code is self explanatory. We get the selected file and retrieve the actual file path using the `path(forResource:ofType:)` method of `Bundle`. Lastly, we add the file using the `addAttachmentURL(_:withAlternateFilename:)` 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. 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:

```
if let messageUrl = URL(string: "sms:123456789&body=Hello") {
    UIApplication.shared.open(messageUrl, options: [:], completionHandler: nil)
}
```

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 it using Safari. If you want to open the Messages app, you can use the SMS URL scheme and specify the recipient. Optionally, you can specify the default content in the `body` parameter.

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 http://www.appcoda.com/resources/swift5/SMSDemo.zip.

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 `MKDirections` API. After going through the chapter, you will learn the following stuff:

- How to get the current user's location
- How to use `#available` to handle multiple versions of APIs
- How to compute the route and draw it on the map
- How to use the segmented control
- How to retrieve the route steps and display the detailed driving/walking instructions

The Sample Route App

I have covered the basics of the `MapKit` framework in the Beginning iOS 13 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 MKDirections API, we'll build a simple map app. You can start with this project template (http://www.appcoda.com/resources/swift5/MapKitDirectionStarter.zip).

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. If you have read our beginner book, 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.

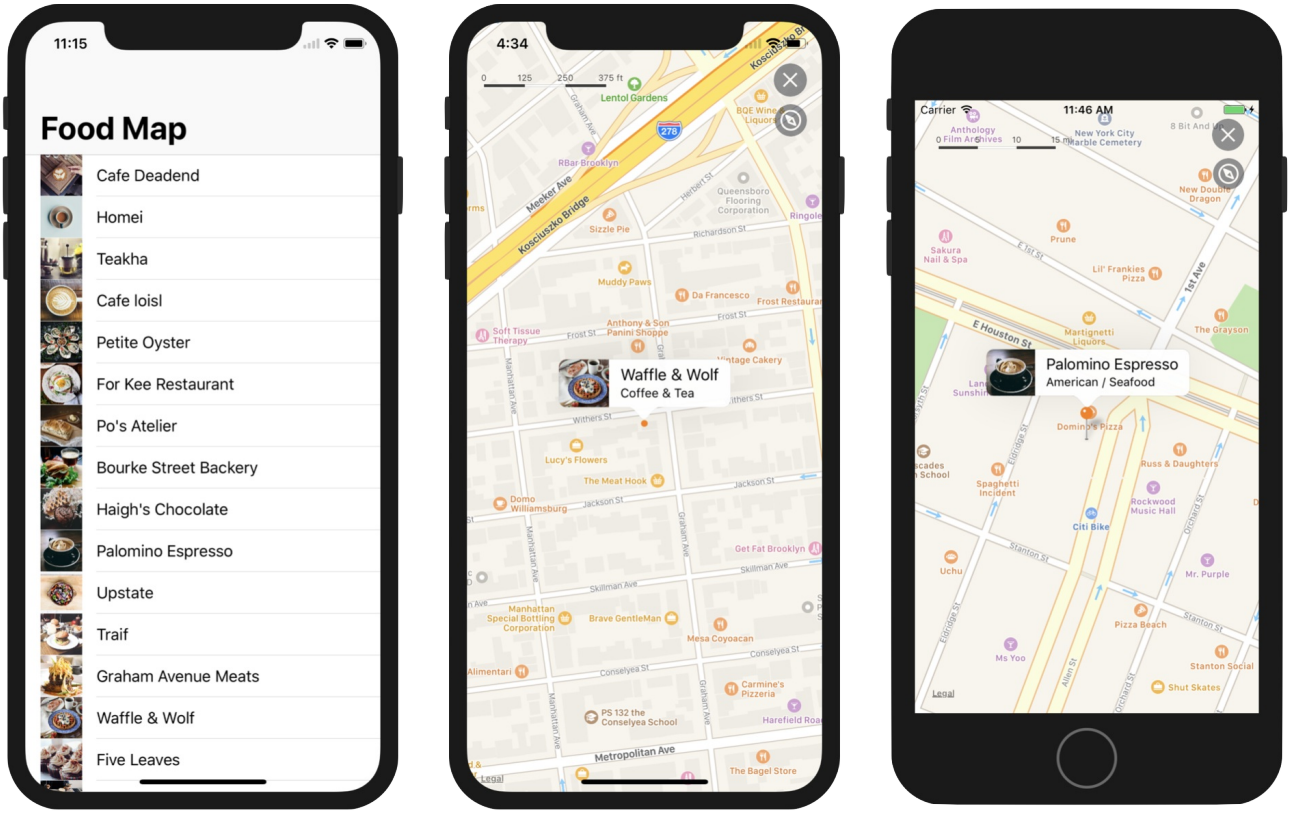


Figure 8.1. The Food Map app running on iOS 10 and 11

There is one thing I want to point out. If you look into the `MapViewController` class, you will find these lines of code:

```
if #available(iOS 11.0, *) {
    var markerAnnotationView: MKMarkerAnnotationView? = mapView.dequeueReusableAnn
otationView(withIdentifier: identifier) as? MKMarkerAnnotationView

    if markerAnnotationView == nil {
        markerAnnotationView = MKMarkerAnnotationView(annotation: annotation, reus
eIdentifier: identifier)
        markerAnnotationView?.canShowCallout = true
    }

    markerAnnotationView?.glyphText = "   "
    markerAnnotationView?.markerTintColor = UIColor.orange

    annotationView = markerAnnotationView

} else {

    var pinAnnotationView: MKPinAnnotationView? = mapView.dequeueReusableAnnotatio
nView(withIdentifier: identifier) as? MKPinAnnotationView

    if pinAnnotationView == nil {
        pinAnnotationView = MKPinAnnotationView(annotation: annotation, reuseIdent
ifier: identifier)
        pinAnnotationView?.canShowCallout = true
        pinAnnotationView?.pinTintColor = UIColor.orange
    }

    annotationView = pinAnnotationView
}
```

The current project is configured to run on iOS 10.0 or up. However, some of the APIs are available for the iOS 11 SDK or later. For example, the `MKMarkerAnnotationView` class was first introduced in iOS 11. On iOS 10, we will fallback to use the `MKPinAnnotationView` class. You can run the demo app on iOS 10 simulator and see what you get.

Similar to this project, if your app is going to support both iOS 10 and 11, you will need to check the OS version before calling some newer APIs. Otherwise, this will cause errors when the app runs on older versions of iOS.

Swift has a built-in support for API availability checking. You can easily define an availability condition such that the block of code will only be executed on certain iOS versions. You use the `#available` keyword in an `if` statement. In the availability condition, you specify the OS versions (e.g. iOS 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 11 (or up).

Creating an Action Method for the Direction Button

Now, open the Xcode project and go to `Main.storyboard`. The starter project already comes with the direction button, but it is not working yet.

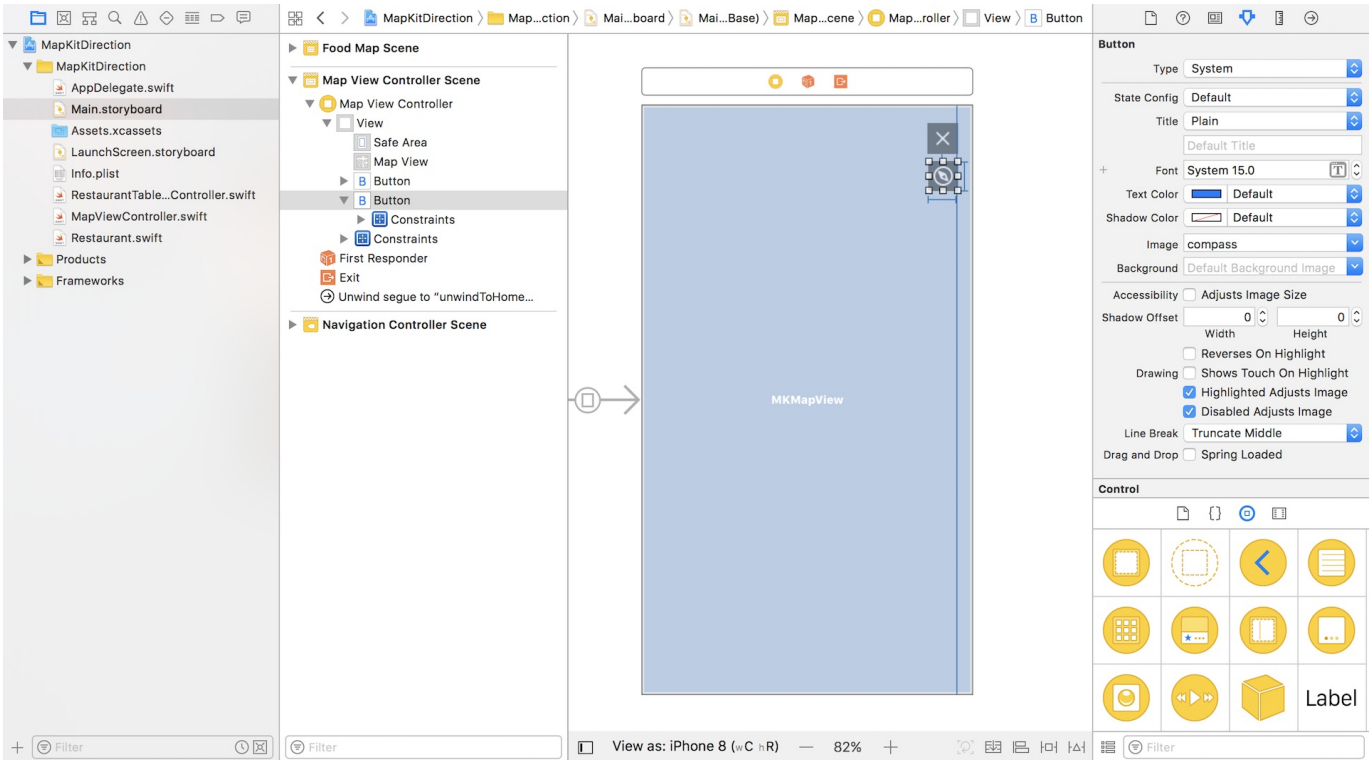


Figure 8.2. The direction button in maps

What we are going to do is to implement this button. When a user taps the button, it shows the user's current location and displays the directions to the selected restaurant.

The map view controller has been associated with the `MapViewController` class. Now, create an empty action method named `showDirection` in the class. We'll provide the implementation in the later section.

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

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 `showDirectionWithSender:` to connect with the action method.

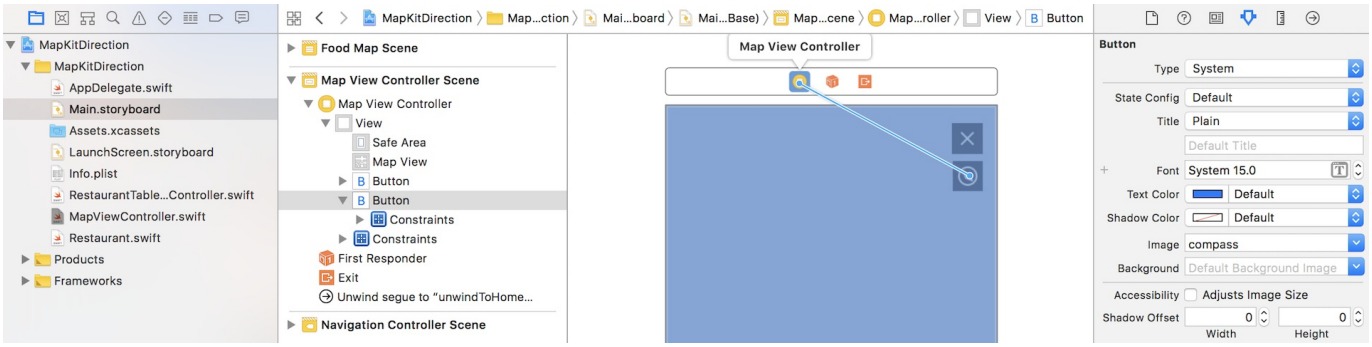


Figure 8.3. Connecting the direction button with the action method

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 will not work as expected. The app doesn't show your current location.

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.

In this project, let's add the `NSLocationWhenInUseUsageDescription` key in `Info.plist`. Select the file and right click any blank. Choose `Add Row` in the popover menu. For the key, set it to `Privacy - Location When in Use Usage Description`, which is actually the `NSLocationWhenInUseUsageDescription` key. For the value, you need to specify the reason (e.g. *We need to find out your current location in order to compute the route.*)

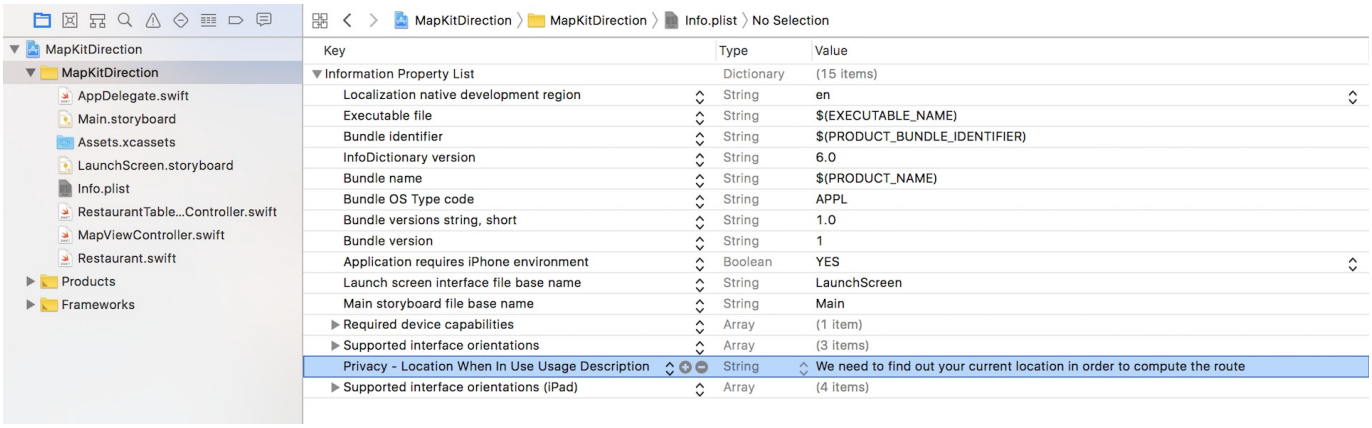


Figure 8.4. Adding a required key to get the user's approval for accessing the user's location

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.

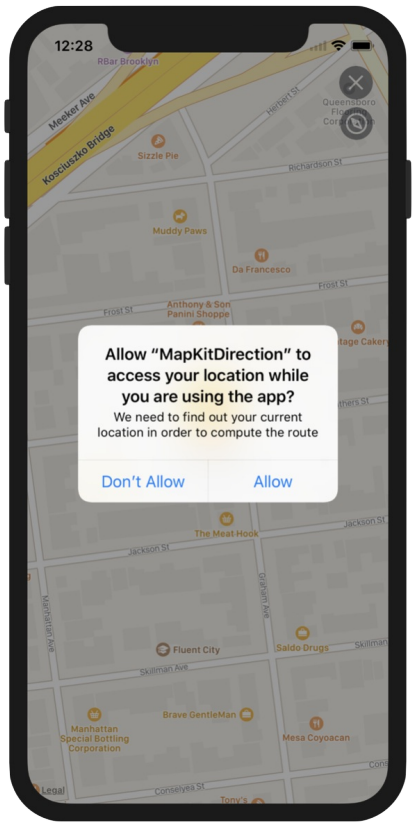


Figure 8.5. When the app launches, you'll be prompted to authorize location services

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 on 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.

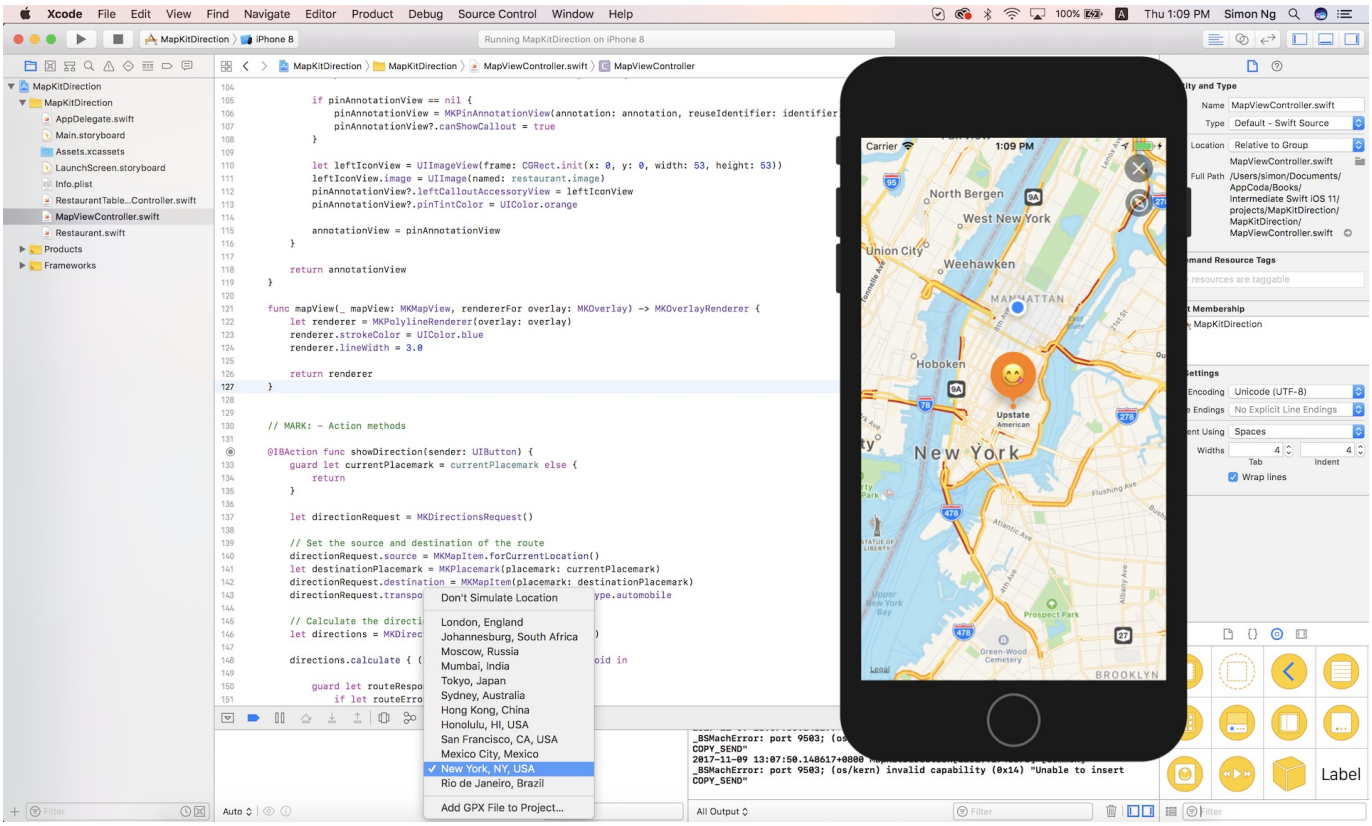


Figure 8.6. Simulating the user's current location

Using MKDirections API to Get the Route info

With the user location enabled, we move on to 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. A placemark in iOS stores information such as country, state, city and street address for a specific latitude and longitude.

In the starter project, we already retrieve the placemark object of the selected restaurant. In the `viewDidLoad` method, you should be able to locate the following line:

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

Next, add the following code right below it to set the value of `currentPlacemark`:

```
self.currentPlacemark = placemark
```

Next, we'll implement the `showDirection` method and use the `MKDirections` 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.forCurrentLocation()
    let destinationPlacemark = MKPlacemark(placemark: currentPlacemark)
    directionRequest.destination = MKMapItem(placemark: destinationPlacemark)
    directionRequest.transportType = MKDirectionsTransportType.automobile

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

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

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

            return
        }

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

    }
}
```

At the beginning of the method, we make sure if `currentPlacemark` contains a value using a `guard` statement. Otherwise, we just skip everything.

To request directions, we first create an instance of `MKDirectionsRequest`. 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 `MKDirectionsRequest` object created, we instantiate an `MKDirections` object and call the `calculate(completionHandler:)` method. The method initiates an asynchronous request for directions and calls your completion handler when the request is completed. The `MKDirections` 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 `MKDirectionsResponse` object. `MKDirectionsResponse` 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 `MKDirectionsRequest` 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 `add(_:level:)` method of the `MKMapView` class. The detailed route geometry (i.e. `route.polyline`) is represented by an `MKPolyline` object. The `add(_:level:)` 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 a 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 `mapView(_:rendererFor:)` method which actually draws the route:

```
func mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) -> MKOverlayRen
derer {
    let renderer = MKPolylineRenderer(overlay: overlay)
    renderer.strokeColor = UIColor.blue
    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.

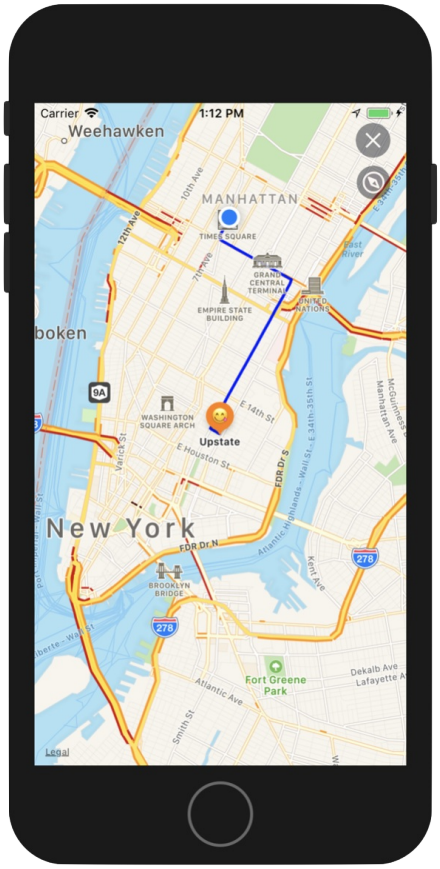


Figure 8.7. Tapping the direction button now shows the route

Scale the Map to Make the Route Fit Perfectly

You may 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. Can 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
self.mapView.setRegion(MKCoordinateRegionForMapRect(rect), animated: true)
```

And place them right after the following line of code:

```
self.mapView.add(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.

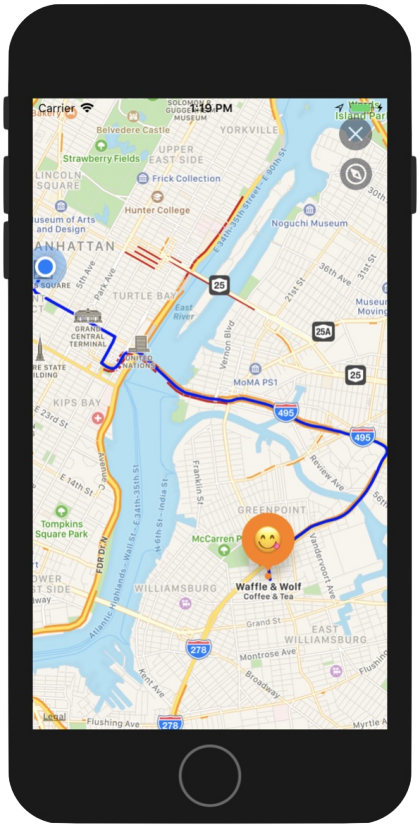


Figure 8.8. Display the route with auto scaling

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. Place it at the lower corner. 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`. Next, click the Pin button to add a couple of auto layout constraints. Your UI should look similar to figure 8.9.

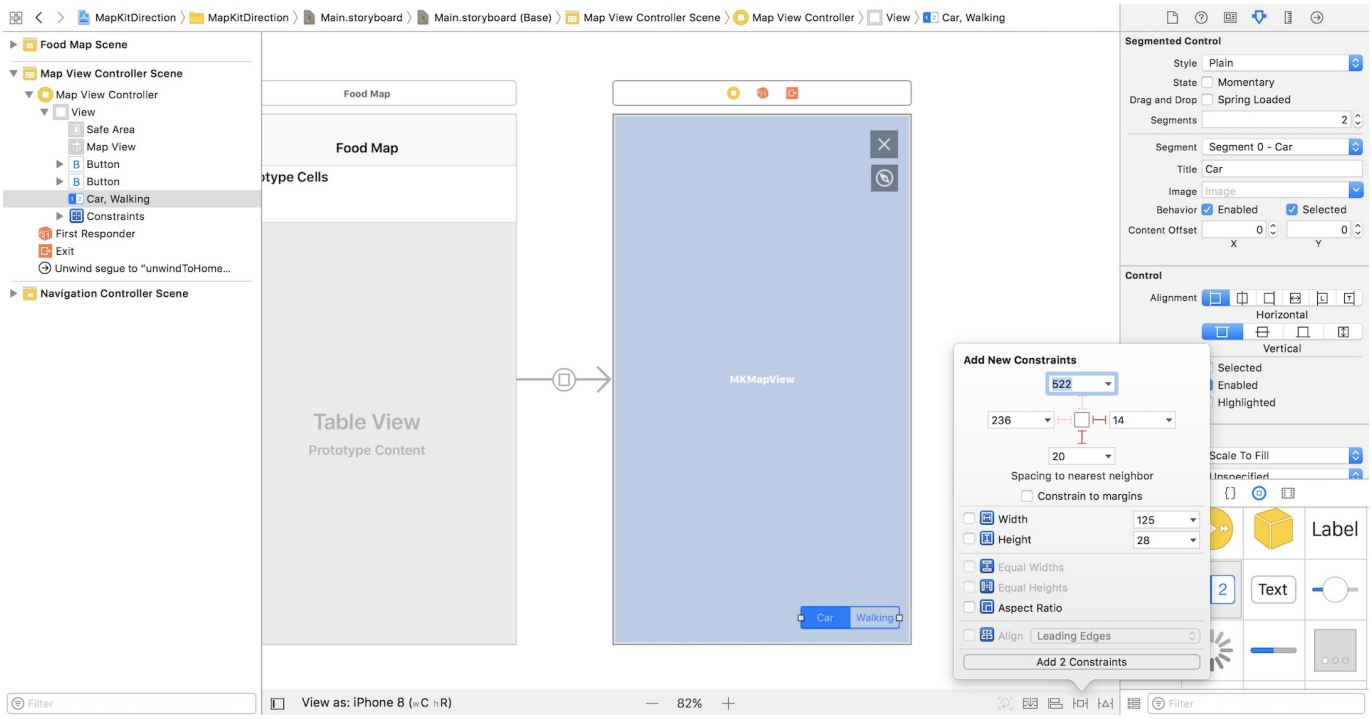


Figure 8.9. Adding the segment control to the map view controller

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.isHidden = 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
```

And replace `MKDirectionsTransportType.automobile` with `currentTransportType` like this:

```
directionRequest.transportType = 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 below. You can put the line of code in the `viewDidLoad` method:

```
segmentedControl.addTarget(self, action: #selector(showDirection), for: .valueChan
ged)
```

Here, we 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`). The `#selector` syntax was first introduced in Swift 2.2. It can check the method you want to call to make sure it actually exists. In other words, if you do not have the `showDirection` method in your code, Xcode will warn you.

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 = .automobile
case 1: currentTransportType = .walking
default: break
}

segmentedControl.isHidden = 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 in the `calculate(completionHandler:)` closure:

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

Place the line of code right before calling the `add(_:level:)` method. Your closure should look like this:

```
directions.calculate { (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.add(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 `mapView(_:rendererFor:)` method of the `MapViewController` class to display a different color. Simply change this line of code:

```
renderer.strokeColor = (currentTransportType == .automobile) ? UIColor.blue : UICo
lor.orange
```

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.

Figure 8.10. Showing the walking direction

Showing Route Steps

Now that you know how to display a route on 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 `MKRoute.Step` objects. An `MKRoute.Step` 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 to `Cell`. Next, embed the table view controller in a navigation controller, and change the title of the navigation bar to "Steps". Also, add a bar button item to the

navigation bar. In the Attributes inspector, change the system item option to `Done`.

Next, connect the map view controller with the new navigation controller using a segue. In the Document Outline of Interface Builder, control-drag the map view controller to the navigation controller. Select *present modally* for the segue type and set the segue's identifier to `showSteps`.

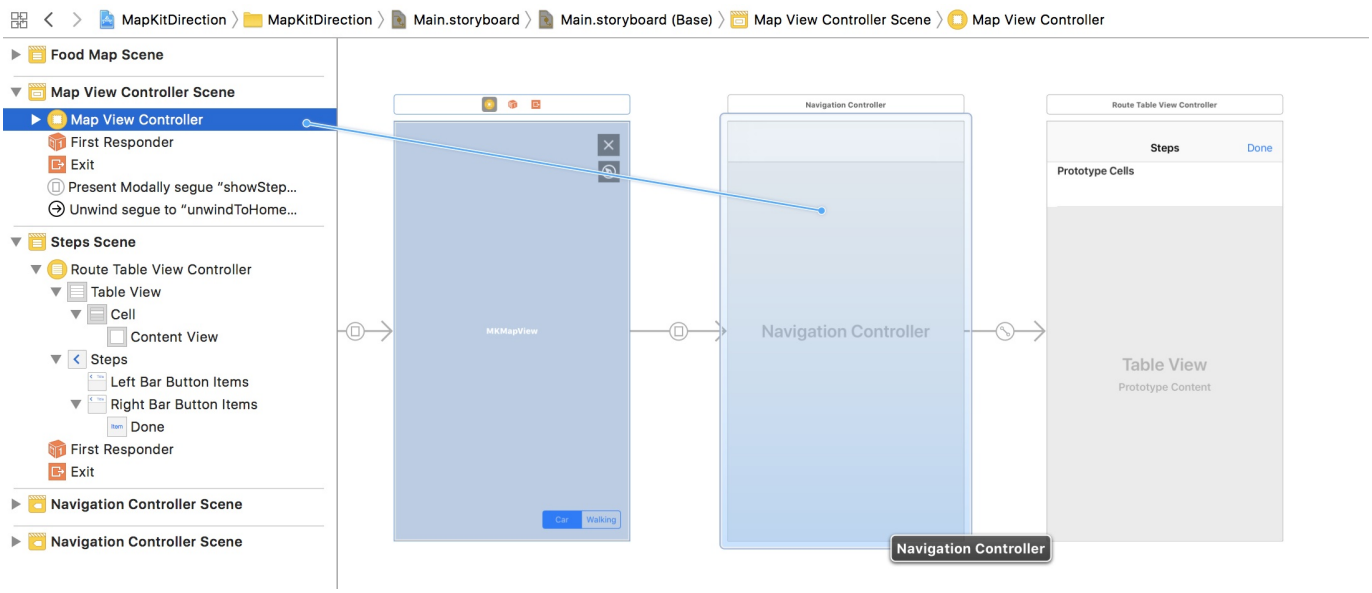


Figure 8.11. Connecting the map view controller with the navigation controller using a segue

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 `MKRoute.Step` objects. Each `MKRoute.Step` 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 to loop through all the `MKRoute.Step` 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 following method of your map view's delegate will be called when a user taps the accessory view:

```
optional func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView,
 calloutAccessoryControlTapped control: UIControl)
```

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
```

Next, declare an instance variable:

```
var routeSteps = [MKRoute.Step]()
```

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

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

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

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPat
h) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPat
h)

    // 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 `mapView(_:viewFor:)` method, insert the following line of code before `return annotationView`:

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

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

```
func mapView(_ mapView: MKMapView, annotationView view: MKAnnotationView, calloutA
ccessoryControlTapped control: UIControl) {

    performSegue(withIdentifier: "showSteps", sender: view)
}
```

In iOS, you're allowed to trigger a segue programmatically by calling the `performSegue(withIdentifier:sender:)` 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 bring up the Steps table view controller when the above `performSegue(withIdentifier:sender:)` method is called.

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

In the body of the `calculate(completionHandler:)` 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.calculate { (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.add(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 `performSegue(withIdentifier:sender:)` method like this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
    if segue.identifier == "showSteps" {
        let routeTableViewController = segue.destination.childViewControllers[0] 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.

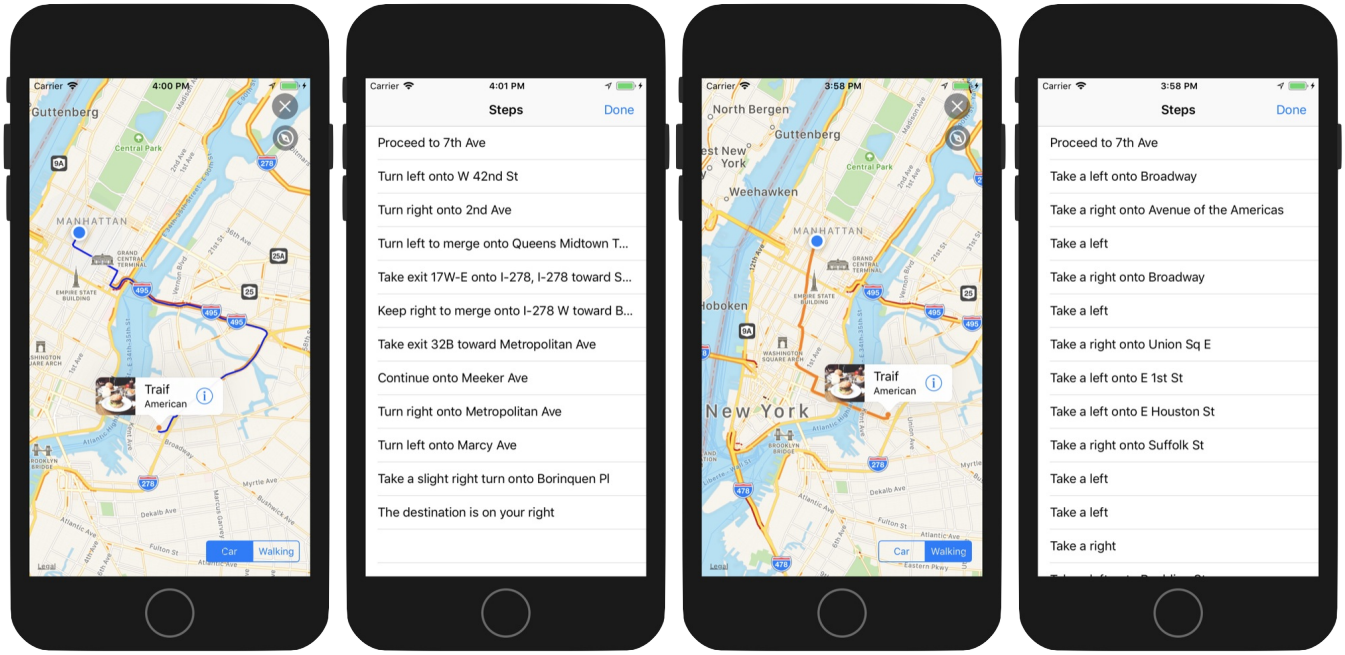


Figure 8.12. Tapping the annotation now shows you the list of steps to follow

But we still miss one thing. When you tap the *Done* button in the route table view controller, it doesn't dismiss the controller. To make it work, create an action method in the `RouteTableViewController` class:

```
@IBAction func close() {
    dismiss(animated: true, completion: nil)
}
```

Then connect the *Done* button with the `close()` method in the storyboard.

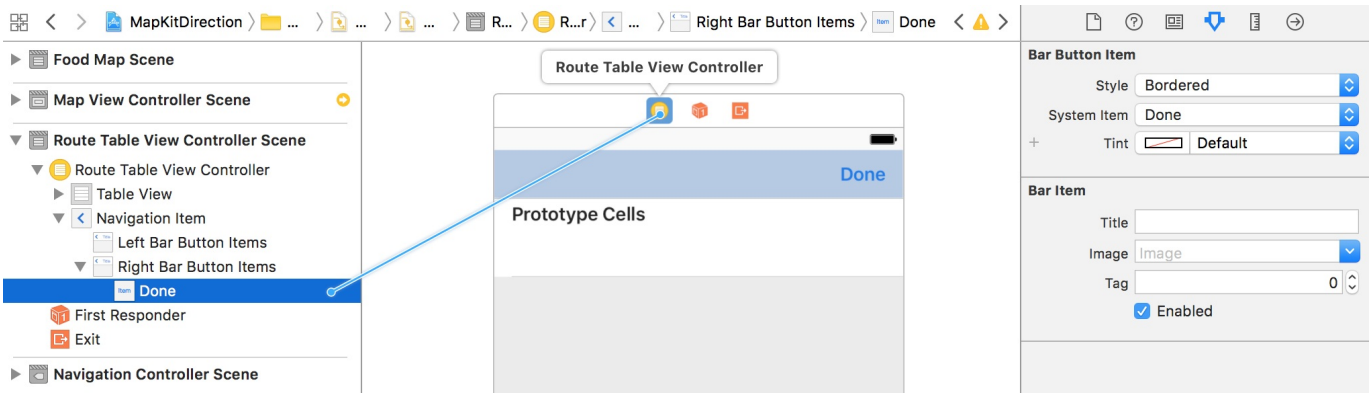


Figure 8.13. Connecting the Done button witht the action method

That's it. You can test the app again. Now you should be able to dismiss the route table controller when you tap the *Done* button.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/MapKitDirection.zip.

Chapter 9
Search for Nearby Points of Interest Using Local Search

The 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 `MKDirections` 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 http://www.appcoda.com/resources/swift5/LocalSearchStarter.zip. Unzip it and open the `MapKitDirection` project.

```
Note: The starter project is exactly the same as the final
project of the MapKit Direction demo, except that it includes a
"gingerbread". If you want to know how it works, please revisit
the previous chapter.
```

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. Set the image of the button to *gingerbread*. After that, add the spacing and size constraints. When tapped, this button will show the nearby restaurants. From now and onwards, I refer this button as the *Nearby* button.

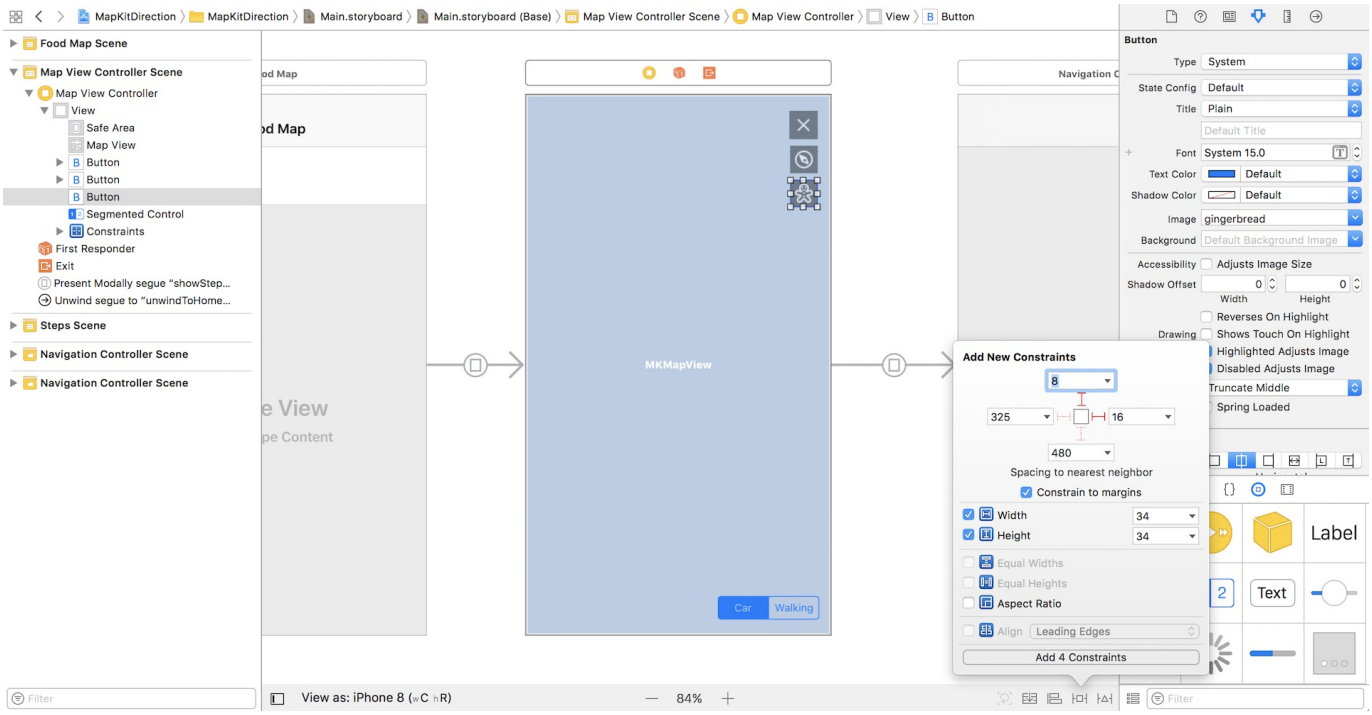


Figure 9.1. Adding a Nearby button to the map view controller

Search Nearby Restaurants and Adding annotations

Once you added the button, open the `MapViewController.swift` file. We will create an action method called `showNearby` for the *Nearby* button. In the implementation, we will search for nearby restaurants and pin the results on the map.

Insert the following code snippet in the class:

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

    let localSearch = MKLocalSearch(request: searchRequest)
    localSearch.start { (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.type` 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 `start(completionHandler:)` 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. To pin multiple annotations on maps, you call the `showAnnotations` method and pass it the array of `MKAnnotation` objects to pin.

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 `showNearbyWithSender:` action method.

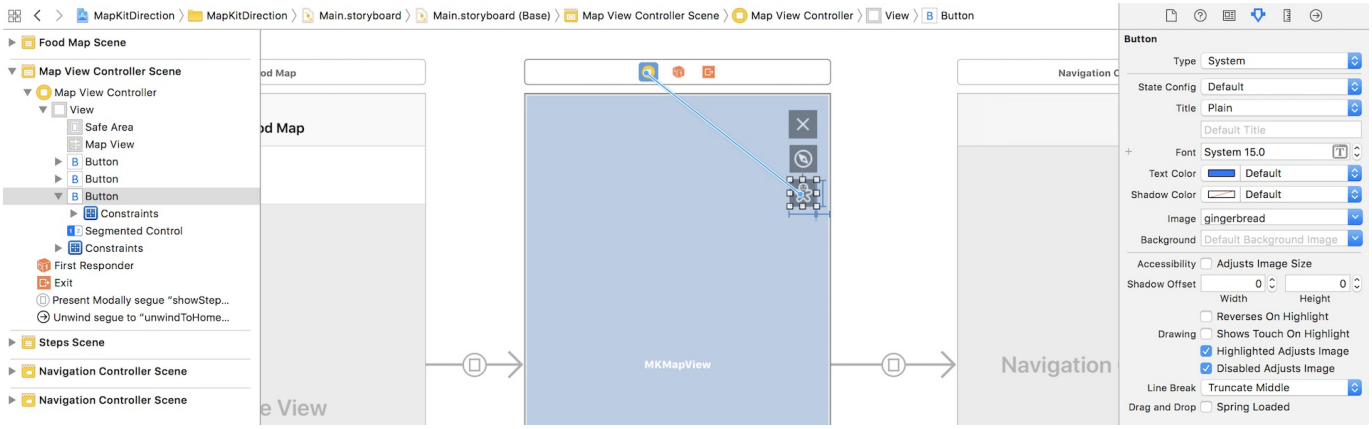


Figure 9.2. Associating the action method with the Nearby button

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.

Figure 9.3. The demo app now shows 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.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/LocalSearch.zip.

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 any 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 *RecordPro* (or any name you like). You can design a user interface like figure 10.1 on your own. However, to free you from setting up the user interface and custom classes, you can download the project template from http://www.appcoda.com/resources/swift5/RecordProStarter.zip. I've created the storyboard and custom classes for you. The user interface is very simple with three buttons: *record*, *stop* and *play*. It also has a timer to show the elapsed time during recording. The buttons have been connected to the corresponding action method in the `RecordProController` class, which is a subclass of `UIViewController`.

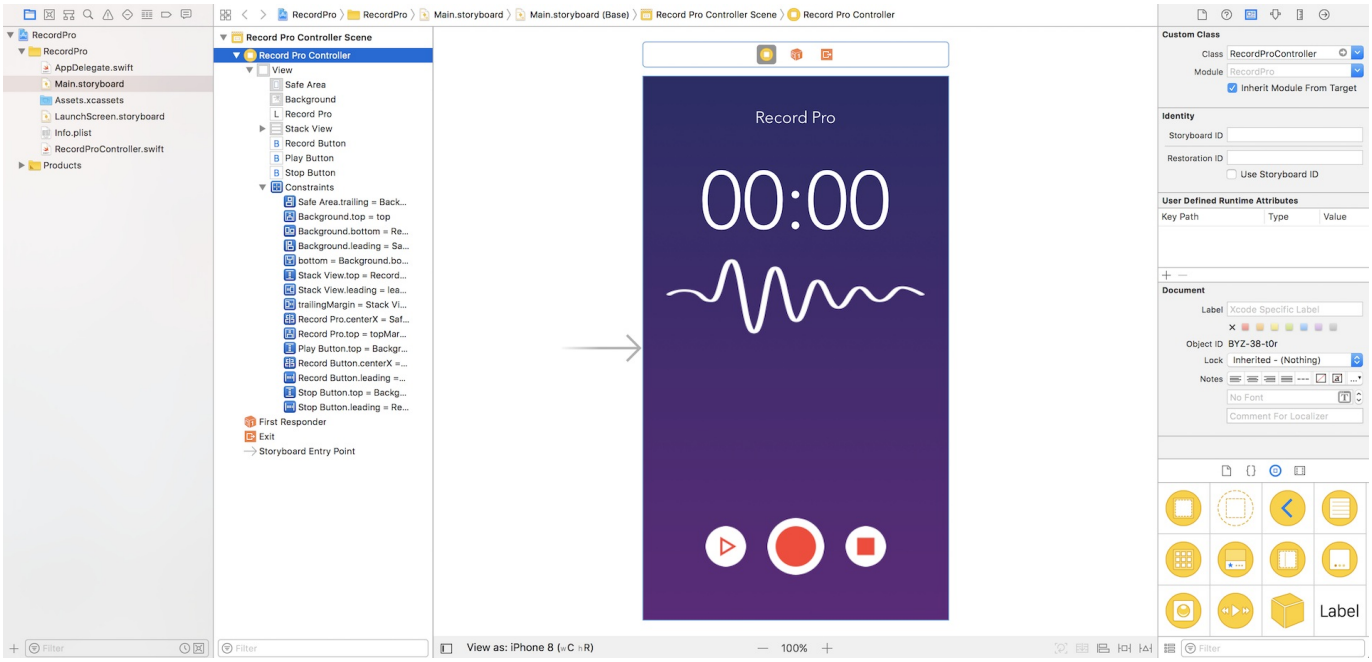


Figure 10.1. RecordPro Project

Before we move onto the implementation, let me give you a better idea of how the demo app works:

- When the user taps the *Record* button, the app starts the timer and begins to record the audio. The *Record* button is then replaced by a *Pause* button. If the user taps the Pause button, the app will pause the recording until the user taps the button again. In terms of coding, it invokes the `record` action method.
- When the user taps the *Stop* button, the app stops the recording. I have already connected the button with the `stop` action method in `RecordProController`.
- To play the recording, the user can tap the *Play* button, which is associated with the `play` method.

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 demo app, the `RecordProController` class serves as the delegate object. Therefore, we adopt the `AVAudioRecorderDelegate` protocol by using an extension like this:

```
extension RecordProController: AVAudioRecorderDelegate {

}
```

We will implement the option method of the protocol in a later section. For now, we just indicate `RecordProController` is responsible for adopting `AVAudioRecorderDelegate`.

Because the protocol 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 `RecordProController.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 create a private method called `configure()` to do the setup. Insert the code into the `RecordProController` class:

```
private func configure() {
    // Disable Stop/Play button when application launches
    stopButton.isEnabled = false
    playButton.isEnabled = false

    // Get the document directory. If fails, just skip the rest of the code
    guard let directoryURL = FileManager.default.urls(for: FileManager.SearchPathD
irectory.documentDirectory, in: FileManager.SearchPathDomainMask.userDomainMask).f
irst else {

        let alertMessage = UIAlertController(title: "Error", message: "Failed to g
et the document directory for recording the audio. Please try again later.", prefe
rredStyle: .alert)
        alertMessage.addAction(UIAlertAction(title: "OK", style: .default, handler
: nil))
```

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

            return
        }

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

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

        do {
            try audioSession.setCategory(.playAndRecord, mode: .default, options: [ .d
efaultToSpeaker ])

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

            // Initiate and prepare the recorder
            audioRecorder = try AVAudioRecorder(url: audioFileURL, settings: recorderS
etting)
            audioRecorder.delegate = self
            audioRecorder.isMeteringEnabled = true
            audioRecorder.prepareToRecord()

        } catch {
            print(error)
        }
    }
```

In the above code, we first disable both the *Stop* and *Play* buttons because we only let users record audio when the app is first launched. We then define the URL of the sound file for saving the recording.

The question is where to save the sound file and how can we get the file path?

My plan is to store the file in the document directory of the user. In iOS, you use `FileManager` to interact with the file system. The class provides the following method for searching common directories:

```
func urls(for directory: FileManager.SearchPathDirectory, in domainMask: FileManag
er.SearchPathDomainMask) -> [URL]
```

The method takes in two parameters: *search path directory* and *file system domain to search*. My plan is to store the sound file under the document directory of the user's home directory. Thus, we set the search path directory to the document directory (`FileManager.SearchPathDirectory.documentDirectory`) and the domain to search to the user's home directory (`FileManager.SearchPathDomainMask.userDomainMask`).

After retrieving the file path, we create the audio file URL and name the audio file `MyAudioMemo.m4a` . In case of failures, the app shows an alert message to the users.

Now that we've prepared the sound file URL, the next thing is to configure the audio session. What's the audio session for? iOS handles the 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 set the audio session category to `.playAndRecord` , 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 above, we use `recorderSetting` to store 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 ready 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 used a `try` keyword when we initialize the `AVAudioRecorder` instance and call the `setCategory` method of `audioSession`. Since the release of Swift 3, Apple changed most of the APIs in favor of the *do-try-catch* error handling model.

If the method call may throw an error, or the initialization may fail, you have to enclose it in a do-catch block like this:

```
do {
    try audioSession.setCategory(.playAndRecord, mode: .default, options: [ .defau
ltToSpeaker ])

     ...

    // Initiate and prepare the recorder
    audioRecorder = try AVAudioRecorder(url: audioFileURL, settings: recorderSetti
ng)

} catch {
    print(error)
}
```

In the `do` clause, you 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 embedded in an `Error` object.

Okay, the `configure()` method is ready. To trigger the configuration, insert the following line of code in the `viewDidLoad()` method:

```
configure()
```

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 further 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 *Pause* button. If the user taps the *Pause* button, the app will pause the audio recording until the button is tapped again. The audio recording will stop when the user taps the *Stop* button.

Now, update the `record` method like this:

```
@IBAction func record(sender: UIButton) {

    // Stop the audio player before recording
    if let player = audioPlayer, player.isPlaying {
        player.stop()
    }

    if !audioRecorder.isRecording {
        let audioSession = AVAudioSession.sharedInstance()

        do {
            try audioSession.setActive(true)

            // Start recording
            audioRecorder.record()

            // Change to the Pause image
            recordButton.setImage(UIImage(named: "Pause"), for: UIControl.State.no
rmal)
        } catch {
            print(error)
        }

    } else {
        // Pause recording
        audioRecorder.pause()

        // Change to the Record image
        recordButton.setImage(UIImage(named: "Record"), for: UIControl.State.norma
l)
    }

    stopButton.isEnabled = true
    playButton.isEnabled = false
}
```

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

If `audioRecorder` is not in the 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.

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

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

As you can see, the AVFoundation API is pretty easy to use. With a few lines of code, you can use the built-in microphone to record audio.

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

Using Microphone Without Users' Permission

If you can't wait to test your app, deploy and run it on a real iOS device. However, you will end up with an error. The app can't even start up properly. If you look into the console, the error message will give you some hints about the issue:

```
RecordPro[66275:16149656] [access] This app has crashed because it attempted to ac
cess privacy-sensitive data without a usage description.  The app's Info.plist mus
t contain an NSMicrophoneUsageDescription key with a string value explaining to th
e user how the app uses this data.
```

Since iOS 10, you can't access the microphone without asking for the user's permission. To do so, you need to add a key named `NSMicrophoneUsageDescription` in the `Info.plist` file and explain to the user why your app needs to use the microphone.

Now, open `Info.plist`, and then right click any blank area to open the popover menu. Choose `Add Row` to add a new entry. In the value field, you specify the reason why you need to use the microphone.

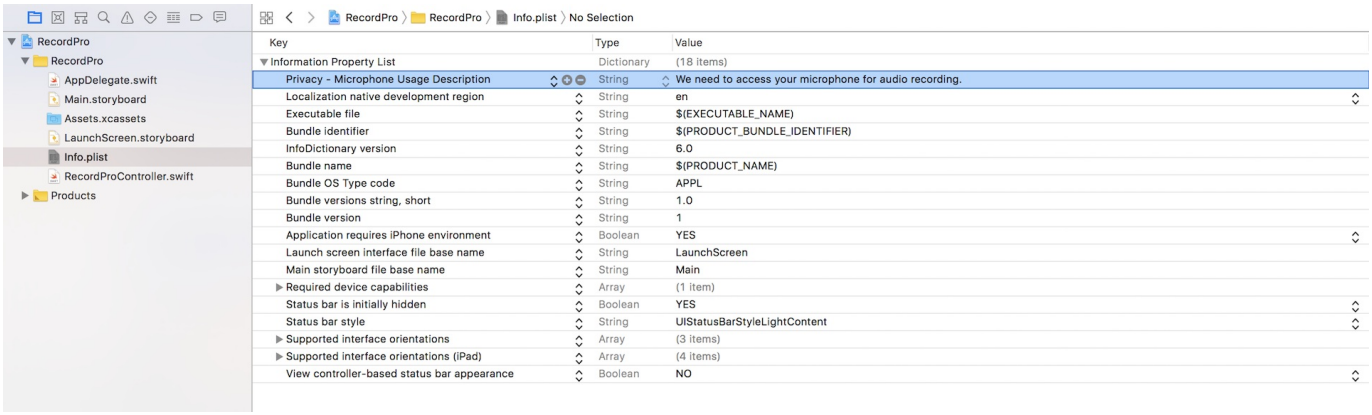


Figure 10.2. Create a new key in Info.plist to explain why you need to use the microphone

Once you add the reason, you can test the app on your device again. This time, the app should display a message (with the explanation you added before) asking for the user's permission for accessing the microphone. Remember to choose OK to authorize the access.

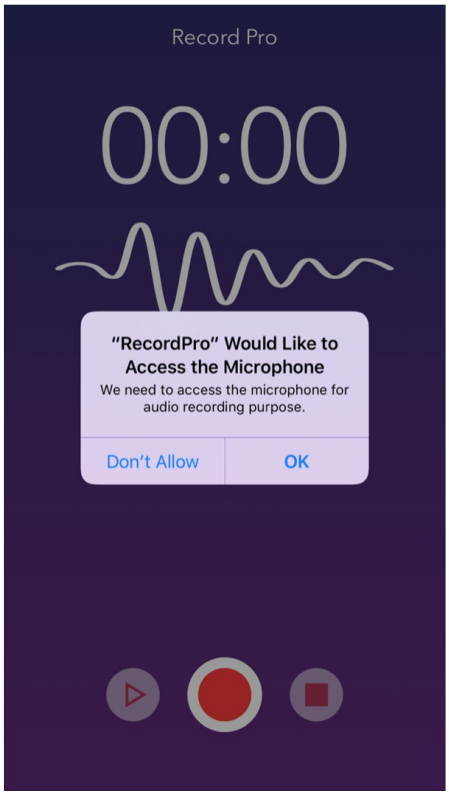


Figure 10.3. You must get the user's approval before accessing the microphone

Implementing the Stop Button

Let's continue to implement the rest of the action method.

The `stop` action method is called when the user taps the *Stop* button. This method is pretty simple. We first reset the state of the buttons and then call the `stop` method of the *AVAudioRecorder* object to stop the recording. Lastly, we deactivate the audio session. Update the `stop` action method to the following code:

```
@IBAction func stop(sender: UIButton) {
    recordButton.setImage(UIImage(named: "Record"), for: UIControl.State.normal)
    recordButton.isEnabled = true
    stopButton.isEnabled = false
    playButton.isEnabled = true

    // Stop the audio recorder
    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, `RecordProController` is the delegate. The methods defined in the `AVAudioRecorderDelegate` protocol are optional. For demo purpose, we'll only implement the `audioRecorderDidFinishRecording(_:successfully:)` method to handle the completion of recording. Update the `RecordProController` extension like this:

```
extension RecordProController: AVAudioRecorderDelegate {
    func audioRecorderDidFinishRecording(_ recorder: AVAudioRecorder, successfully
 flag: Bool) {
        if flag {
            let alertMessage = UIAlertController(title: "Finish Recording", messag
e: "Successfully recorded the audio!", preferredStyle: .alert)
            alertMessage.addAction(UIAlertAction(title: "OK", style: .default, han
dler: nil))
            present(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 the 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 `RecordProController` class, edit the `play` action method using the following code:

```
@IBAction func play(sender: UIButton) {

    if !audioRecorder.isRecording {
        guard let player = try? AVAudioPlayer(contentsOf: audioRecorder.url) else
{
            print("Failed to initialize AVAudioPlayer")
            return
        }

        audioPlayer = player
        audioPlayer?.delegate = self
        audioPlayer?.play()
    }

}
```

The above code is very straightforward. We first initialize an instance of `AVAudioPlayer` with the URL of the audio file (`audioRecorder.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.

You may wonder what the keyword `try?` means. The initialization of `AVAudioPlayer` may throw an error. Normally, you can use the `do-try-catch` block when initializing the `AVAudioPlayer` instance like this:

```
do {
  ...

  audioPlayer = try AVAudioPlayer(contentsOf: recorder.url)

  ...
} catch {
  print(error)
}
```

In some cases, we may just want to ignore the error. So you can use `try?` to make things simpler without wrapping the statement with a `do-catch` block:

```
audioPlayer = try? AVAudioPlayer(contentsOf: recorder.url)
```

If the initialization fails, the error is handled by turning the result into an optional value. Hence, we use `guard` to check if the optional has a value.

Implementing the AVAudioPlayerDelegate Protocol

The delegate of an `AVAudioPlayer` object must adopt the `AVAudioPlayerDelegate` protocol. Again, `RecordProController` is set as the delegate, so create an extension of `RecordProController` to adopt the protocol:

```
extension RecordProController: AVAudioPlayerDelegate {
    func audioPlayerDidFinishPlaying(_ player: AVAudioPlayer, successfully flag: B
ool) {
        playButton.isSelected = false

        let alertMessage = UIAlertController(title: "Finish Playing", message: "Fi
nish playing the recording!", preferredStyle: .alert)
        alertMessage.addAction(UIAlertAction(title: "OK", style: .default, handler
: nil))
        present(alertMessage, animated: true, completion: nil)
    }
}
```

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.

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.

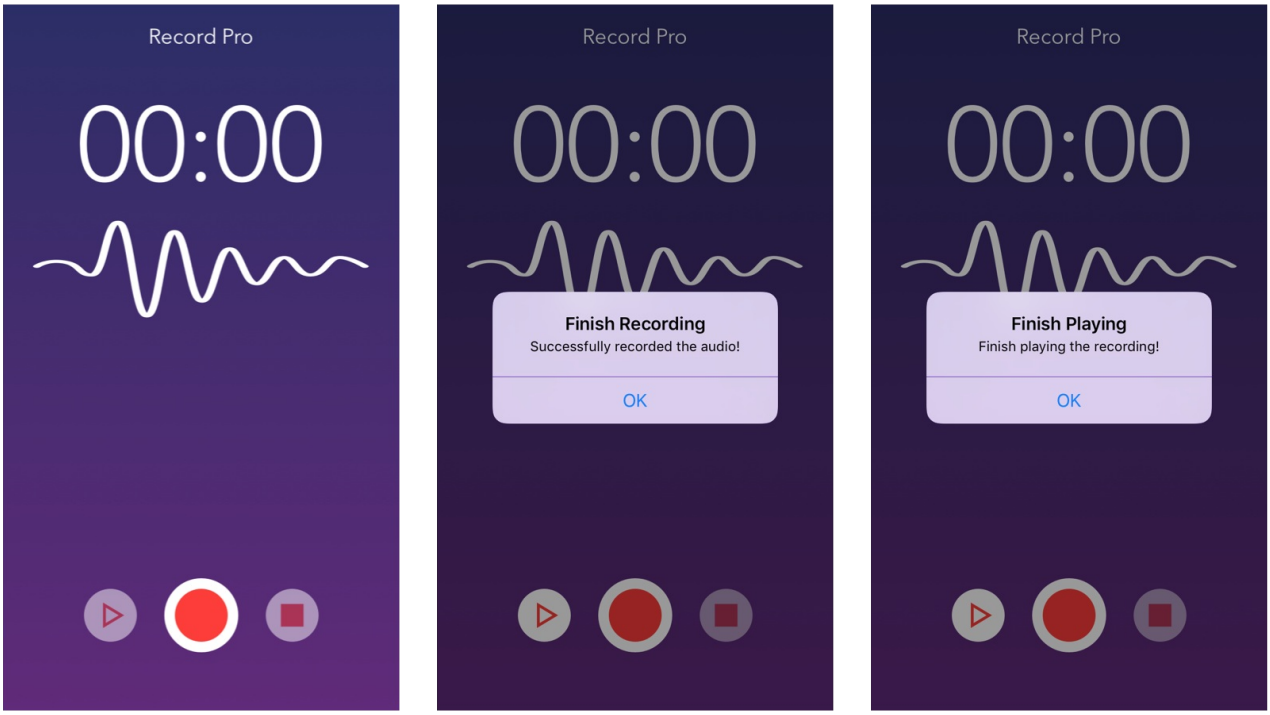


Figure 10.4. RecordPro Demo App

Implementing the Timer

Now that the audio recording and playback should work, there is still one thing missing. We haven't implemented the timer yet.

The time label should be updated every second to indicate the elapsed time of the recording and playback. To do so, we utilize a built-in class named `Timer` for the implementation. You can tell a `Timer` object to wait until a certain time interval has

elapsed and then run a block of code. In this case, we want the `Timer` object to execute the block of code every second, so we can update the time label accordingly.

With some ideas about the implementation, insert the following code in the `RecordProController` class:

```
private var timer: Timer?
private var elapsedTimeInSecond: Int = 0

func startTimer() {
    timer = Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true, block: { (t
imer) in
        self.elapsedTimeInSecond += 1
        self.updateTimeLabel()
    })

}

func pauseTimer() {
    timer?.invalidate()
}

func resetTimer() {
    timer?.invalidate()
    elapsedTimeInSecond = 0
    updateTimeLabel()
}

func updateTimeLabel() {

    let seconds = elapsedTimeInSecond % 60
    let minutes = (elapsedTimeInSecond / 60) % 60

    timeLabel.text = String(format: "%02d:%02d", minutes, seconds)
}
```

Here, we declare four methods to work with the timer. Let's begin with the `startTimer` method. As mentioned before, we utilize `Timer` to execute certain code every second. To create a `Timer` object, you can use a method called

`scheduledTimer(withTimeInterval:repeats:block)`. In the above code, we set the time interval to one second and create a repeatable timer. In other words, the timer fires every second.

We have a `elapsedTimeInSecond` variable to keep track of the recording/playback time in second. Every time when the timer fires, the code block is executed. We increase the variable by one second and then call the `updateTimeLabel` method to update the label.

For a repeating timer, it is required to explicitly call the `invalidate()` method to disable it. Otherwise, it will run forever. When the user taps the *Pause* button during recording, we will invalidate the timer. Therefore, we create a method called `pauseTimer()`.

As soon as the user finishes a recording, he/she taps the *Stop* button. In this case, we have to invalidate the timer. At the same time, the `elapsedTimeInSecond` should be reset to zero too. This is what we have implemented in the `resetTimer()` method.

Now that you understand the timer implementation, it is time to modify some code to use the methods.

When the app starts to record an audio note, it should start the timer and update the timer label. So locate the following line of code in the `record` action method and insert the `startTimer()` method after it:

```
// Start recording
recorder.record()
startTimer()
```

The same applies to audio playback. When you start to play the audio file, the app should start the timer too. In the `play` action method, call the `startTimer()` method right after the line below:

```
audioPlayer?.play()
```

When the user pauses a recording, we should call `pauseTimer()` to invalidate the timer object. In the `record` action method, locate the following line of code and insert `pauseTimer()` after it:

```
recorder.pause()
```

Lastly, we need to stop and reset the timer when finishing an audio recording or playback. Locate the following line of code in the `stop` action method and insert `resetTimer()` after that:

```
audioRecorder?.stop()
```

For audio playback, it calls the `audioPlayerDidFinishPlaying` method when complete. So add `resetTimer()` in the method to reset the timer.

Great! You're ready to try out the app again. Now, the timer is ticking.

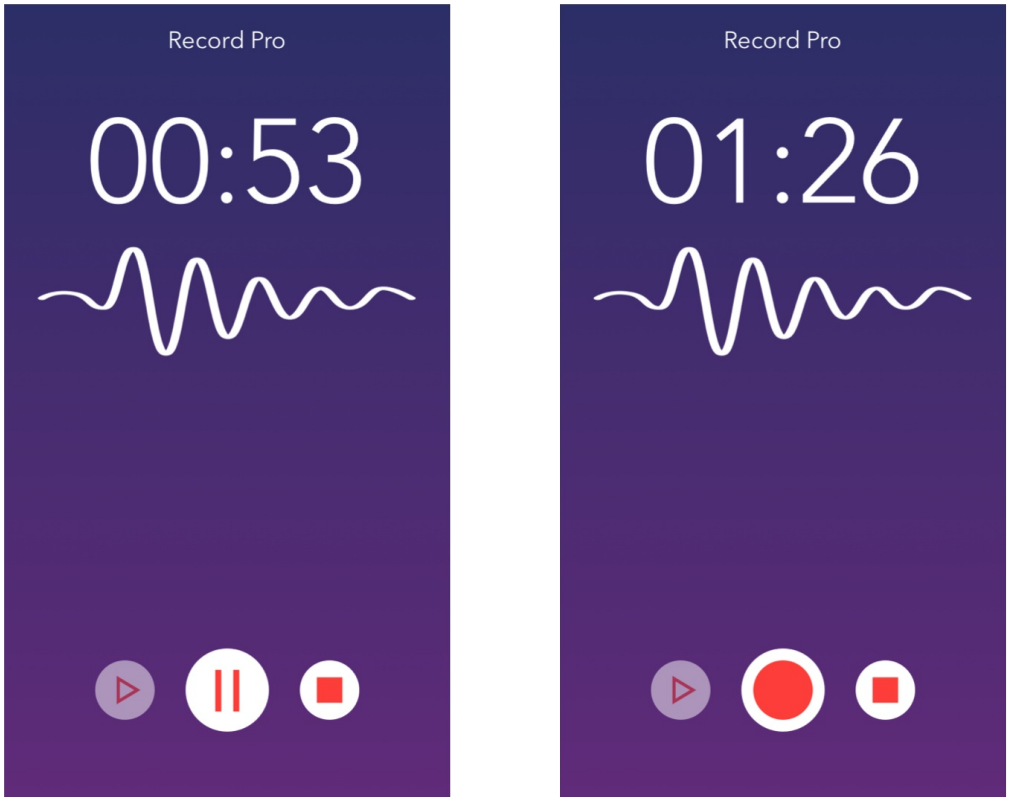
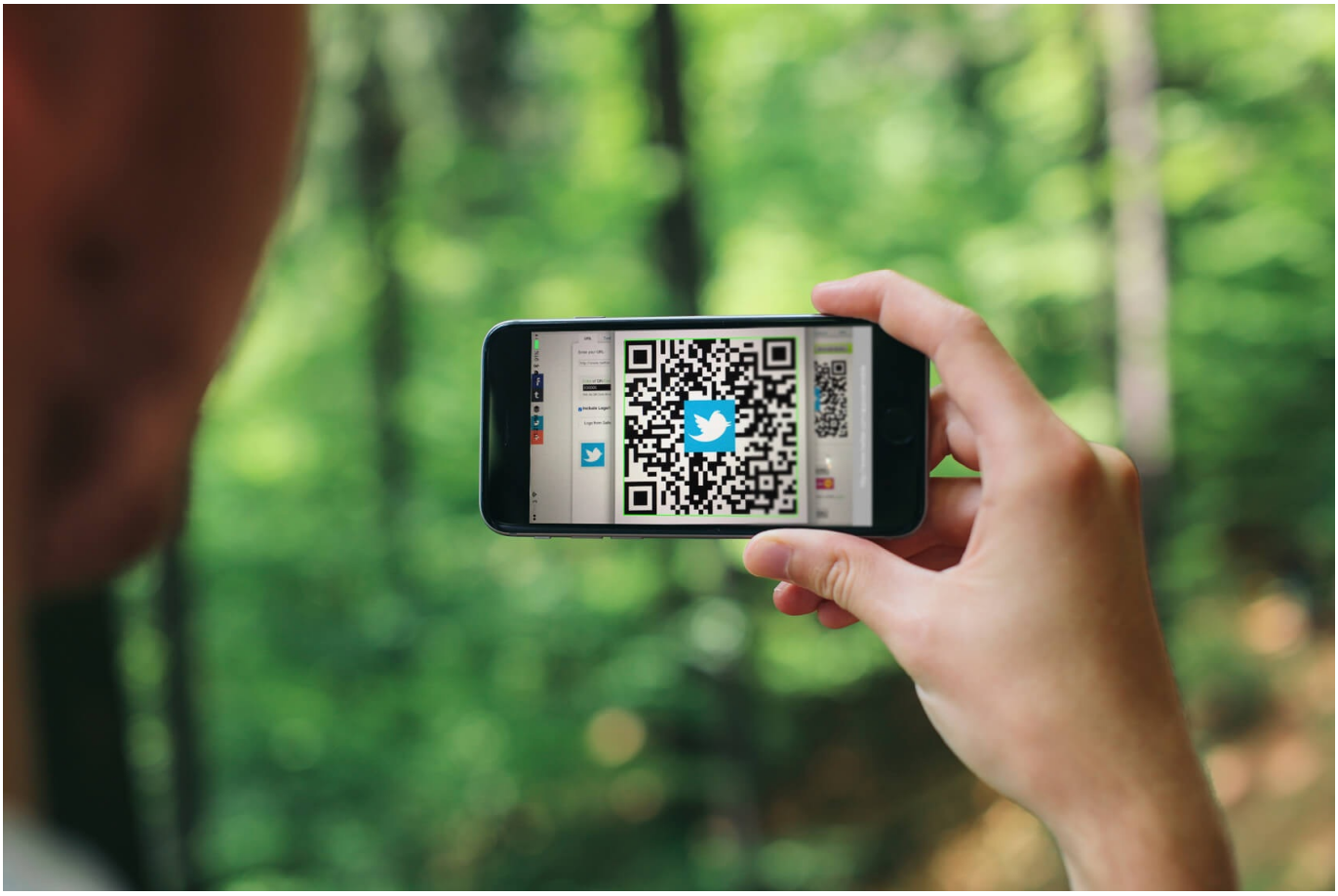


Figure 10.5. The timer is now working for both audio recording and playback

For reference, you can download the Xcode project from http://www.appcoda.com/resources/swift5/RecordPro.zip.

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, name cards and even food menu. 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

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, however, it's important to understand that any barcode scanning in iOS, including QR code scanning, is totally based on video capture. 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.

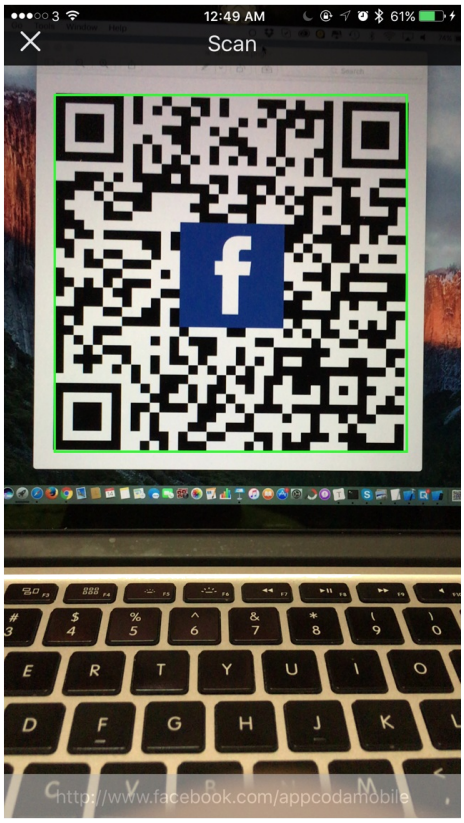


Figure 11.1. QR code reader demo

It's that simple.

To build the app, you can start by downloading the project template from http://www.appcoda.com/resources/swift5/QRCodeReaderStarter.zip. I have pre-built the storyboard and linked up the message label for you. The main screen is associated with the `QRCodeViewController` class, while the scanner screen is associated with the `QRScannerController` class.

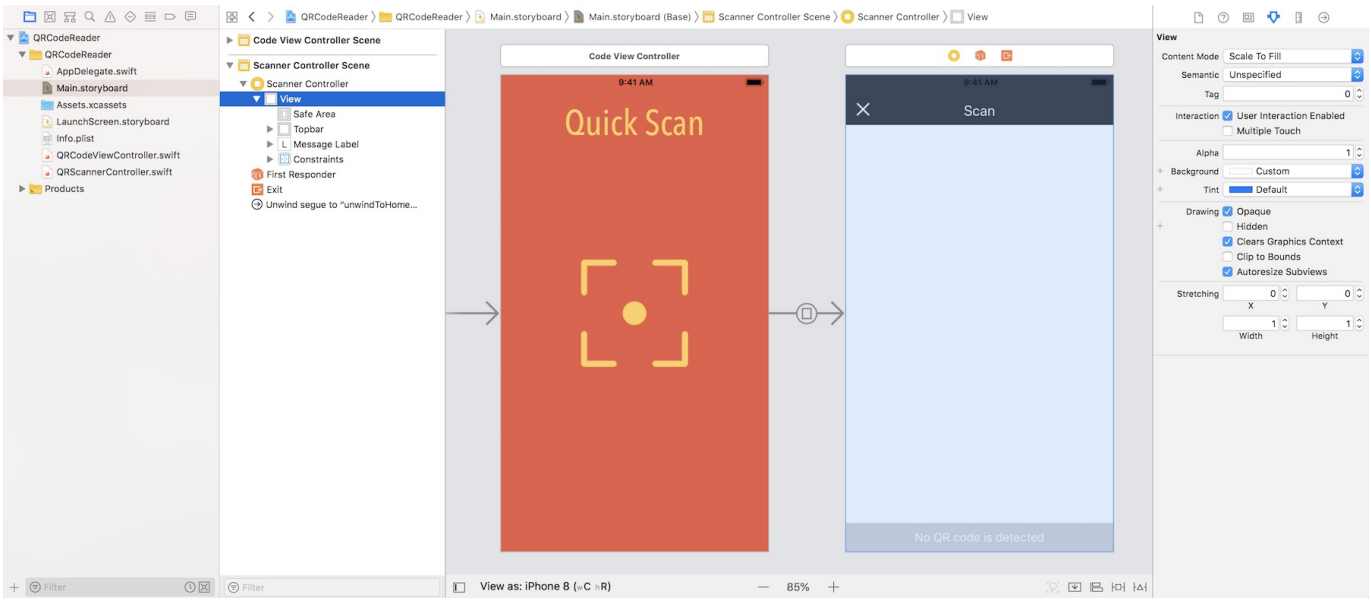


Figure 11.2. The starter project

You can run the starter project to have a look. After launching the app, you can tap the scan button to bring up the scan view. Later we will implement this view controller for QR code scanning.

Now that you understand how the starter project works, 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 is used to display the decoded information of the QR code and it is associated with the `messageLabel` property of the `QRScannerController` class.

As I mentioned earlier, we rely on the AVFoundation framework to implement the QR code scanning feature. First, open the `QRScannerController.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, adopt the protocol with an extension:

```
extension QRScannerController: AVCaptureMetadataOutputObjectsDelegate {

}
```

Before moving on, declare the following variables in the `QRScannerController` 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 capture. To perform a real-time capture, all we need to do is:

1. Look up the back camera device.
2. Set the input of the `AVCaptureSession` object to the appropriate `AVCaptureDevice` for video capturing.

Insert the following code in the `viewDidLoad` method of the `QRScannerController` class:

```
// Get the back-facing camera for capturing videos
guard let captureDevice = AVCaptureDevice.default(.builtInWideAngleCamera, for: .v
ideo, position: .back) else {
    print("Failed to get the camera device")
    return
}

do {
    // Get an instance of the AVCaptureDeviceInput class using the previous device
 object.
    let input = try AVCaptureDeviceInput(device: captureDevice)

    // 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
}
```

Assuming you've read the previous chapter, you should know that the `AVCaptureDevice.default` function is designed to find the default capture device matching a specific device type. In the code above, we specify to retrieve the device that supports the media type `.video`.

To perform a real-time capture, we use the `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.

```
// Set delegate and use the default dispatch queue to execute the call back
captureMetadataOutput.setMetadataObjectsDelegate(self, queue: DispatchQueue.main)
captureMetadataOutput.metadataObjectTypes = [AVMetadataObject.ObjectType.qr]
```

When new metadata objects are captured, they are forwarded to the delegate object for further processing. In the above code, we specify the dispatch queue on which to execute the delegate's methods. A dispatch queue can be either serial or concurrent. According to Apple's documentation, the queue must be a serial queue. So, we use `DispatchQueue.main` to get the default serial queue.

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 `AVMetadataObject.ObjectType.qr` clearly indicates our purpose. We want to do QR code scanning.

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. Insert the code below in the `do-catch` block:

```
// Initialize the video preview layer and add it as a sublayer to the viewPreview
view's layer.
videoPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
videoPreviewLayer?.videoGravity = AVLayerVideoGravity.resizeAspectFill
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 crashes unexpectedly with the following error when you tap the scan button:

```
This app has crashed because it attempted to access privacy-sensitive data without
 a usage description.  The app's Info.plist must contain an NSCameraUsageDescripti
on key with a string value explaining to the user how the app uses this data.
```

Similar to what we have done in the audio recording chapter, iOS requires app developers to obtain the user's permission before allowing to access the camera. To do so, you have to add a key named `NSCameraUsageDescription` in the `Info.plist` file. Open the file and right-click any blank area to add a new row. Set the key to *Privacy - Camera Usage Description*, and value to *We need to access your camera for scanning QR code*.

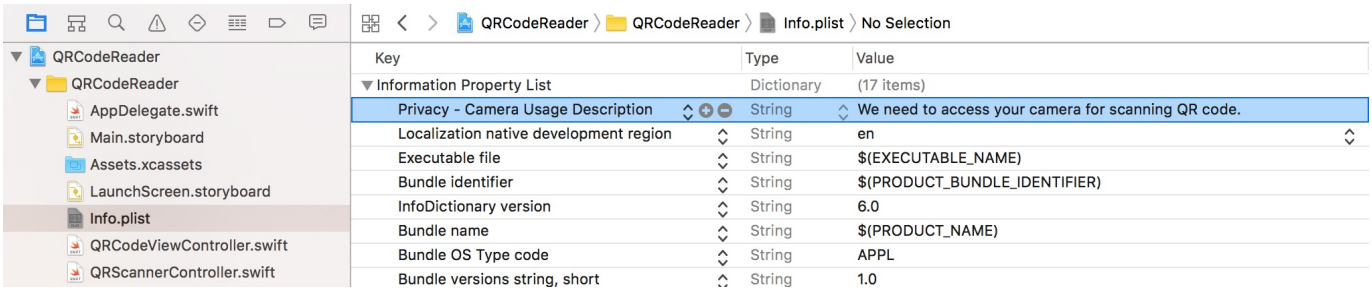


Figure 11.3. Editing Info.plist to add a new key

Once you finish the editing, deploy the app and run it on a real device again. Tapping the scan button should bring up the built-in camera and start capturing video. However, at this point the message label and the top bar are hidden. You can fix it by adding the following line of code. This will move the message label and top bar to appear on top of the video layer.

```
// Move the message label and top bar to the front
view.bringSubviewToFront(messageLabel)
view.bringSubviewToFront(topbar)
```

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

Implementing QR Code Reading

As of now, the app looks pretty much like a video capture app. 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. 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.green.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 metadataOutput(_ output: AVCaptureMetadataOutput, didOutput metadata
Objects: [AVMetadataObject], from 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 metadataOutput(_ output: AVCaptureMetadataOutput, didOutput metadataObjects:
[AVMetadataObject], from connection: AVCaptureConnection) {
    // Check if the metadataObjects array is not nil and it contains at least one
object.
    if metadataObjects.count == 0 {
        qrCodeFrameView?.frame = CGRect.zero
        messageLabel.text = "No QR code is detected"
        return
    }

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

    if metadataObj.type == AVMetadataObject.ObjectType.qr {
        // 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?.transformedMetadataObject(for: meta
dataObj)
        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 `transformedMetadataObject(for:)` 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.

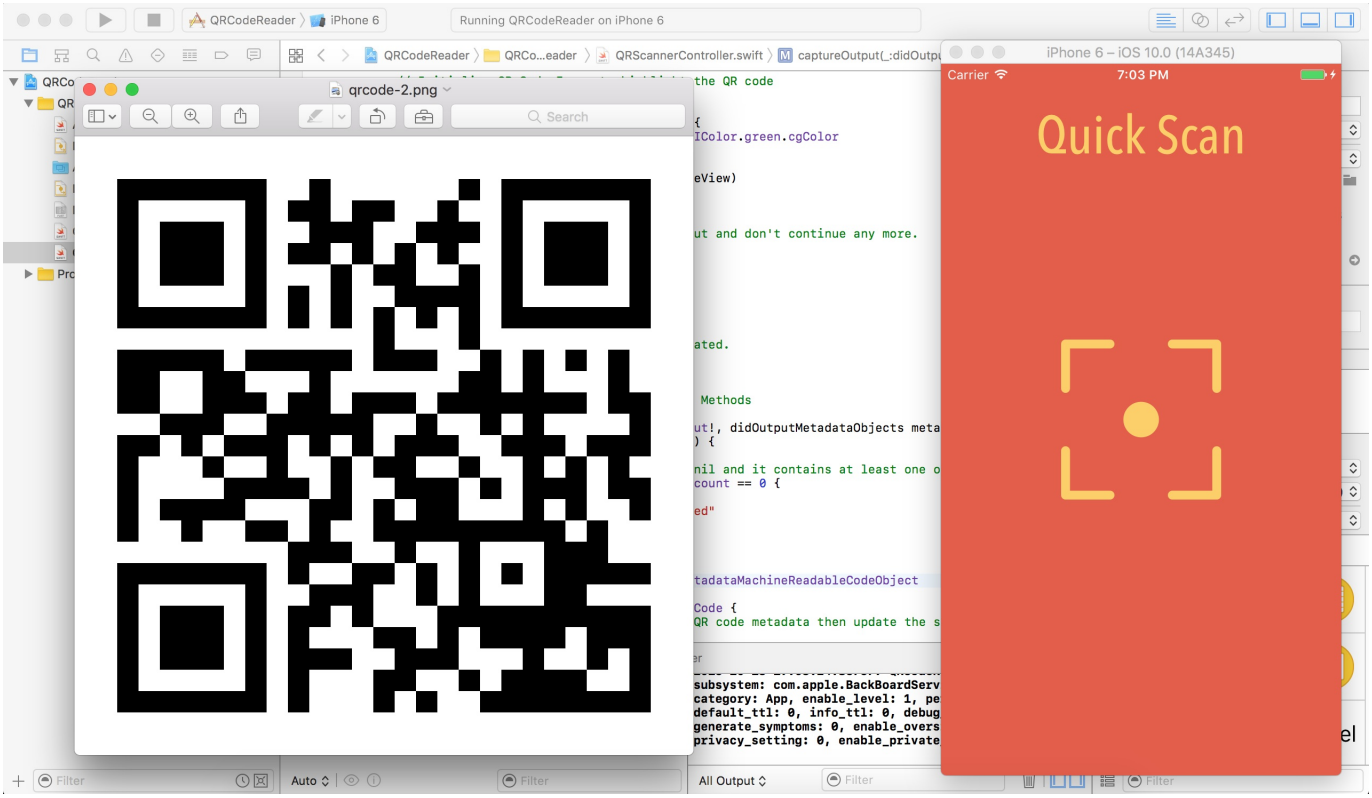


Figure 11.4. Sample QR code

Once launched, tap the scan button and then point the device to the QR code in figure 11.4. The app immediately detects the code and decodes the information.

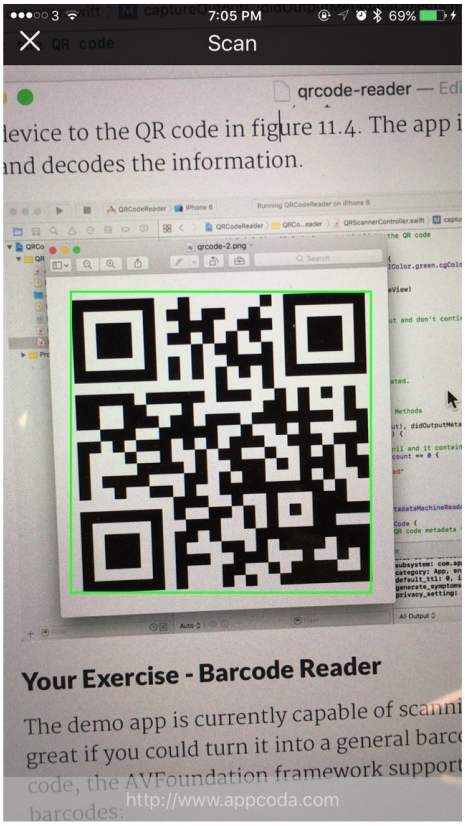


Figure 11.5. Scanning and decoding QR codes

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 (`AVMetadataObject.ObjectType.upce`)
- Code 39 (`AVMetadataObject.ObjectType.code39`)
- Code 39 mod 43 (`AVMetadataObject.ObjectType.code39Mod43`)
- Code 93 (`AVMetadataObject.ObjectType.code93`)
- Code 128 (`AVMetadataObject.ObjectType.code128`)
- EAN-8 (`AVMetadataObject.ObjectType.ean8`)
- EAN-13 (`AVMetadataObject.ObjectType.ean13`)
- Aztec (`AVMetadataObject.ObjectType.aztec`)

- PDF417 (`AVMetadataObject.ObjectType.pdf417`)
- ITF14 (`AVMetadataObject.ObjectType.itf14`)
- Interleaved 2 of 5 codes (`AVMetadataObject.ObjectType.interleaved2of5`)
- Data Matrix (`AVMetadataObject.ObjectType.dataMatrix`)

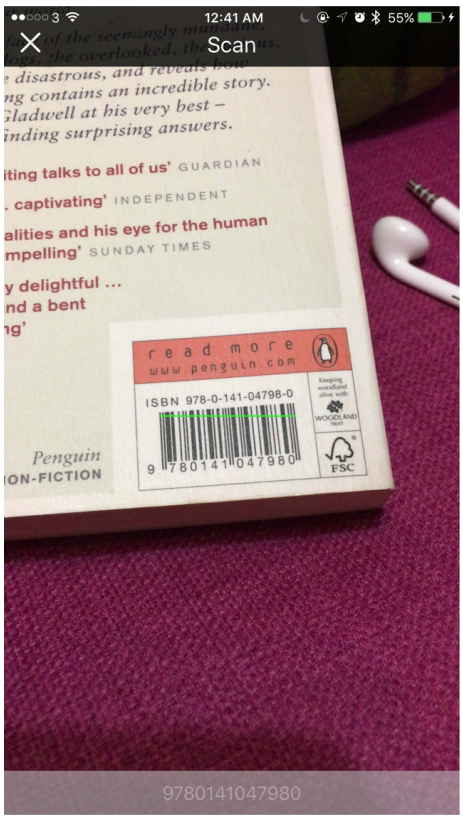


Figure 11.6. Scanning a barcode

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 = [AVMetadataObject.ObjectType.qr]
```

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 http://www.appcoda.com/resources/swift5/QRCodeReader.zip.

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 http://www.appcoda.com/resources/swift5/QRCodeReader.zip. 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.

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.

Figure 12.1. Sample QR codes

Using URL Schemes

For most of the built-in applications, Apple provides support for 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 `open(_:options:completionHandler:)` method of the `UIApplication` class. Here is the line of code:

```
UIApplication.shared.open(url, options: [:], completionHandler: nil)
```

Now, we will modify the demo app to open the corresponding app when a QR code is decoded. Open the Xcode project and select the `QRScannerController.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: .default, handler:
{ (action) -> Void in

        if let url = URL(string: decodedURL) {
            if UIApplication.shared.canOpenURL(url) {
                UIApplication.shared.open(url, options: [:], completionHandler: nil
)
            }
        }
    })

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

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

    present(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 `URL` object and opens it accordingly. iOS will then open the corresponding app based on the given URL.

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

```
launchApp(decodedURL: 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.

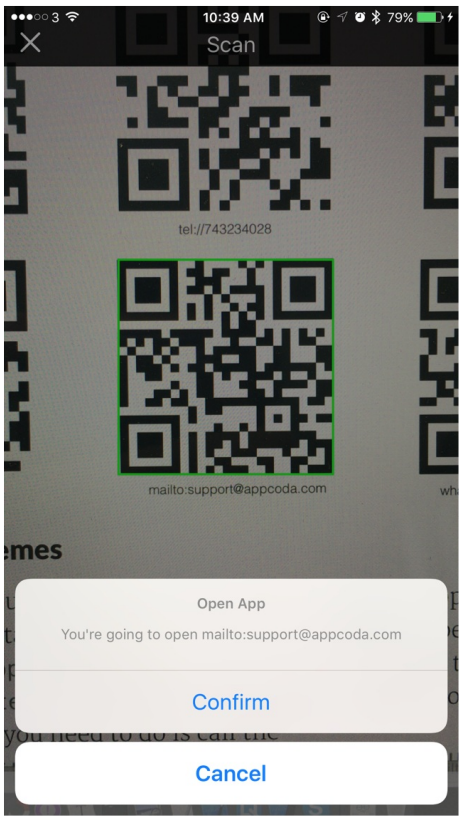


Figure 12.2. The app displays an action sheet once the QR code is decoded.

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

```
2020-01-21 18:34:16.802771+0800 QRCodeReader[4120:1013537] Warning: Attempt to pre
sent <UIAlertController: 0x10500d000>  on <QRCodeReader.QRScannerController: 0x108
103d70> which is already presenting <UIAlertController: 0x104818e00>
```

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 `present(_:animated:completion:)` method.

In iOS, when you present a view controller modally using the `present(_:animated:completion:)` method, the presented view controller is stored in the `presentedViewController` property of the current view controller. For example, when the `QRScannerController` object calls the `present(_:animated:completion:)` 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 to 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.

One thing you may notice is that the app cannot open these two URLs:

- `fb://feed`
- `whatsapp://send?text=Hello!`

These URLs are known as custom URL schemes created by third-party apps. For iOS 9 and later, 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`. If you refer to the console messages, you should see the error like this:

```
2020-01-21 18:36:22.384643+0800 QRCodeReader[4123:1014524] -canOpenURL: failed for
 URL: "fb://feed" - error: "This app is not allowed to query for scheme fb"
```

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 blank areaU and select `Add Row` to create a new key.

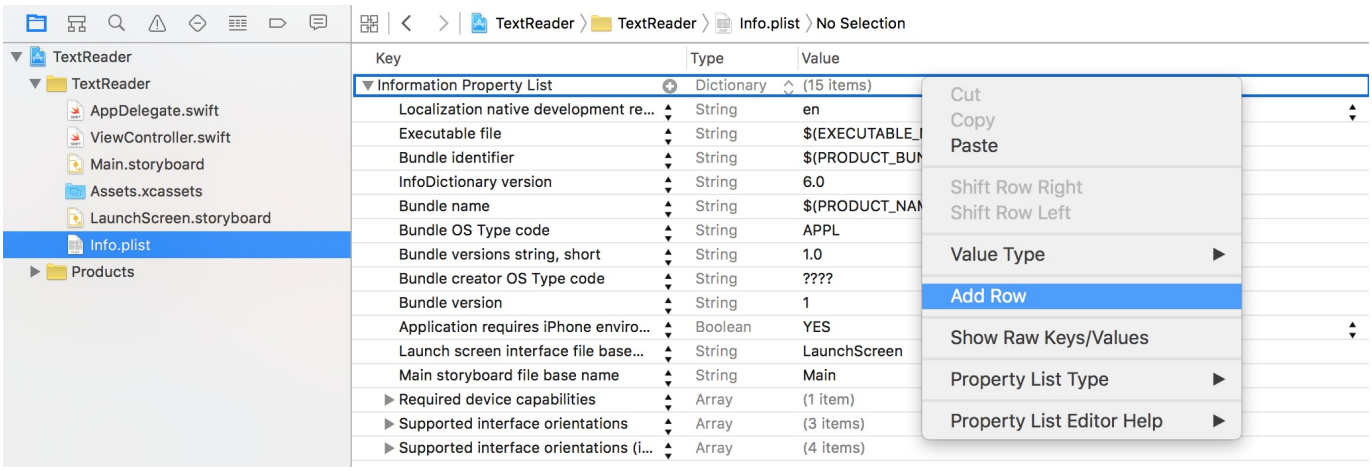


Figure 12.3. Add new row in Info.plist

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. You can further click the disclosure icon (i.e. triangle) to expand it. Next, 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.

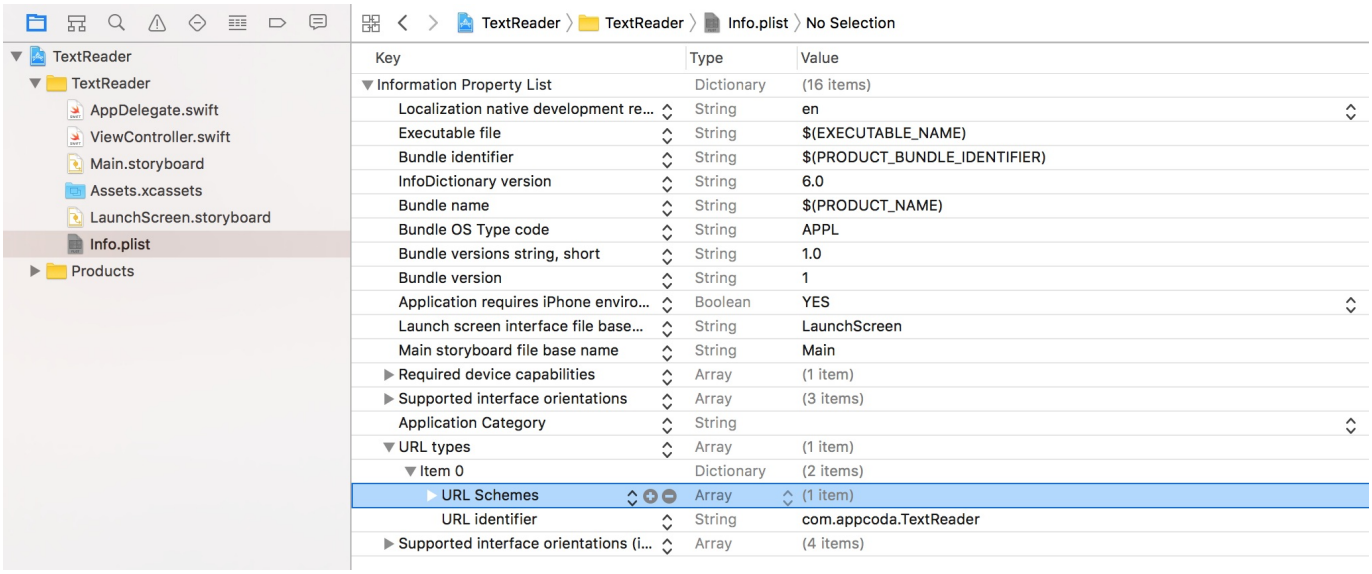


Figure 12.4. Adding a URL Schemes key

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:

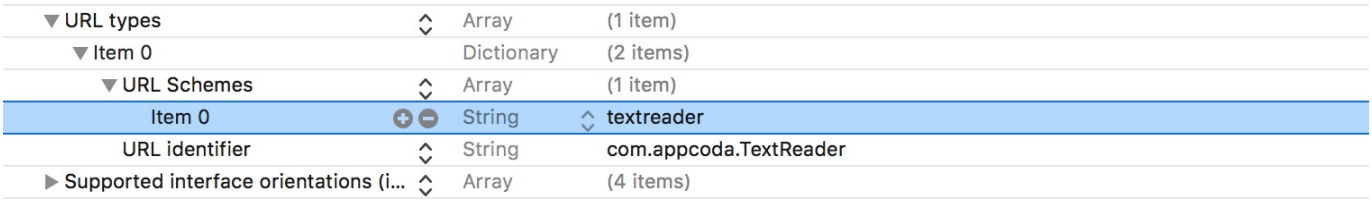


Figure 12.5. Set the value to textreader

That's it. We have configured a custom URL scheme in the TextReader app. Now the app accepts the URL in the form of `textreader://<message>`. We still need to write a few lines of code such 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.

If there is a *Open a URL* event sent to your app, the system calls the `application(_:open:options:)` method of the app delegate. Therefore, you'll need to implement the method in order to respond to the launch of the custom URL.

```
optional func application(_ app: UIApplication, open url: URL, options: [UIApplica
tionOpenURLOptionsKey : Any] = [:]) -> Bool
```

Open `AppDelegate.swift` and insert the following code to implement the method:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplicationOpenU
RLOptionsKey : Any] = [:]) -> Bool {

    let message = url.host?.removingPercentEncoding
    let alertController = UIAlertController(title: "Incoming Message", message: me
ssage, preferredStyle: .alert)
    let okAction = UIAlertAction(title: "OK", style: UIAlertActionStyle.default, h
andler: nil)
    alertController.addAction(okAction)

    window?.rootViewController?.present(alertController, animated: true, completio
n: nil)

    return true
}
```

From the arguments of the `application(_:open:options:)` 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 `URL` structure.

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

`removingPercentEncoding` 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!

Figure 12.6. Sample QR code

The console should show you the following error:

```
2020-01-21 18:45:52.999747+0800 QRCodeReader[4151:1019370] -canOpenURL: failed for
 URL: "textreader://Great!%20It%20works!" - error: "This app is not allowed to que
ry for scheme textreader"
```

As explained earlier, Apple has made some changes to the `canOpenURL` method since iOS 9. 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

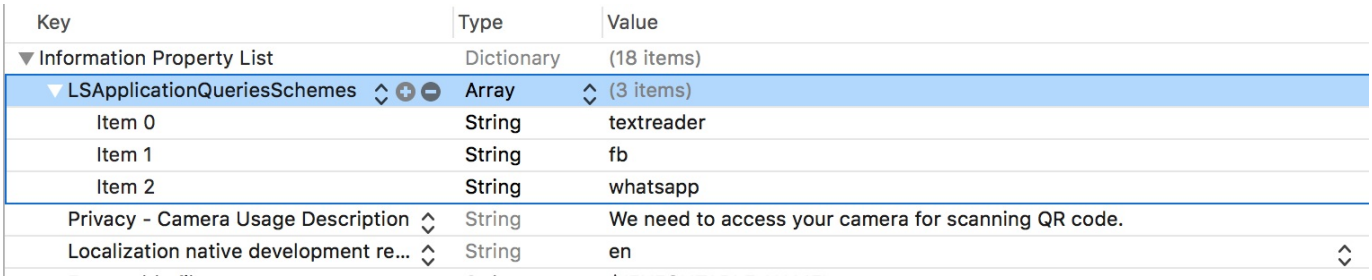


Figure 12.7. Register the custom schemes in Info.plist

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.

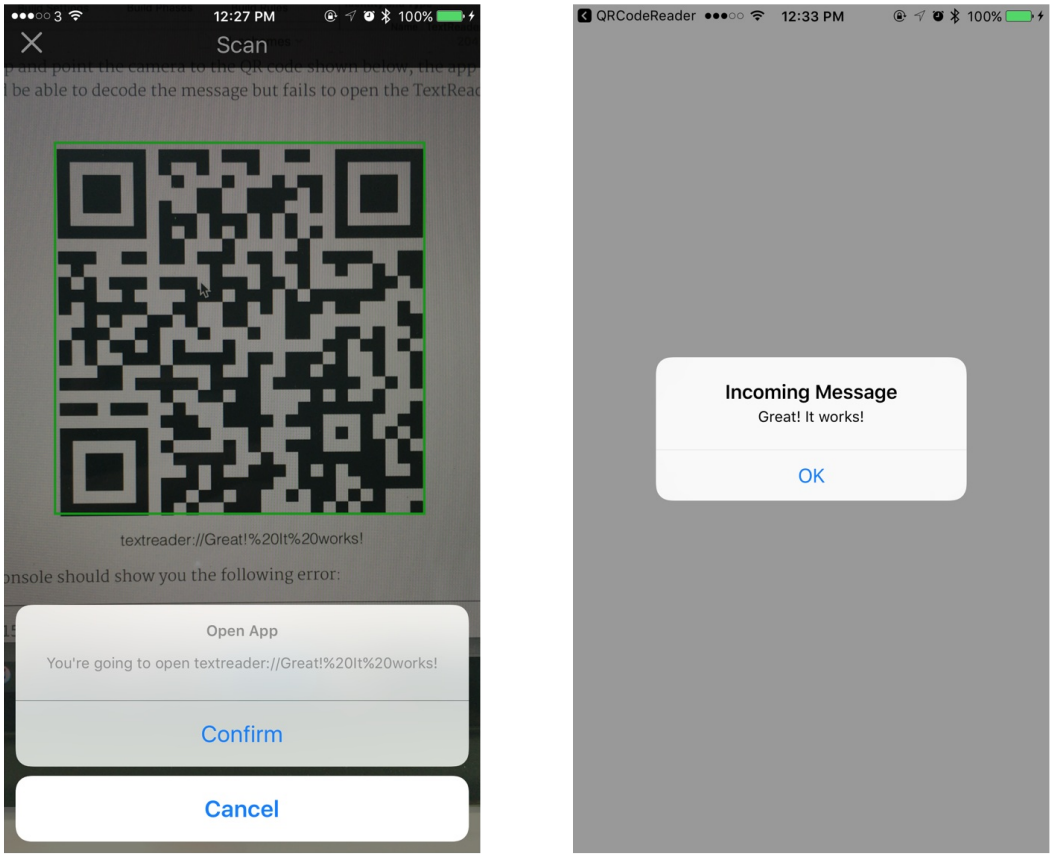


Figure 12.8. Opening the TextReader app with a custom URL scheme

Furthermore, if you scan the QR code of the Facebook scheme or Whatsapp scheme, the app should now be able to launch the Facebook/Whatsapp app accordingly.

For reference, you can download the projects here:

- QRCodeReader project http://www.appcoda.com/resources/swift5/QRReaderURLScheme.zip
- TextReader project http://www.appcoda.com/resources/swift5/TextReader.zip.

Chapter 13
Building a Full Screen Camera with Gesture-based Controls

iOS provides two ways for developers to access the built-in camera for taking photos. The simple approach is to use `UIImagePickerViewController`, which I briefly covered in the Beginning iOS 13 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 the AVFoundation framework. Compared to `UIImagePickerViewController`, 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. an 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.

Figure 13.1. Simple Camera App

To begin, you can download the Xcode project template from http://www.appcoda.com/resources/swift5/SimpleCameraStarter.zip. The template includes a pre-built storyboard and custom classes. If you open the Storyboard, you will find two view controllers. The Simple Camera Controller is used to display the camera interface, while the Photo View Controller is designed for displaying a still image after taking the photo. Both view controllers are associated with the corresponding class. The Simple Camera Controller is connected with the `SimpleCameraController` 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 connected with the `save` action method, which is now without any implementation.

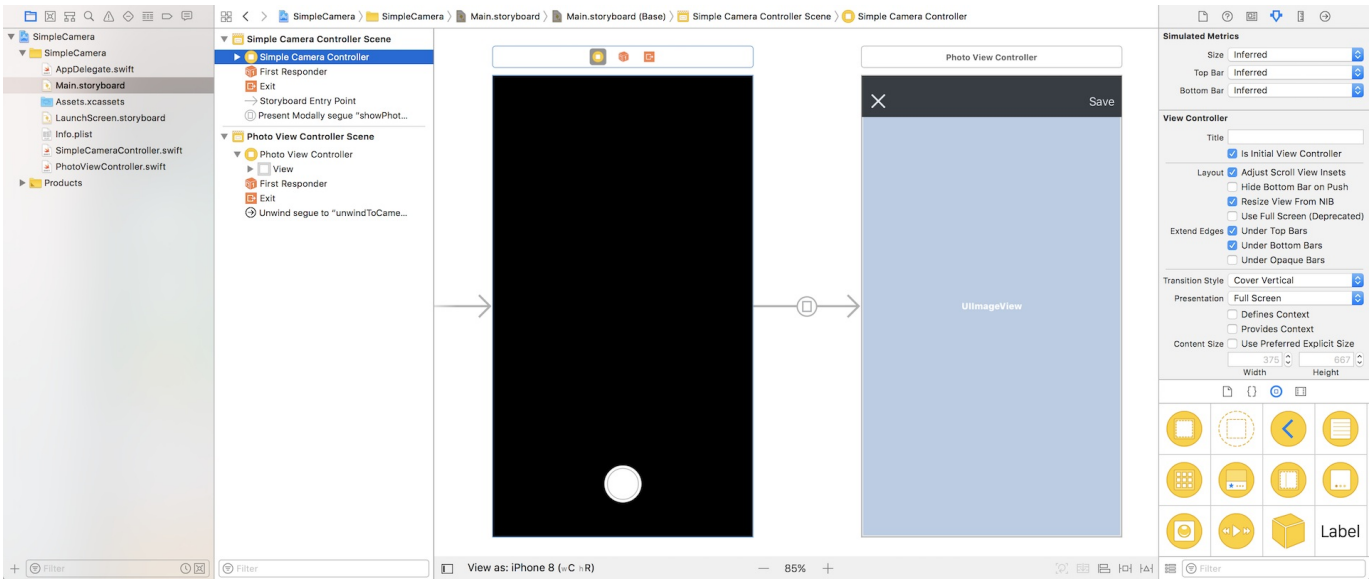


Figure 13.2. The storyboard of the starter project

Configuring a Session

The heart of AVFoundation media capture is the `AVCaptureSession` object. So open `SimpleCameraController.swift` and declare a variable of the type `AVCaptureSession`:

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

Since the APIs is available in the AVFoundation framework, make sure you import the package in order to use it:

```
import AVFoundation
```

Create a `configure()` method to configure the session and insert the following code:

```
private func configure() {
    // Preset the session for taking photo in full resolution
    captureSession.sessionPreset = AVCaptureSession.Preset.photo
}
```

You use the `sessionPreset` property to specify the image quality and resolution you want. Here we preset it to `AVCaptureSession.Preset.photo`, 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 `SimpleCameraController` 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.

Continue to insert the following code in the `configure()` method:

```
// Get the front and back-facing camera for taking photos
let deviceDiscoverySession = AVCaptureDevice.DiscoverySession(deviceTypes: [.built
InWideAngleCamera], mediaType: AVMediaType.video, position: .unspecified)

for device in deviceDiscoverySession.devices {
    if device.position == .back {
        backFacingCamera = device
    } else if device.position == .front {
        frontFacingCamera = device
    }
}

currentDevice = backFacingCamera

guard let captureDeviceInput = try? AVCaptureDeviceInput(device: currentDevice) el
se {
    return
}
```

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.DiscoverySession` class is designed to find all available capture devices matching a specific device type (such as a microphone or wide-angle camera), supported media types for capture (such as audio, video, or both), and position (front- or back-facing).

In the code snippet, we create a device discovery session to find the available capture devices that are capable of capturing video/still image (i.e. `AVMediaType.video`). The iPhone device now comes with several cameras: wide angle camera, telephoto, and true depth camera. Here we specify to find the cameras (i.e. `.builtInWideAngleCamera`) without a specific position.

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 `SimpleCameraController` class for the device output:

```
var stillImageOutput: AVCapturePhotoOutput!
var stillImage: UIImage?
```

Insert the following code in the `configure` method:

```
// Configure the session with the output for capturing still images
stillImageOutput = AVCapturePhotoOutput()
```

Here we create an instance of `AVCapturePhotoOutput` for capturing still images. Introduced in iOS 10, this class supports the basic capture of still images, RAW-format capture, and Live Photos.

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 `configure` 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 `configure` method:

```
// Provide a camera preview
cameraPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
view.layer.addSublayer(cameraPreviewLayer!)
cameraPreviewLayer?.videoGravity = AVLayerVideoGravity.resizeAspectFill
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 `AVLayerVideoGravity.resizeAspectFill`. You're free to explore the other two options (`AVLayerVideoGravity.resize` and `AVLayerVideoGravity.resizeAspect`) 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.

Before you test the app, insert the following line of code in the `viewDidLoad()` method to call the `configure()` method:

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

    configure()
}
```

There is one last thing you have to add. You will have to insert an entry in the `Info.plist` file to specify the reason why you need to access the camera. The message will be displayed to the user when the app is first used. It is mandatory to ask for the user's permission, otherwise, your app will not be able to access the camera.

```
SimpleCamera[569:412396] [access] This app has crashed because it attempted to acc
ess privacy-sensitive data without a usage description.  The app's Info.plist must
 contain an NSCameraUsageDescription key with a string value explaining to the use
r how the app uses this data.
```

In the `Info.plist` file, insert a key named `Privacy - Camera Usage Description` and specify the reason (e.g. for capturing photos) in the value field.

That's it. 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 `SimpleCameraController.swift` file to the following:

```
@IBAction func capture(sender: UIButton) {

    // Set photo settings
    let photoSettings = AVCapturePhotoSettings(format: [AVVideoCodecKey: AVVideoCo
decType.jpeg])
    photoSettings.isAutoStillImageStabilizationEnabled = true
    photoSettings.isHighResolutionPhotoEnabled = true
    photoSettings.flashMode = .auto

    stillImageOutput.isHighResolutionCaptureEnabled = true
    stillImageOutput.capturePhoto(with: photoSettings, delegate: self)

}
```

To capture a photo using `AVCapturePhotoOutput`, the first thing you need to do is create an `AVCapturePhotoSettings` object to specify settings for the capture. For example, do you need to enable image stabilization? What's the flash mode? In the code above, we specify to capture the photo in high-resolution JPEG format with image stabilization enabled.

With the photo settings, you can then call the `capturePhoto` method to begin capturing the photo. The method takes in the photo settings and a delegate object. Once the photo is captured, it will call its delegate for further processing.

The delegate object, which is `SimpleCameraController`, should implement the `AVCapturePhotoCaptureDelegate` protocol. Again, we will create an extension to adopt the protocol. Insert the following code in the `SimpleCameraController.swift` file:

```
extension SimpleCameraController: AVCapturePhotoCaptureDelegate {
    func photoOutput(_ output: AVCapturePhotoOutput, didFinishProcessingPhoto phot
o: AVCapturePhoto, error: Error?) {
        guard error == nil else {
            return
        }

        // Get the image from the photo buffer
        guard let imageData = photo.fileDataRepresentation() else {
            return
        }

        stillImage = UIImage(data: imageData)
        performSegue(withIdentifier: "showPhoto", sender: self)
    }
}
```

When the capture is complete, the `photoOutput(_:didFinishProcessingPhoto:error:)` method is called. In the implementation, we first check if there is any error. The captured photo is embedded in the `photo` parameter. You can access the image data by calling the `fileDataRepresentation()` method. With the image data, we can construct the image by using `UIImage`.

Lastly, we invoke the *showPhoto* segue to display the still image in the Photo View Controller. So, remember to add the `prepare(for:sender:)` method in the `SimpleCameraController` class:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    // Get the new view controller using segue.destinationViewController.
    // Pass the selected object to the new view controller.
    if segue.identifier == "showPhoto" {
        let photoViewController = segue.destination 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 `configure()` method:

```
// Toggle Camera recognizer
toggleCameraGestureRecognizer.direction = .up
toggleCameraGestureRecognizer.addTarget(self, action: #selector(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 `SimpleCameraController` class:

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

    // Change the device based on the current camera
    guard let newDevice = (currentDevice?.position == AVCaptureDevice.Position.bac
k) ? frontFacingCamera : backFacingCamera else {
        return
    }

    // 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 `SimpleCameraController` class, declare two instance variables of `UISwipeGestureRecognizer`:

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

Next, insert the following lines of code in the `configure()` method:

```
// Zoom In recognizer
zoomInGestureRecognizer.direction = .right
zoomInGestureRecognizer.addTarget(self, action: #selector(zoomIn))
view.addGestureRecognizer(zoomInGestureRecognizer)

// Zoom Out recognizer
zoomOutGestureRecognizer.direction = .left
zoomOutGestureRecognizer.addTarget(self, action: #selector(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`:

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

@objc func zoomOut() {
    if let zoomFactor = currentDevice?.videoZoomFactor {
        if zoomFactor > 1.0 {
            let newZoomFactor = max(zoomFactor - 1.0, 1.0)
            do {
                try currentDevice?.lockForConfiguration()
                currentDevice?.ramp(toVideoZoomFactor: 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 `ramp(toVideoZoomFactor:withRate:)` 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 `ramp(toVideoZoomFactor:withRate:)` 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: Any?, _
completionSelector: Selector?, _ contextInfo: UnsafeMutableRawPointer?)
```

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: UIButton) {
    guard let imageToSave = image else {
        return
    }

    UIImageWriteToSavedPhotosAlbum(imageToSave, nil, nil, nil)
    dismiss(animated: 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.

Before you can build the project to test the app again, you have to edit a key in `Info.plist`. In iOS 10, you can no longer save photos to the album without user consent. To ask for the user's permission, add a new row in the `Info.plist` file. Set the key to *Privacy - Photo Library Additions Usage Description*, and the value to `To save your photos`. This is the message that explains why our app has to access the photo library, and it will be prompted when the app first tries to access photo library for saving photos.

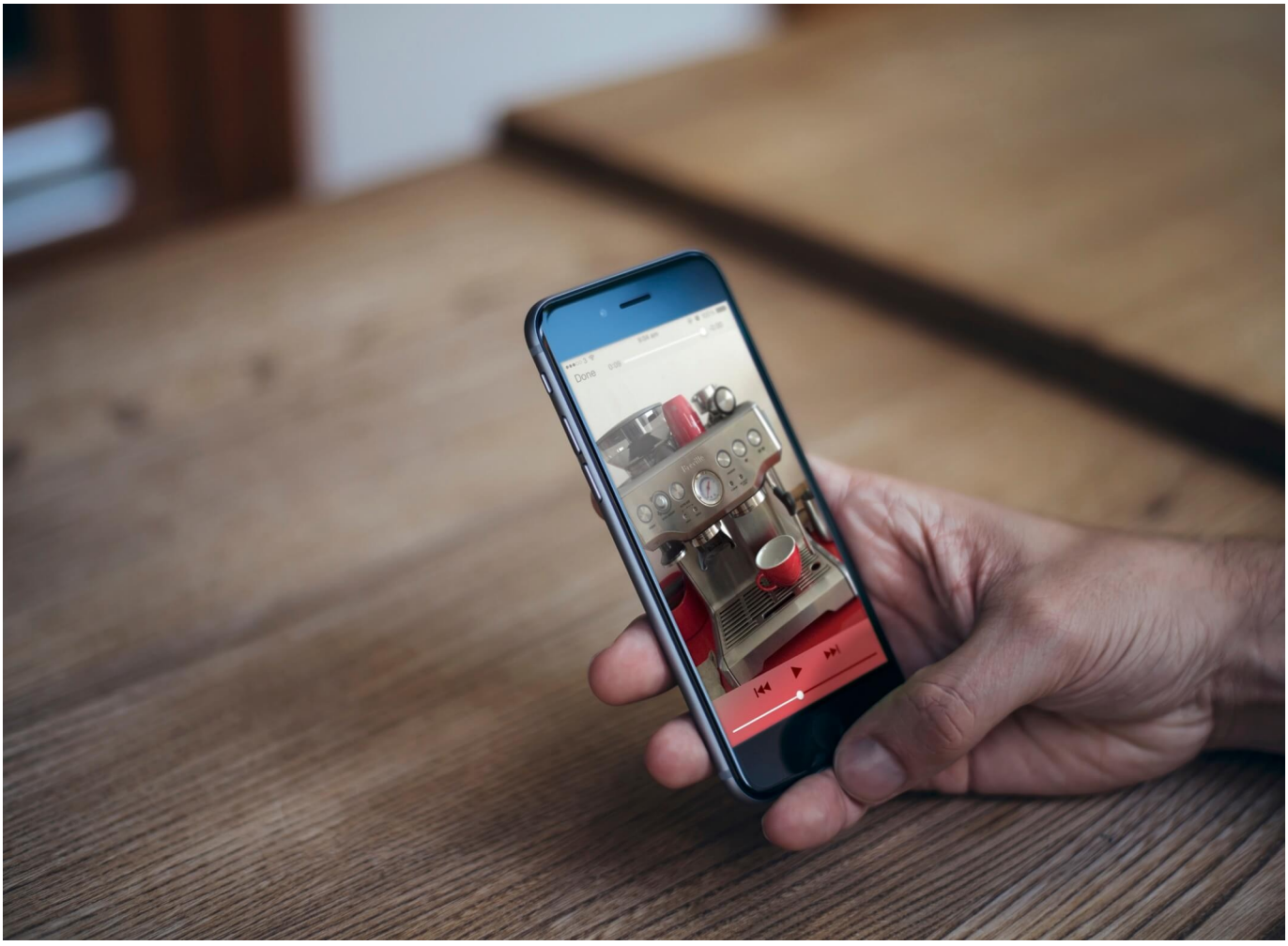
Hit the Run button again to test the app. The Camera app should now be able to save photos to your photo album. To verify the result, you can open the stock Photos app to take a look. The photo should be saved in the album.

Figure 13.3. Demo app

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 reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/SimpleCamera.zip.

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 will we explore video capturing, but I will also show you a 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 starter project from http://www.appcoda.com/resources/swift5/SimpleVideoCamStarter.zip. The starter project is very similar to the one you worked on in the previous chapter. If you run the project, you will see a blank screen with a red button (which is the record button) at the bottom part of the screen.

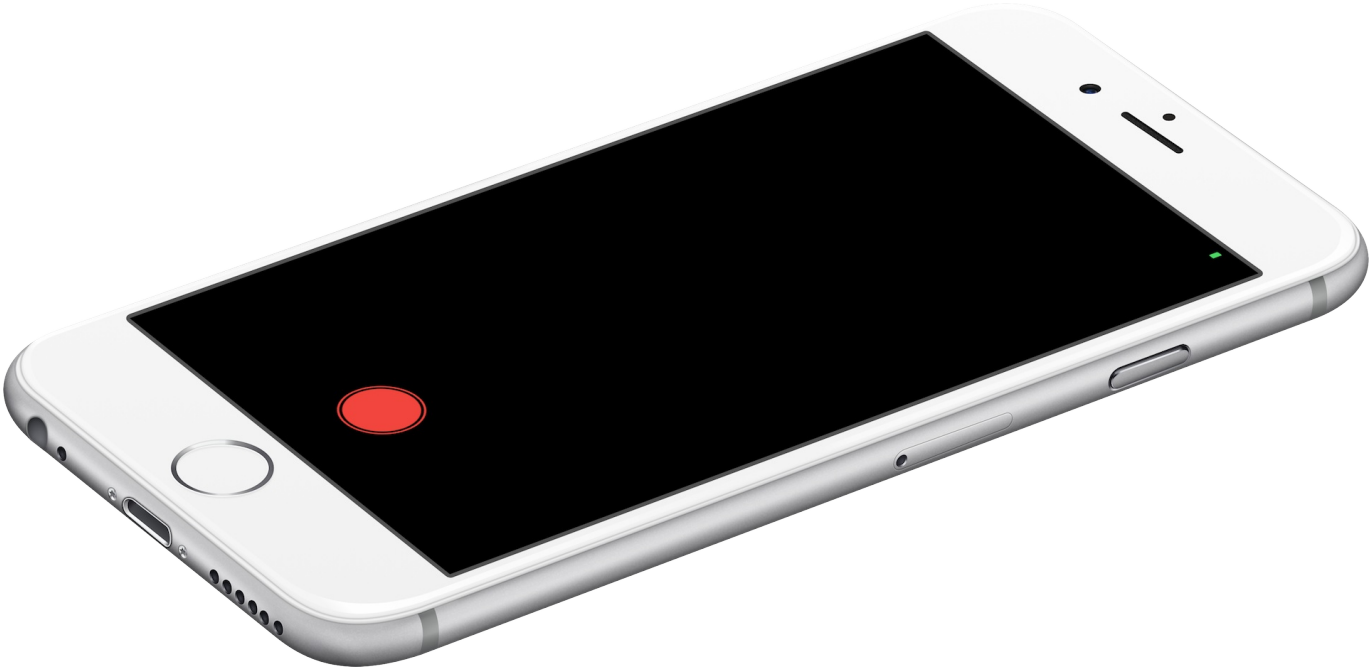


Figure 14.1. Running the starter project will give you a blank screen with the camera button

Configuring a Session

Similar to image capturing, the first thing to do is import the `AVFoundation` framework and prepare the `AVCaptureSession` object. In the `SimpleVideoCamController.swift` file, insert the following statement at the beginning of the file:

```
import AVFoundation
```

And, declare an instance variable of `AVCaptureSession`:

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

Let's create a `configure()` method to handle all the configurations:

```
private func configure() {
    // Preset the session for taking photo in full resolution
    captureSession.sessionPreset = AVCaptureSession.Preset.high
}
```

Here, we define the session and preset it to `AVCaptureSession.Preset.high`, which indicates a high quality output. Alternatively, you can set the value to `AVCaptureSession.Preset.low`, 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 low.

Selecting the Input Device

Next, we have to find out the camera devices for shooting videos. First declare the following instance variable in the `SimpleVideoCamController` class:

```
var currentDevice: AVCaptureDevice!
```

Then, insert the following code in the `configure()` method:

```
// Get the back-facing camera for capturing videos
guard let device = AVCaptureDevice.default(.builtInWideAngleCamera, for: .video, p
osition: .back) else {
    print("Failed to get the camera device")
    return
}

currentDevice = device

// Get the input data source
guard let captureDeviceInput = try? AVCaptureDeviceInput(device: currentDevice) el
se {
    return
}
```

If you've read the previous chapter, you may find the code looks very similar. We use the `AVCaptureDevice.default` to retrieve the default camera for capturing videos.

The `AVCaptureDevice.default` function is designed to find the default capture devices matching a specific device type (such as a microphone or wide-angle camera), supported media types for capture (such as audio, video, or both), and position (front- or back-facing). Here we want to find the built-in wide camera, which is back-facing.

With the camera device, we then create an instance of `AVCaptureDeviceInput` as the data input source.

Configuring an Output Device

The input device is ready, let's see how to configure the output device. Declare the following variable in the `SimpleVideoCamController` class for the device output:

```
var videoFileOutput: AVCaptureMovieFileOutput!
```

Insert the following code in the `configure()` 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 `configure()` 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?
```

Continue to insert the following code in the `configure()` method:

```
// Provide a camera preview
cameraPreviewLayer = AVCaptureVideoPreviewLayer(session: captureSession)
view.layer.addSublayer(cameraPreviewLayer!)
cameraPreviewLayer?.videoGravity = AVLayerVideoGravity.resizeAspectFill
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. `AVLayerVideoGravity.resizeAspectFill` indicates that the video's aspect ratio should be preserved and filled the layer's bounds.

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. However, the app is not ready for video capturing yet.

Let's continue to implement the feature.

Saving Video Data to a Movie File

The recording process starts when the user taps the red button. While the app is in recording mode, the camera button is animated to indicate the recording is in progress. Once the user taps the button again, the animation stops and the video is saved to a file.

In order to keep track of the status (recording / non-recording) of the app, we first declare a Boolean variable to indicate whether video recording is taking place:

```
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.animate(withDuration: 0.5, delay: 0.0, options: [.repeat, .autoreve
rse, .allowUserInteraction], animations: { () -> Void in
            self.cameraButton.transform = CGAffineTransform(scaleX: 0.5, y: 0.5)
        }, completion: nil)

        let outputPath = NSTemporaryDirectory() + "output.mov"
        let outputFileURL = URL(fileURLWithPath: outputPath)
        videoFileOutput?.startRecording(to: outputFileURL, recordingDelegate: self
)
    } else {
        isRecording = false

        UIView.animate(withDuration: 0.5, delay: 1.0, options: [], animations: { (
) -> Void in
            self.cameraButton.transform = CGAffineTransform(scaleX: 1.0, y: 1.0)
        }, completion: nil)
        cameraButton.layer.removeAllAnimations()
        videoFileOutput?.stopRecording()
    }
}
```

In the above code, 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 Chapter 16 of the Beginning iOS 13 Programming book, the `animate(withDuration:delay:options:animations:completion:)` 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.animate(withDuration: 0.5, delay: 0.0, options: [.repeat, .autoreverse, .al
lowUserInteraction], animations: { () -> Void in
                self.cameraButton.transform = CGAffineTransform(scaleX: 0.5, y: 0.5
)
                }, completion: nil)
```

For step #1, we use `CGAffineTransform` 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, the user will 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 `startRecording` to capture data to a movie file. All you need to do is specify an output file path and the delegate object.

```
videoFileOutput?.startRecording(to: outputFileURL, recordingDelegate: self)
```

Here, we save the video to the temporary folder (`NSTemporaryDirectory()`) with the file name *output.mov*.

Once the recording is completely written to the movie file, it will notify the delegate object by calling the following method of the `AVCaptureFileOutputRecordingDelegate` protocol:

```
func fileOutput(_ output: AVCaptureFileOutput, didFinishRecordingTo outputFileURL:
 URL, from connections: [AVCaptureConnection], error: Error?)
```

As `SimpleVideoCamController` is set as the delegate object, create an extension to adopt the `AVCaptureFileOutputRecordingDelegate` protocol like this:

```
extension SimpleVideoCamController: AVCaptureFileOutputRecordingDelegate {
    func fileOutput(_ output: AVCaptureFileOutput, didFinishRecordingTo outputFile
URL: URL, from connections: [AVCaptureConnection], error: Error?) {
        guard error == nil else {
            print(error ?? "")
            return
        }
    }
}
```

For now we just print out any errors to the console. Later, we will further implement this method for video playback.

Before you're going to test the app, there are a couple of things we need to do. First, insert a line of code in the `viewDidLoad()` method to call `configure()`:

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

    configure()
}
```

Secondly, remember to edit the `Info.plist` file to specify the reason why you need to access the device's camera. Otherwise, you will end up with the following error:

```
[access] This app has crashed because it attempted to access privacy-sensitive dat
a without a usage description.  The app's Info.plist must contain an NSCameraUsage
Description key with a string value explaining to the user how the app uses this d
ata.
```

Open `Info.plist` and insert a row for the key `Privacy - Camera Usage Description`. You can specify your own reason in the value field.

Now, you may have a quick test for the app. When you tap the record button, the app will start recording video (indicated by an animated button). Tapping the button again will stop the recording.

Using AVKit for Video Playback

Now that the app should be able to capture a video and save it to a movie file, the next question is how can you play the video in your app? The iOS SDK provides the AVKit framework for handling video playback. If you have some experience with older versions 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 on iOS. Basically, you just need to use a class named `AVPlayerViewController` to handle the video playback. 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 to 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.

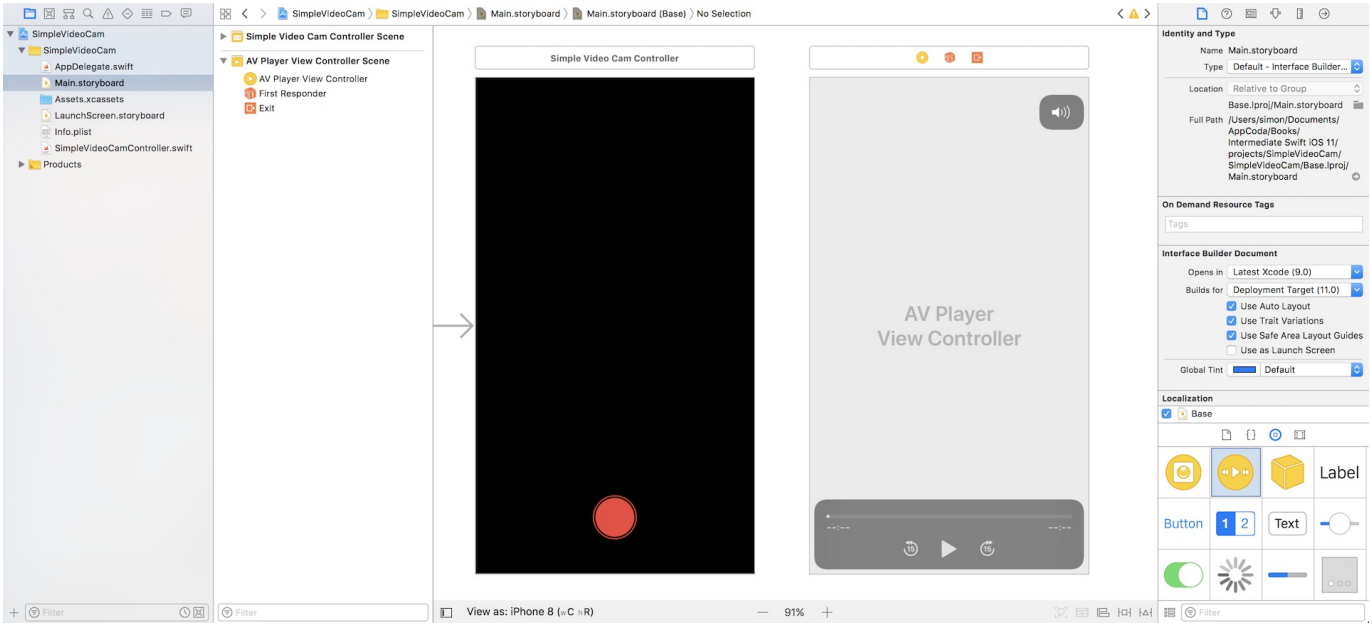


Figure 14.2. Adding the AV Player View Controller to the storyboard

Next, connect the Simple Video Cam Controller to the AV Player View Controller using a segue. In the Document Outline, control-drag from the Simple Video Cam 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`.

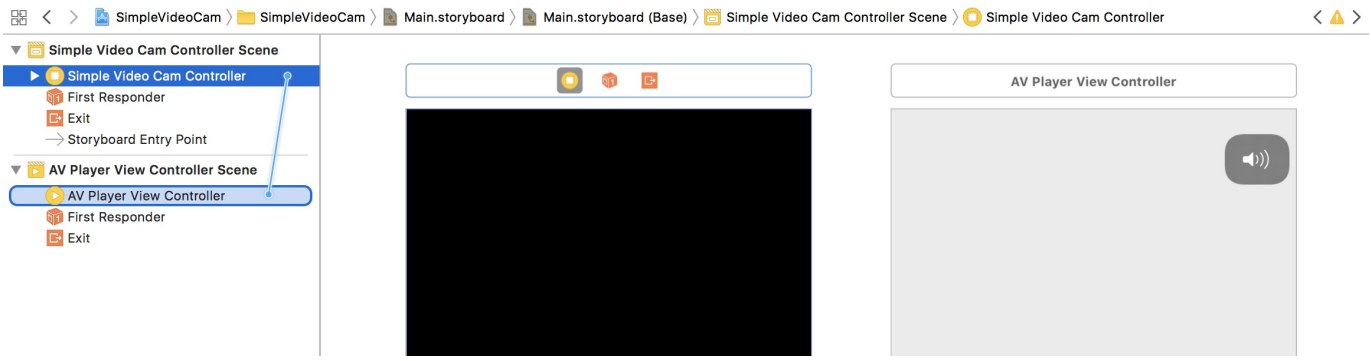


Figure 14.3. Connecting the two view controllers using a segue

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 `fileOutput(_:didFinishRecordingTo:from:error:)` method of the delegate object, once the video has been completely written to a movie file.

The `SimpleVideoCamController` is now assigned as the delegate object, which has adopted the `AVCaptureFileOutputRecordingDelegate` protocol. However, we haven't provided any detailed implementation to take care of the video playback. Now let's do it.

Open the `SimpleVideoCamController.swift` file, import `AVKit`:

```
import AVKit
```

Next, update the delegate method of the extension like this:

```
extension SimpleVideoCamController: AVCaptureFileOutputRecordingDelegate {
    func fileOutput(_ output: AVCaptureFileOutput, didFinishRecordingTo outputFile
URL: URL, from connections: [AVCaptureConnection], error: Error?) {
        guard error == nil else {
            print(error ?? "")
            return
        }

        performSegue(withIdentifier: "playVideo", sender: outputFileURL)
    }
}
```

And then insert the following segue method in the `SimpleVideoCamController` class:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "playVideo" {
        let videoPlayerViewController = segue.destination as! AVPlayerViewControll
er
        let videoFileURL = sender as! URL
        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 `performSegue(withIdentifier:sender:)` method with the video file URL.

In the `performSegue(withIdentifier:sender:)` 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.

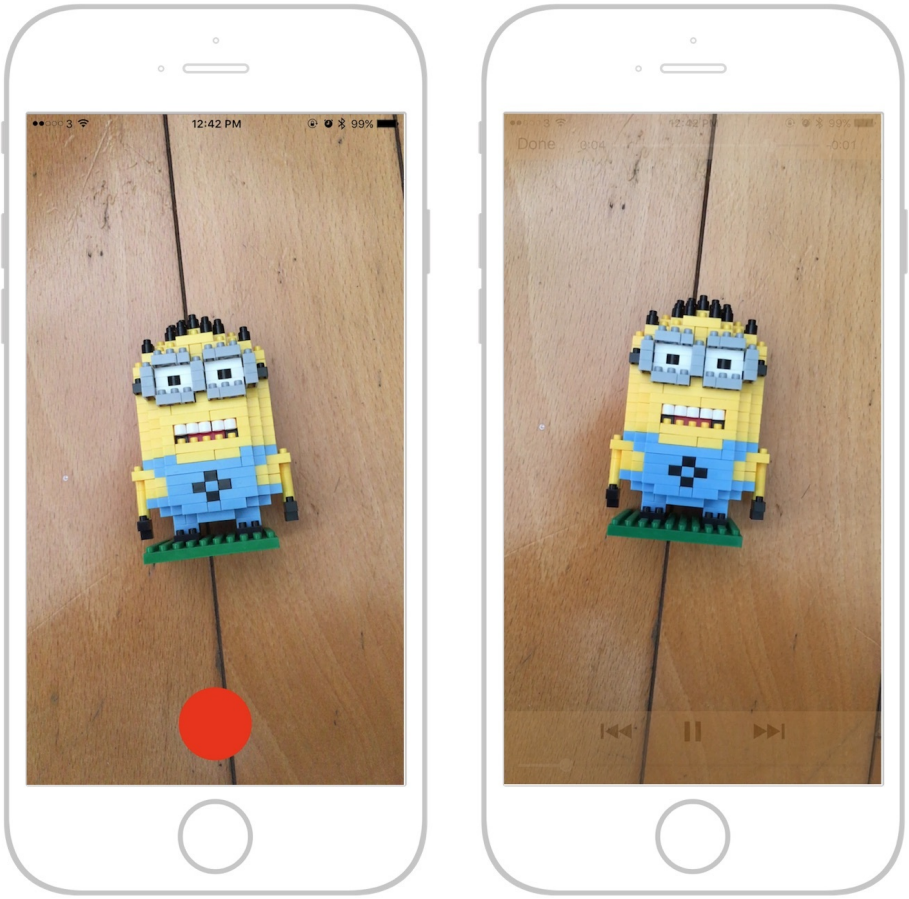
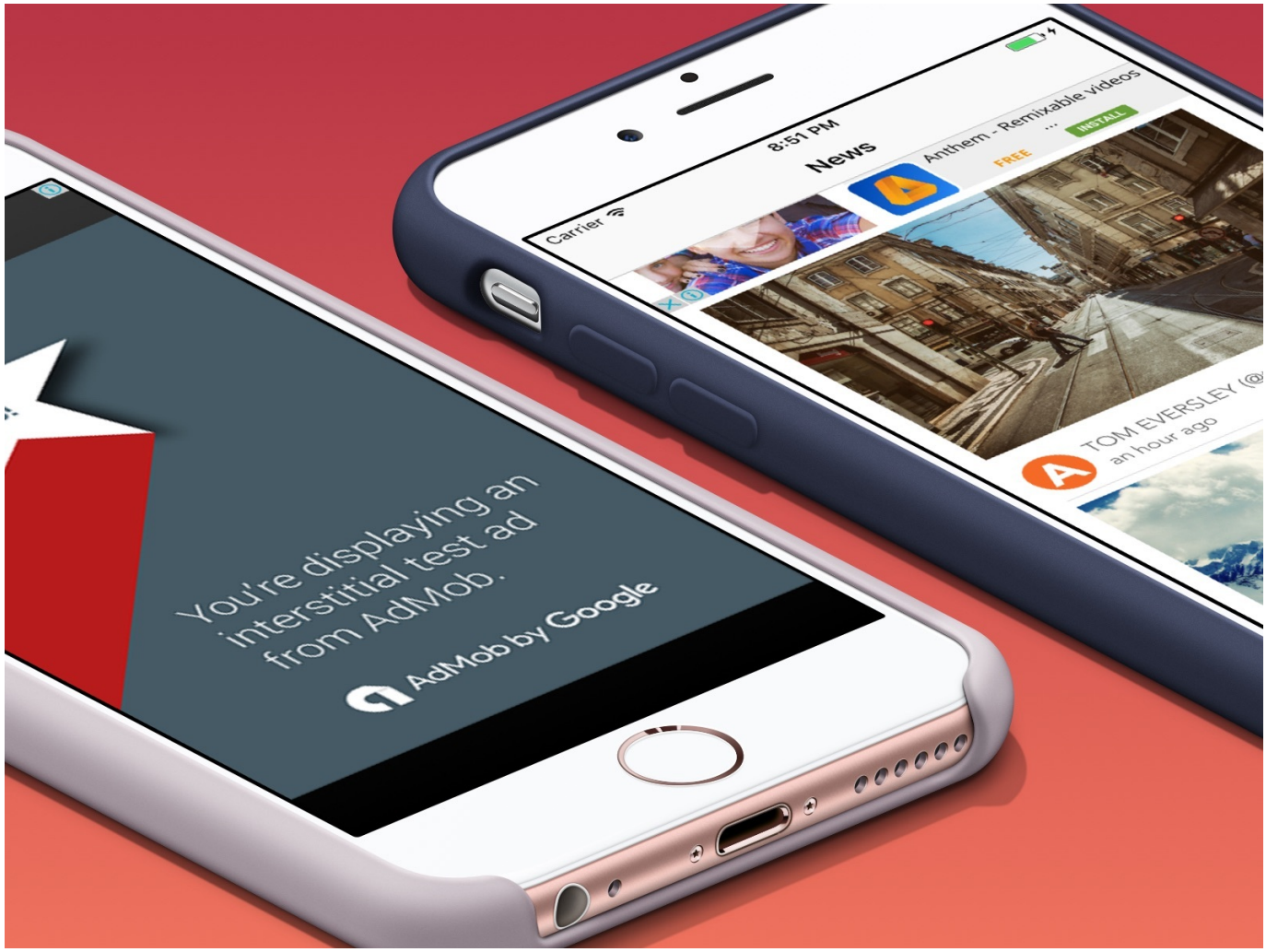


Figure 14.4. Testing the video camera app

For reference, you can download the Xcode project from http://www.appcoda.com/resources/swift5/SimpleVideoCam.zip.

Chapter 15
Displaying Banner Ads using Google AdMob



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 Google AdMob.

Hey, why Google AdMob? We're developing iOS apps. Why don't we use Apple's iAd?

Apple discontinued its iAd App Network on June 30, 2016. Therefore, you can no longer use iAd as your advertising solution for iOS apps. You have to look for other alternatives for placing banner ads.

Among all the mobile ad networks, it is undeniable that Google's AdMob is the most popular one. Similar to iAd, Google provides SDK for developers to embed ads in their iOS app. Google sells the advertising space (e.g. banner) within your app to a bunch of advertisers. You earn ad revenue when a user views or clicks your ads.

To use AdMob in your apps, you will need to use the Google Mobile Ads SDK. The integration is not difficult. To display a simple ad banner, it just takes a few lines of code and you're ready to start making a profit from your app.

There is no better way to learn the AdMob integration than by trying it out. As usual, we'll work on a sample project and then add a banner ad. You can download the Xcode project template from http://www.appcoda.com/resources/swift5/GoogleAdDemoStarter.zip.

On top of the AdMob integration, you will also learn how to perform *lazy* loading in Swift.

Apply a Google AdMob Account

Before you can integrate your apps with Google AdMob, you'll need to first enroll into the AdMob service. Now open the link below using Safari or your favorite browser:

https://www.google.com/admob/

As AdMob is now part of Google, you can simply sign in with your Google account or register a new one. AdMob requires you to have a valid AdSense account and AdWords account. If you don't have one or both of these accounts, follow the sign-up process and connect them to your Google Account.

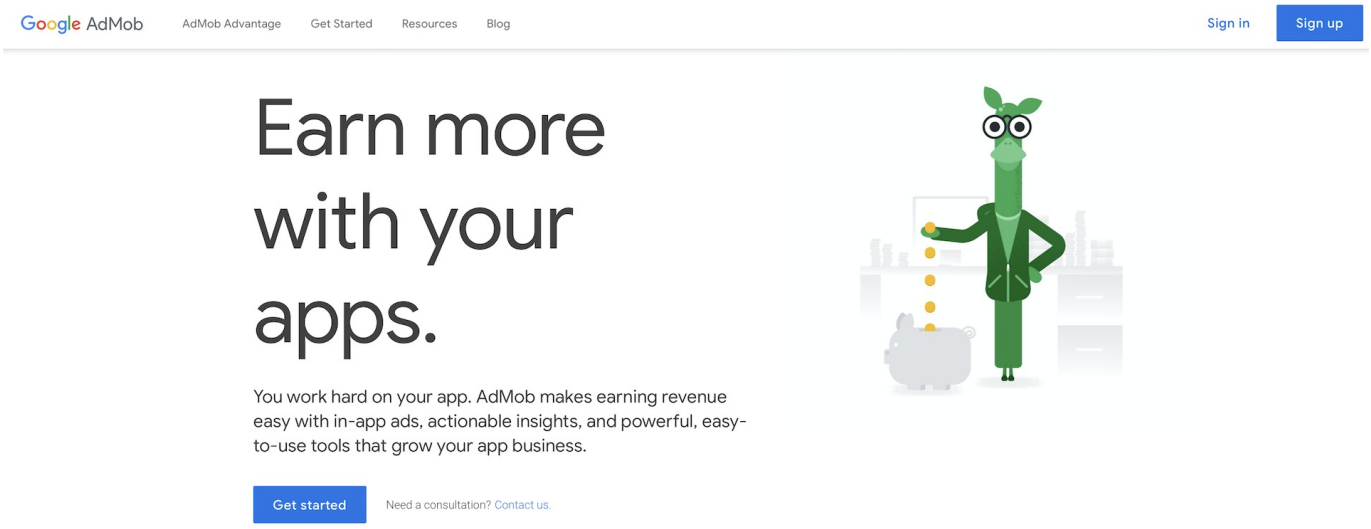


Figure 15.1. Sign into Google AdMob

Once you finish the registration, you will be brought to the AdMob dashboard. In the navigation on your left, select the `Apps` option.

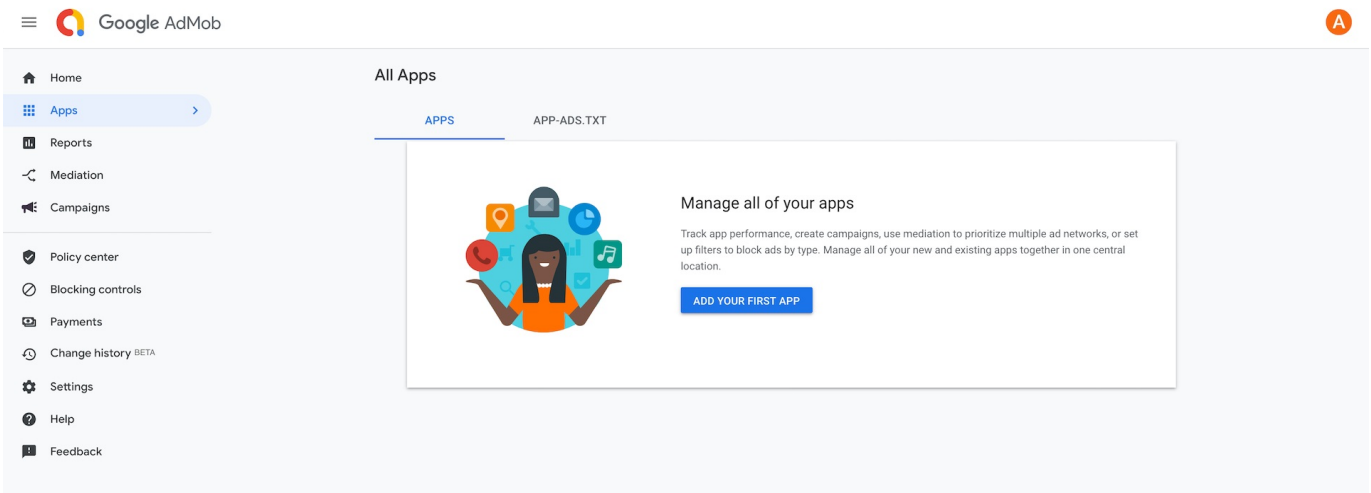


Figure 15.2. AdMob Dashboard

Here, choose the *Add Your First App* option. AdMob will first ask you if your app has been published on the App Store. Assuming your app has not been published, choose the option "No". We will register the app by filling in the form manually. In future, if you already have an app on the App Store, you can let AdMob retrieve your app information.

Set the app name to *GoogleAdMobDemo* and choose iOS for the platform option. Click `Add` to proceed to the next step. AdMob will then generate an App ID for the app. Please make a note of this app ID. We will need to add it to our app in order to integrate with AdMob.

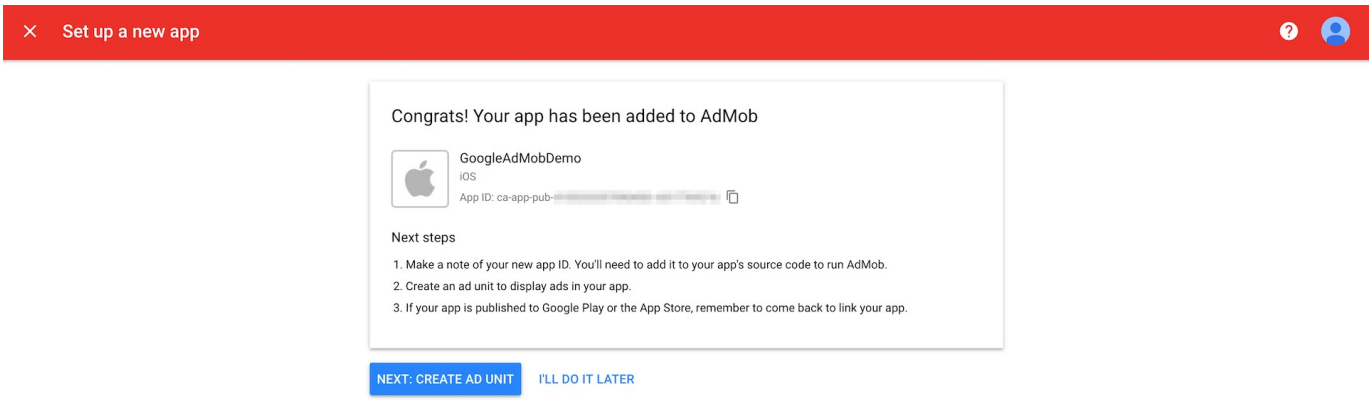


Figure 15.3. Your AdMob App ID

Next, we need to create at least an ad unit. Click `Next: Create Ad Unit` to proceed. In this demo, we use the banner ad. Select *Banner* and accept the default options. For the Ad unit name, set it to *AdBanner*.

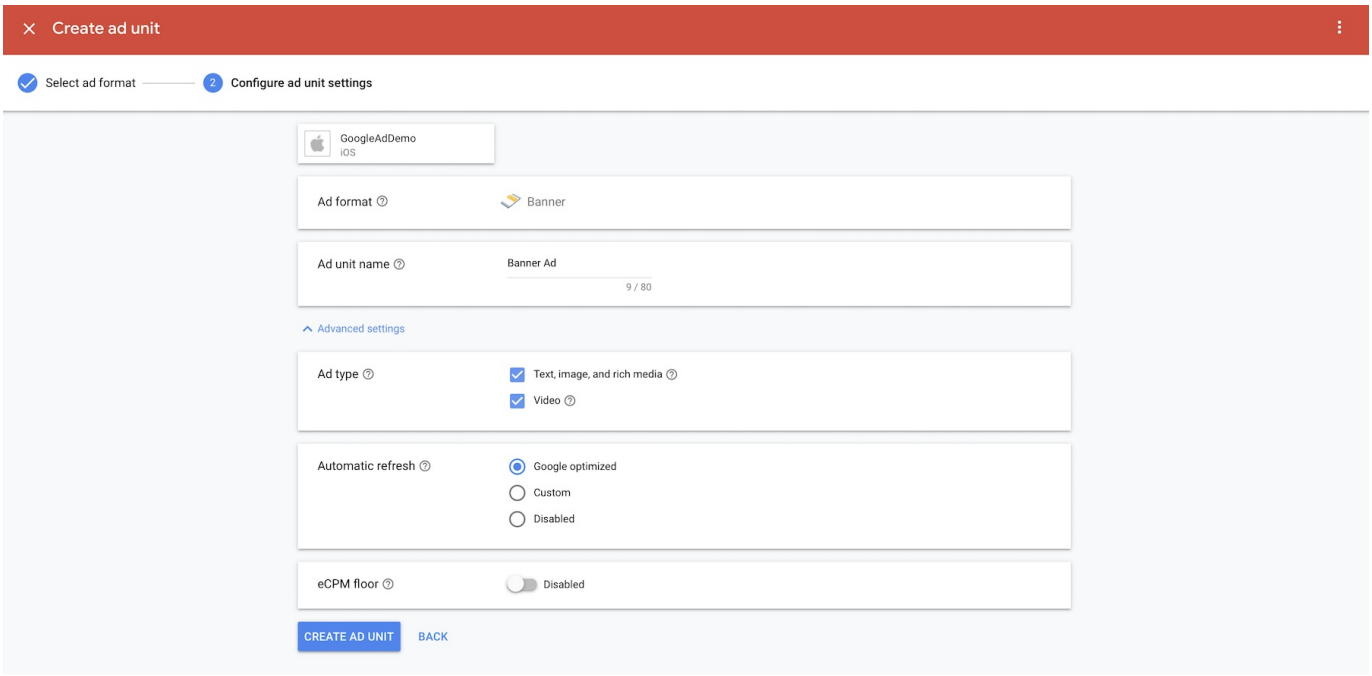


Figure 15.4. Create a banner ad

Click `Save` to generate the ad unit ID. This completes the configuration of your new app. You will find the App ID and Ad unit ID in the implementation instructions. Please save these information. We will need them in the later section when we integrate AdMob with our Xcode project.

However, you can skip the download of the Google Mobile Ads SDK. The starter project already bundles the SDK for you. In case if you need the SDK for your own project, I would recommend you to use CocoaPods to install the SDK. We will discuss more about it in the next section.

Using Google Mobile Ads Framework

Now that you have completed the configuration in AdMob, let's move to the actual implementation. Fire up Xcode and open `GoogleAdDemo.xcworkspace` of the starter project. Please note that it is `GoogleAdDemo.xcworkspace` instead of `GoogleAdDemo.xcodeproj` 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.

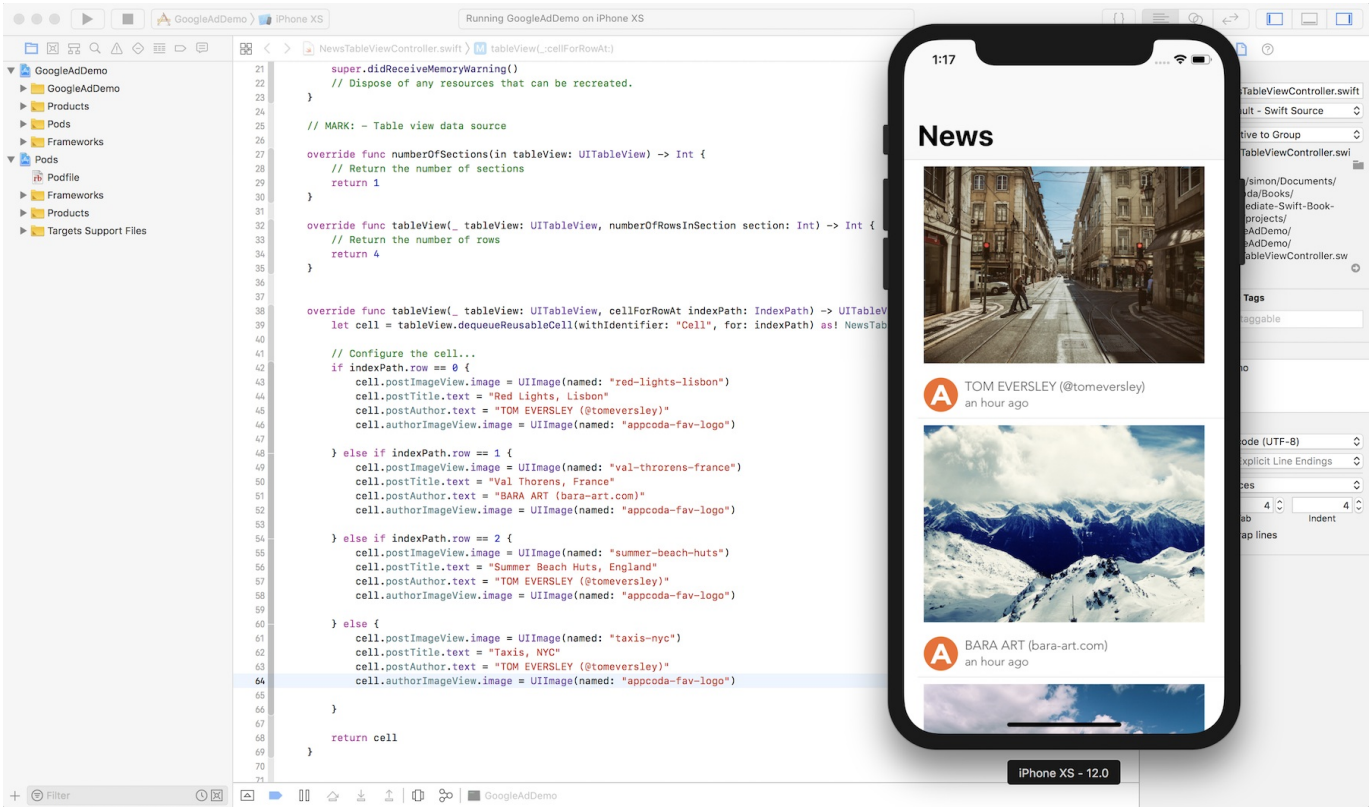


Figure 15.5. Demo app

To integrate Google AdMob into your app, the first thing you need to do is install the Google Mobile Ads framework into the Xcode project. For the starter project, I have already added the framework using CocoaPods. In brief, you need to create a Podfile in your Xcode project, and add the following line to your app's target in the Podfile:

```
pod 'Google-Mobile-Ads-SDK'
```

Then you run `pod install` to grab the SDK, and let CocoaPods integrate the SDK into your Xcode project. Anyway, I assume you use the starter project to follow this chapter.

In the starter project, if you look closely at the project navigator, you will find two projects: *GoogleAdDemo* and *Pods*. The former is the original project, while the *Pods* is the project that bundles the Google Mobile Ads SDK. For details about how to install CocoaPods and use it to install the SDK, I recommend you to check out chapter 33. We will discuss CocoaPods in details.

To use the Google Mobile Ads SDK in your code, you will have to import the framework and register your App ID. We will do the initialization in the `AppDelegate.swift` file. Insert the import statement at the beginning of the file:

```
import GoogleMobileAds
```

Next, insert the following line of code in the `application(_:didFinishLaunchingWithOptions:)` method:

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

    // Override point for customization after application launch.

    GADMobileAds.sharedInstance().start(completionHandler: nil)

    return true
}
```

Initializing the Google Mobile Ads SDK at app launch allows the SDK to perform configuration tasks as early as possible. Next, open `Info.plist` and add a new row named `GADApplicationIdentifier`. Recall that AdMob created a unique application ID for your application, you have to set the ID in this entry. For me, I set it to:

```
ca-app-pub-8501671653071605~9497926137
```

Please make sure you replace the App ID with yours.

App Transport Security (ATS) is enabled by default on iOS. To prevent iOS from blocking the Google Ads, you are also required to create a new row named "App Transport Security Settings" and enable *Arbitrary Loads*. You can refer to figure 15.6 for the settings.

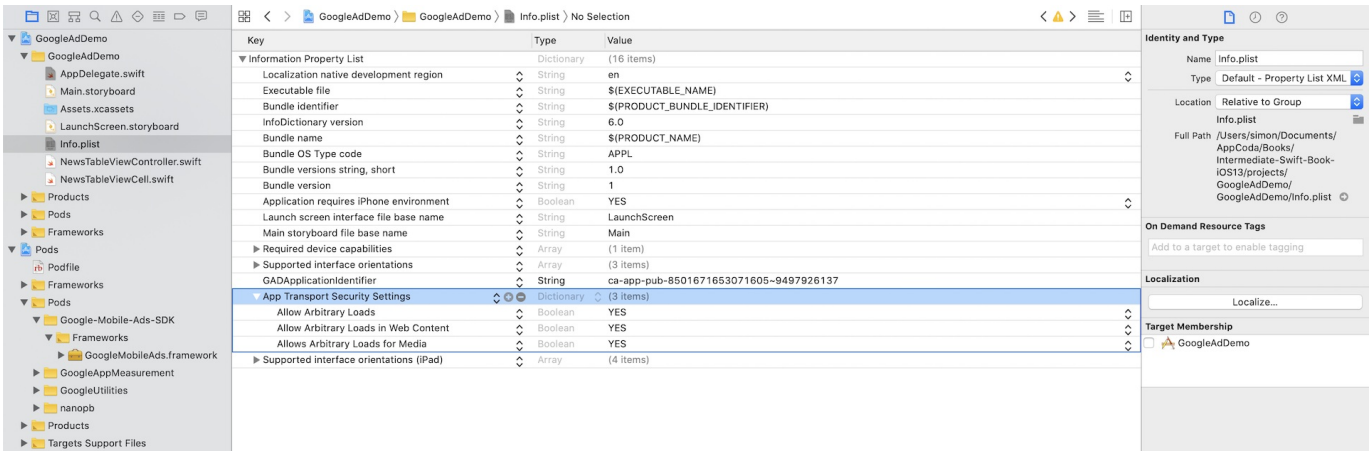


Figure 15.6. Setting App Transport Security

Displaying Banner Ads at the Table View Header

Let's start with the simplest way to display a banner ad in your app. We will request an ad banner from Google and display the ad at the table header.

To display a banner ad at that position, all you need to do is create a `GADBannerView` object, set its delegate, and root view controller. Then you call its `load` method with an ad request to retrieve a banner ad. When the ad is ready to display, it will call the `adViewDidReceiveAd(bannerView:)` method of the `GADBannerViewDelegate` protocol. So you just need to implement the method to show the banner ad in the table view header.

Okay, let's now go into the implementation.

Now open `NewsTableViewController.swift`. First, import the *GoogleMobileAds* framework:

```
import GoogleMobileAds
```

Next, declare a variable of the type `GADBannerView`. This is the variable for holding the banner view:

```
lazy var adBannerView: GADBannerView = {
    let adBannerView = GADBannerView(adSize: kGADAdSizeSmartBannerPortrait)
    adBannerView.adUnitID = "ca-app-pub-8501671653071605/1974659335"
    adBannerView.delegate = self
    adBannerView.rootViewController = self

    return adBannerView
}()
```

In the code above, we use a closure to initialize the `adBannerView` variable, which is an instance of `GADBannerView`. During the initialization, we tell the SDK that we want to retrieve a smart banner (`kGADAdSizeSmartBannerPortrait`). Smart banners, as the name suggests, are ad units that are clever enough to detect the screen width and adjust its size accordingly. We also set the ad unit ID, delegate and root view controller. Again, please replace the ad unit ID with yours.

We use lazy initialization (sometimes it is called lazy instantiation or loading) to initialize the `adBannerView` variable. In Swift, you use the `lazy` keyword to indicate that the variable can be initialized later. More specificially, the variable will only be instantiated when it is used. This technique for delaying an object creation is especially useful when it takes a considerable time to load an object or the object you're referring to is not ready at the time of object creation. During initialization, we set the `delegate` and `rootViewController` properties to `self`. As the `NewsTableViewController` is not ready at the time, we use `lazy` to defer the initialization of `adBannerView`.

Is it a must to use lazy initialization for creating a banner view? No, I want to take this chance to introduce you lazy initialization, and demonstrate how to use a closure for variable initialization. You can do the same without using lazy initialization like this:

```
var adBannerView: GADBannerView?

override func viewDidLoad() {
    super.viewDidLoad()

    adBannerView = GADBannerView(adSize: kGADAdSizeSmartBannerPortrait)
    adBannerView?.adUnitID = "ca-app-pub-8501671653071605/1974659335"
    adBannerView?.delegate = self
    adBannerView?.rootViewController = self
}
```

However, as you can see, the former way of initialization allows us group all the initialization code in the closure. The code is more readable and manageable.

Now that we have created the `adBannerView` variable, the next thing is to request the ad. To do that, all you need to do is add the following lines of code in the `viewDidLoad` method:

```
GADMobileAds.sharedInstance().requestConfiguration.testDeviceIdentifiers = [ "2077
ef9a63d2b398840261c8221a0c9b" ]
adBannerView.load(GADRequest())
```

To load a production ad on a test environment, Google requires you to set your test device. This is the purpose of the first line of the code. If you want to test it on your iPhone/iPad, please make sure you replace the sample ID with your own device ID. You can find the device ID by connecting your iPhone to Mac and then open *Window* > *Device and Simulator* in Xcode.

You may wonder why you need to define the test devices. What if you omit that line of code? Your app will work and display ads. But it's Google's policy that you have to comply to.

> Once you register an app in the AdMob UI and create your own ad unit IDs for use in your app, you'll need to explicitly configure your device as a test device when you're developing. This is **extremely important**. Testing with real ads (even if you never tap on them) is against AdMob policy and can cause your account to be suspended.
>
> - AdMob Integration Guide

Lastly, we need to adopt the `GADBannerViewDelegate` protocol. We will create an extension to adopt the protocol and implement two optional methods like this:

```
extension NewsTableViewController: GADBannerViewDelegate {

    func adViewDidReceiveAd(_ bannerView: GADBannerView) {
        print("Banner loaded successfully")
        tableView.tableHeaderView?.frame = bannerView.frame
        tableView.tableHeaderView = bannerView
    }

    func adView(_ bannerView: GADBannerView, didFailToReceiveAdWithError error: GA
DRequestError) {
        print("Fail to receive ads")
        print(error)
    }
}
```

When the ad is successfully loaded, the `adViewDidReceiveAd` method is called. In the method, we simply assign the banner view to the table's header view. This allows the app to show the banner ad in the table header. If the ad is failed to load, we just print the error message to console.

Try to run the demo app and play around with it. When the app is launched, you will see a banner ad at the top of the table view.

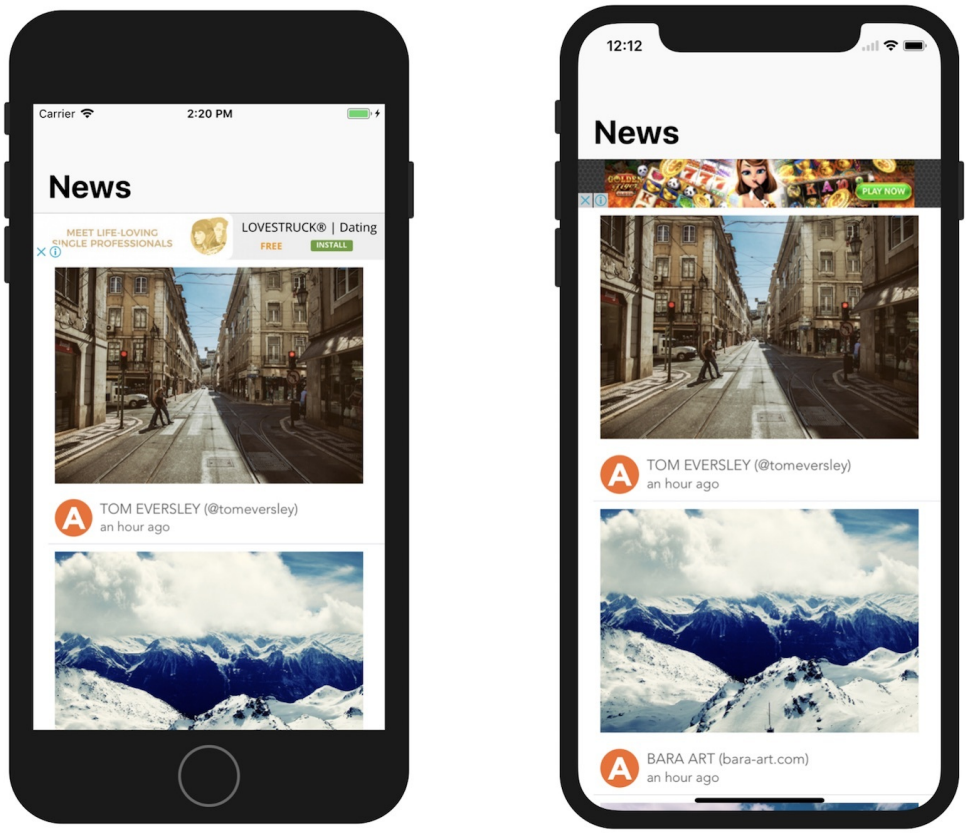


Figure 15.7. Banner ads at the table view header

> **Note:** If your app fails to retrieve the ad, please try to change the value of *adUnitId* to ca-app-pub-3940256099942544/2934735716.

Adding a Subtle Animation

Sometimes adding a subtle animation to banner ads can give a better user experience about how the ad transitions onto the screen. In this section, I will show you how to animate the banner ad. We will add a slide-down animation when the ad transitions onto the screen.

The trick is to apply `UIView` animations to the ad banner. When the ad is first loaded, we reposition the ad banner off the screen. Then we bring it back using a slide down animation.

As mentioned before, the `adViewDidReceiveAd` method is called when an ad is ready. To animate the ad banner, all we need to do is modify the method like this:

```
func adViewDidReceiveAd(_ bannerView: GADBannerView) {
    print("Banner loaded successfully")

    // Reposition the banner ad to create a slide down effect
    let translateTransform = CGAffineTransform(translationX: 0, y: -bannerView.bou
nds.size.height)
    bannerView.transform = translateTransform

    UIView.animate(withDuration: 0.5) {
        self.tableView.tableHeaderView?.frame = bannerView.frame
        bannerView.transform = CGAffineTransform.identity
        self.tableView.tableHeaderView = bannerView
    }
}
```

We first create a `translateTransform` to move the banner view off the screen. We then call `UIView.animate` to slide the banner down onto the screen.

Run the project again to test the app. The ad will be displayed with an animated effect.

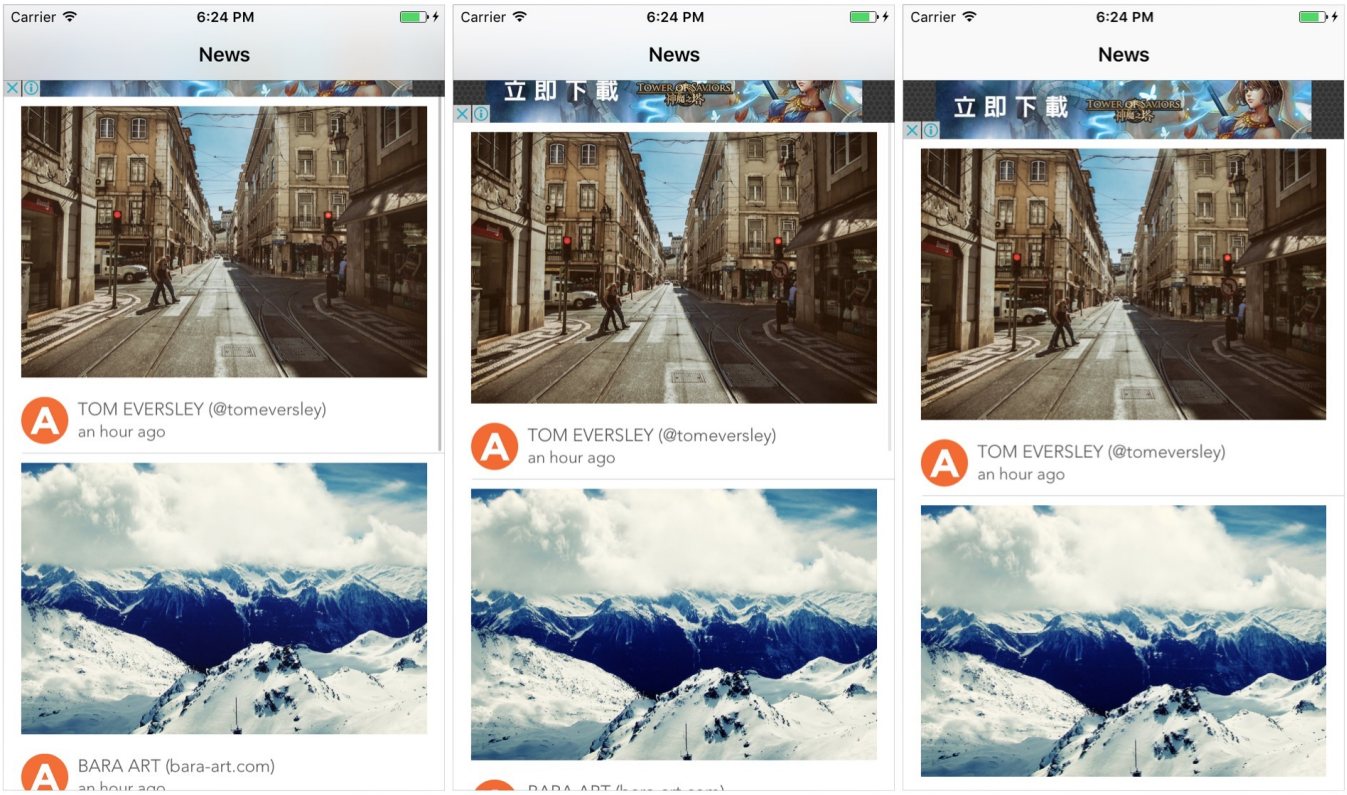


Figure 15.8. Animating the ad banner

Displaying a Sticky Banner Ad

As you scroll through the table view, the ad banner disappears. It doesn't stick to the table header. You may wonder how you can display a sticky banner ad. That's what we're going to explore in this section.

The banner ad is now inserted into the table header view. If you want to make it sticky, you can add it to the section's header view instead of the table's header view.

Let's see how to implement it.

Insert the following methods in the `NewsTableViewController` class:

```
override func tableView(_ tableView: UITableView, viewForHeaderInSection section:
Int) -> UIView? {
    return adBannerView
}

override func tableView(_ tableView: UITableView, heightForHeaderInSection section
: Int) -> CGFloat {

    return adBannerView.frame.height
}
```

We override the default `tableView(_:viewForHeaderInSection:)` method with our own method to return the ad banner view.

The default height of the header is too small for the ad banner. So we also override the `tableView(_:heightForHeaderInSection:)` method and return the height of the banner view frame.

Lastly, modify the `adViewDidReceiveAd` method like this:

```
func adViewDidReceiveAd(_ bannerView: GADBannerView) {
    print("Banner loaded successfully")

    // Reposition the banner ad to create a slide down effect
    let translateTransform = CGAffineTransform(translationX: 0, y: -bannerView.bou
nds.size.height)
    bannerView.transform = translateTransform

    UIView.animate(withDuration: 0.5) {
        bannerView.transform = CGAffineTransform.identity
    }
}
```

We just remove those lines of code that are related to table view header, which are no longer necessary.

That's it. It is now ready to test the app again. The ad banner displays at the fixed position.

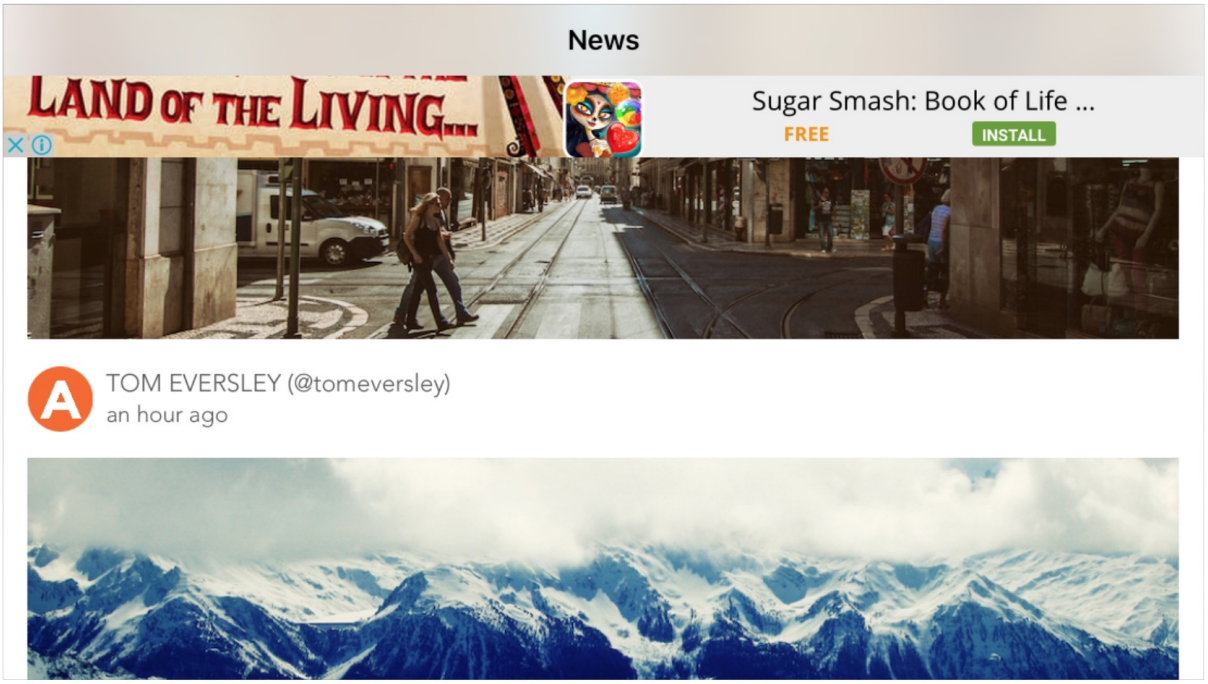


Figure 15.9. Displaying a sticky banner ad

Working with Interstitial Ads

Not only can you include banner ads in your apps, but the Google Mobile Ads SDK also lets you easily display interstitial ads (i.e. full screen ads). Typically you can earn more ad revenue through interstitial ads as it completely blocks out the app's content and catches users' attention.

The downside of this kind of mobile ad is sometimes irritating as it forces users to view the ad until they click out. But it really depends how frequent and at what point you display the full screen ad. I have used some apps that keep displaying interstitial ads every couple of minutes. A well-thought ad placement can make your app less irritating. For example, you only display an ad once or between game levels if you're developing a game. Anyway, my focus here is to show you how to display an interstial ad in iOS apps. My plan is to display the ad right after a user launches the demo app.

To do that, you have to first go back to AdMob's dashboard (https://apps.admob.com) to create an interstial ad for the demo app. Select `Apps` in the side menu and choose `GoogleAdMobDemo`. In the next screen, click `Add Ad Unit` to create a new ad unit.

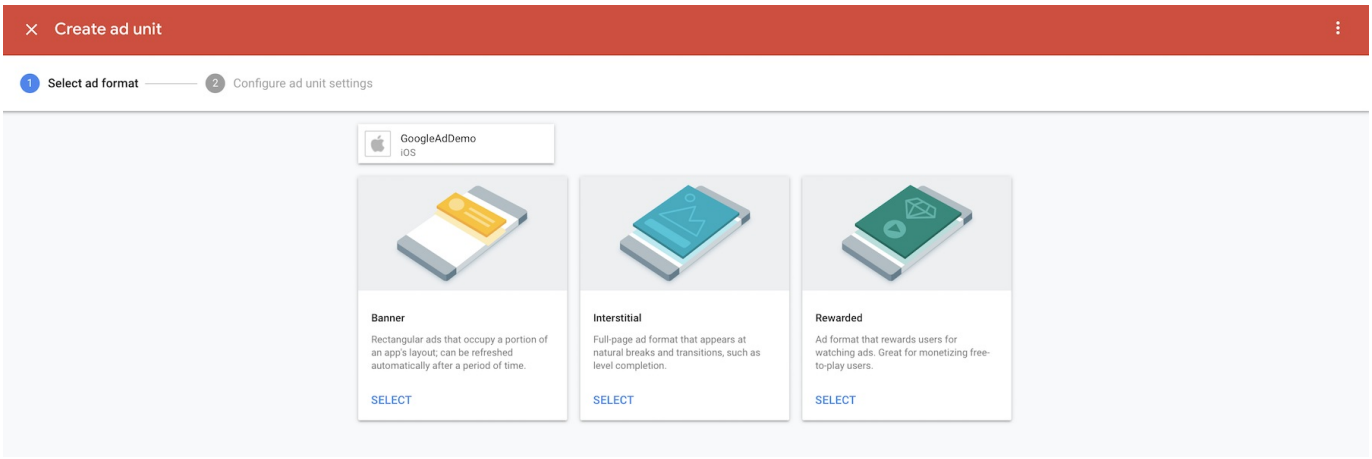


Figure 15.10. Adding a new Interstitial ad unit

This time, we create an interstitial ad. Give the ad unit a name and then click `Save` to create the unit. AdMob should generate another ad unit ID for you.

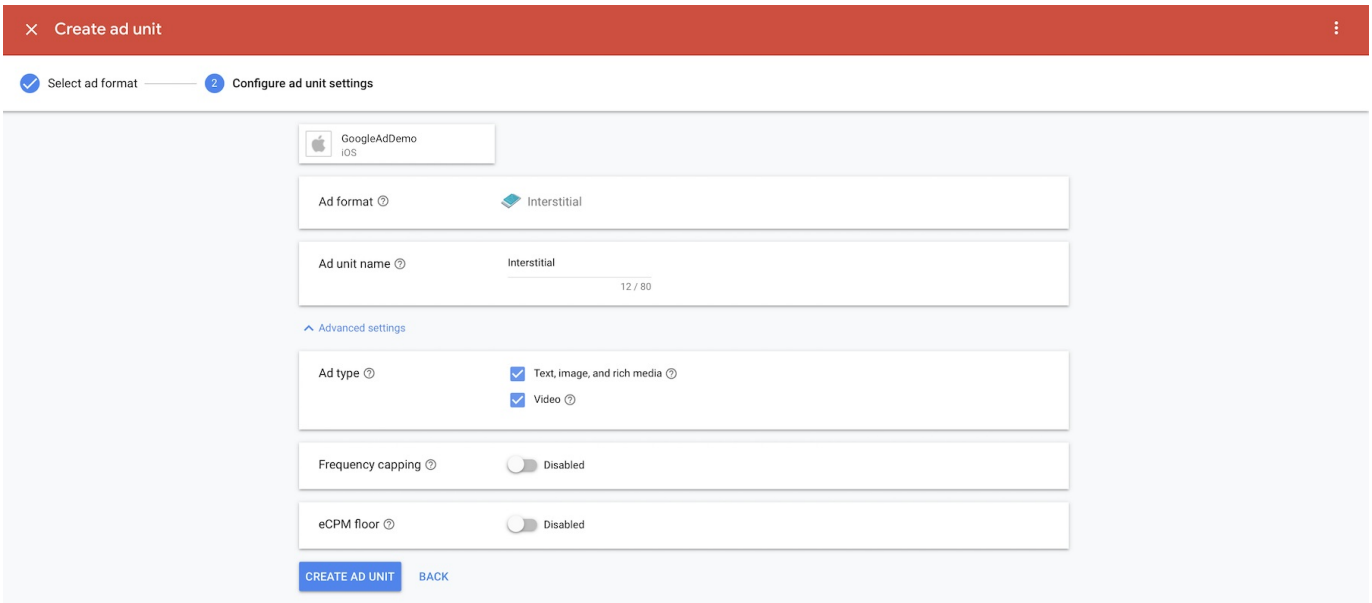


Figure 15.11. Adding an interstitial ad unit

Now go back to the Xcode project.

The code of implementing an interstitial ad is very similar to that of a banner ad. Instead of using the `GADBannerView` class, you use the `GADInterstitial` class. So, declare a variable for storing the `GADInterstitial` object in the `NewsTableViewController` class:

```
var interstitial: GADInterstitial?
```

However, one main difference between `GADBannerView` and `GADInterstitial` is that `GADInterstitial` is a one time use object. That means the interstitial can't be used to load another ad once it is shown.

Due to this reason, we create a helper method called `createAndLoadInterstitial()` to create the ad. Insert the method in the `NewsTableViewController` class:

```
private func createAndLoadInterstitial() -> GADInterstitial? {
    interstitial = GADInterstitial(adUnitID: "ca-app-pub-8501671653071605/2568258533")

    guard let interstitial = interstitial else {
        return nil
    }

    let request = GADRequest()
    interstitial.load(request)
    interstitial.delegate = self

    return interstitial
}
```

We first initialize a `GADInterstitial` object with the ad unit ID (remember to replace it with yours). Then we create a `GADRequest`, call the `load` method and set the delegate to `self`. That's pretty much the same as what we have done for the banner ad. You may notice that we also set the `testDevices` property of the ad request. Without setting its value, you will not be able to load interstitial ads on test devices. Here `kGADSimulatorID` indicates that our test device is the built-in simulator.

We will create the ad when the view is loaded. Insert the following line of code in the `viewDidLoad()` method:

```
interstitial = createAndLoadInterstitial()
```

Similar to `GADBannerView`, we need to adopt a protocol in order to check the status of an ad. Create an extension to implement the `GADInterstitialDelegate` protocol:

```
extension NewsTableViewController: GADInterstitialDelegate {
    func interstitialDidReceiveAd(_ ad: GADInterstitial) {
        print("Interstitial loaded successfully")
        ad.present(fromRootViewController: self)
    }

    func interstitialDidFail(toPresentScreen ad: GADInterstitial) {
        print("Fail to receive interstitial")
    }
}
```

When the interstitial is ready, `interstitialDidReceiveAd(ad:)` will be called. In the method, we call the `present` method of `GADInterstitial` to display the ad on screen.

Now you're ready to test the app. After launching the app, you should see a full-screen test ad.

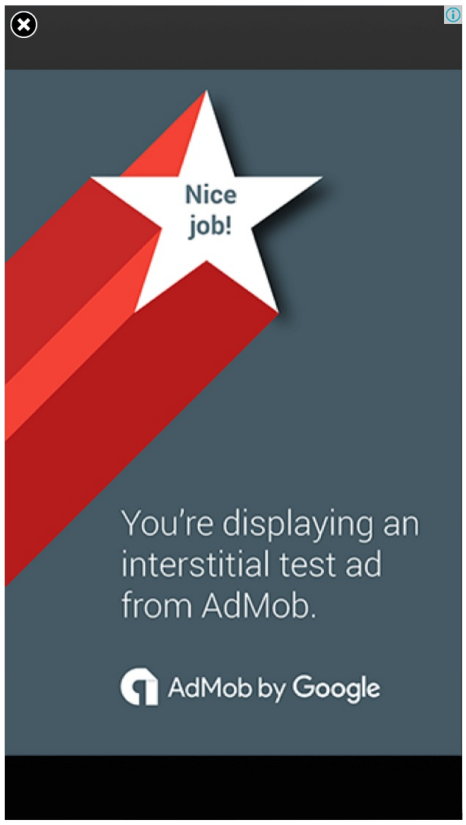


Figure 15.12. A test interstitial ad

Note: If your app fails to retrieve the ad, please try to change the value of *adUnitId* to ca-app-pub-3940256099942544/4411468910.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/GoogleAdDemo.zip.

Chapter 16
Working with Custom Fonts and Dynamic Type

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 http://www.appcoda.com/resources/swift5/CustomFontStarter.zip.

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:

- 100 Greatest Free Fonts Collection for 2015 http://www.awwwards.com/100-greatest-free-fonts-collection-for-2015.html
- The 100 best free fonts http://www.creativebloq.com/graphic-design-tips/best-free-fonts-for-designers-1233380
- A Curated Collection of the 40 Best Google Fonts https://www.typewolf.com/google-fonts
- 100+ Best Fonts Collection for 2016 https://www.designwall.com/blog/100-best-fonts-collection/
- Trendy Google Fonts Combination http://fonts.greatsimple.io/
- 56 best free fonts for designers 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 in 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.

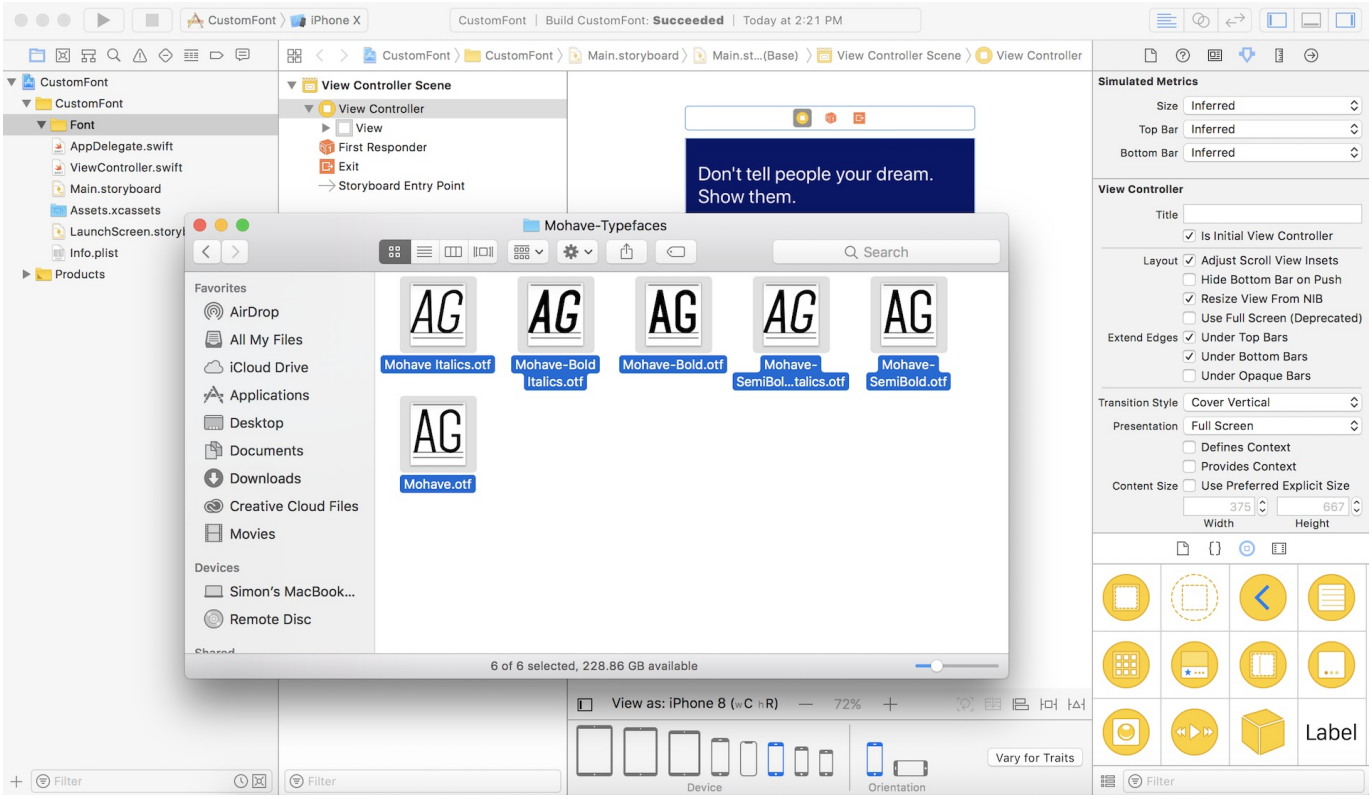


Figure 16.1. Adding fonts file to the Xcode project

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.

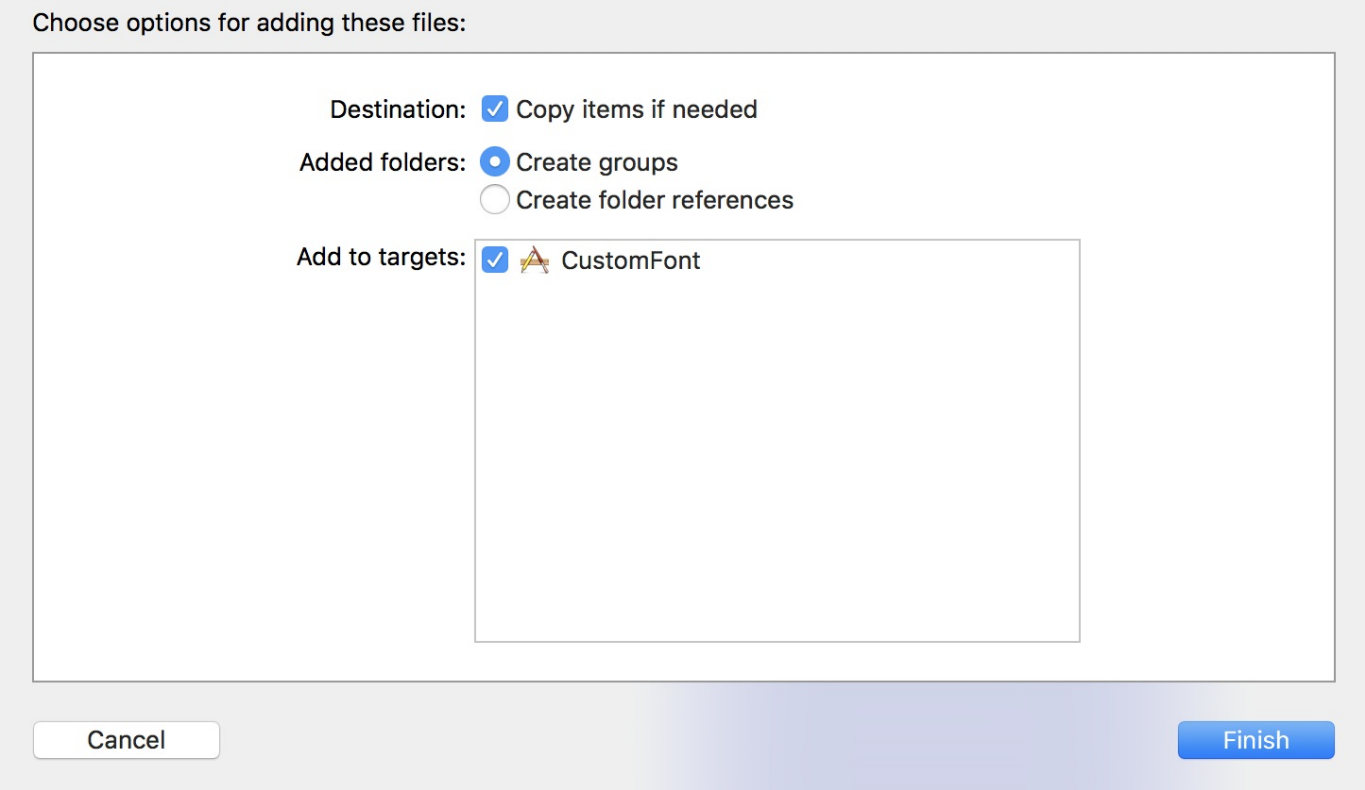


Figure 16.2. Choose options for adding the files

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.

Figure 16.3. Previewing the fonts in Xcode

One thing to make sure is that the checkbox of *CustomFont* in the Target Membershiop section should be enabled.

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 drop down menu. Click the disclosure icon (i.e. triangle) to expand the key. You should see Item 0. Double click the value field and enter *Hallo 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.

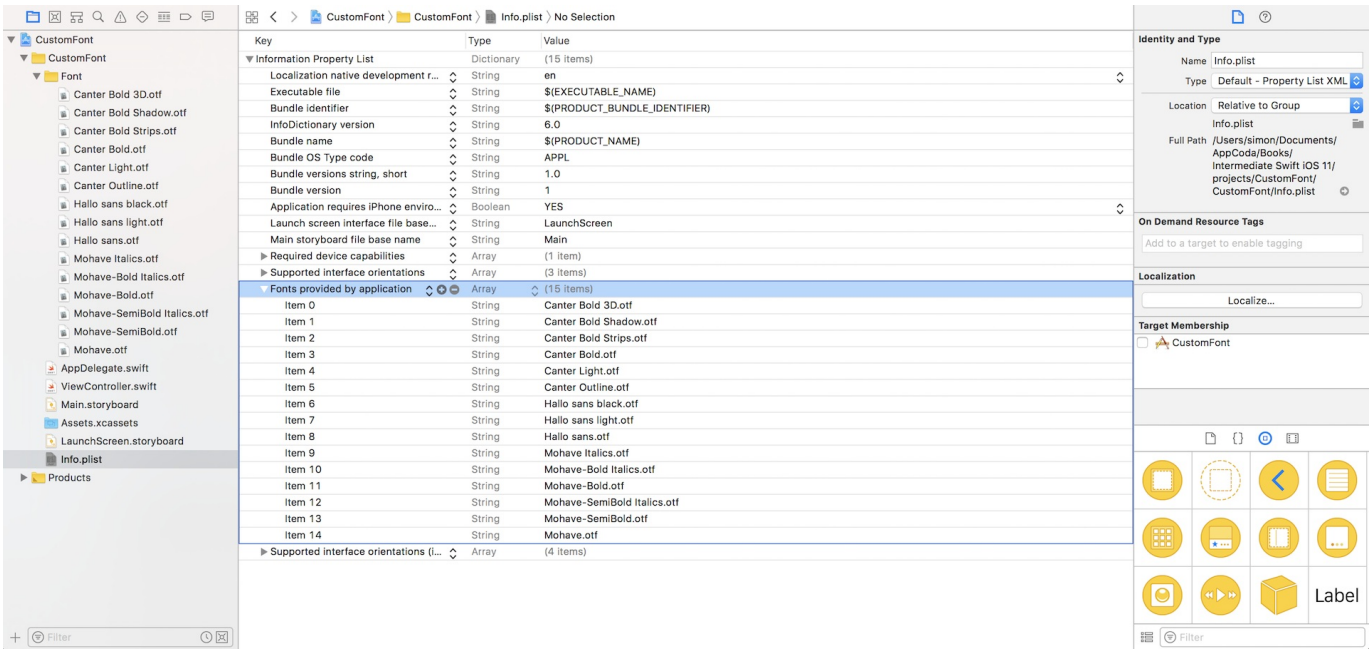


Figure 16.4. Configuring the font files in Info.plist

Using Custom Fonts in Interface Builder

Once you embed the fonts files in your project, Xcode allows you 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.

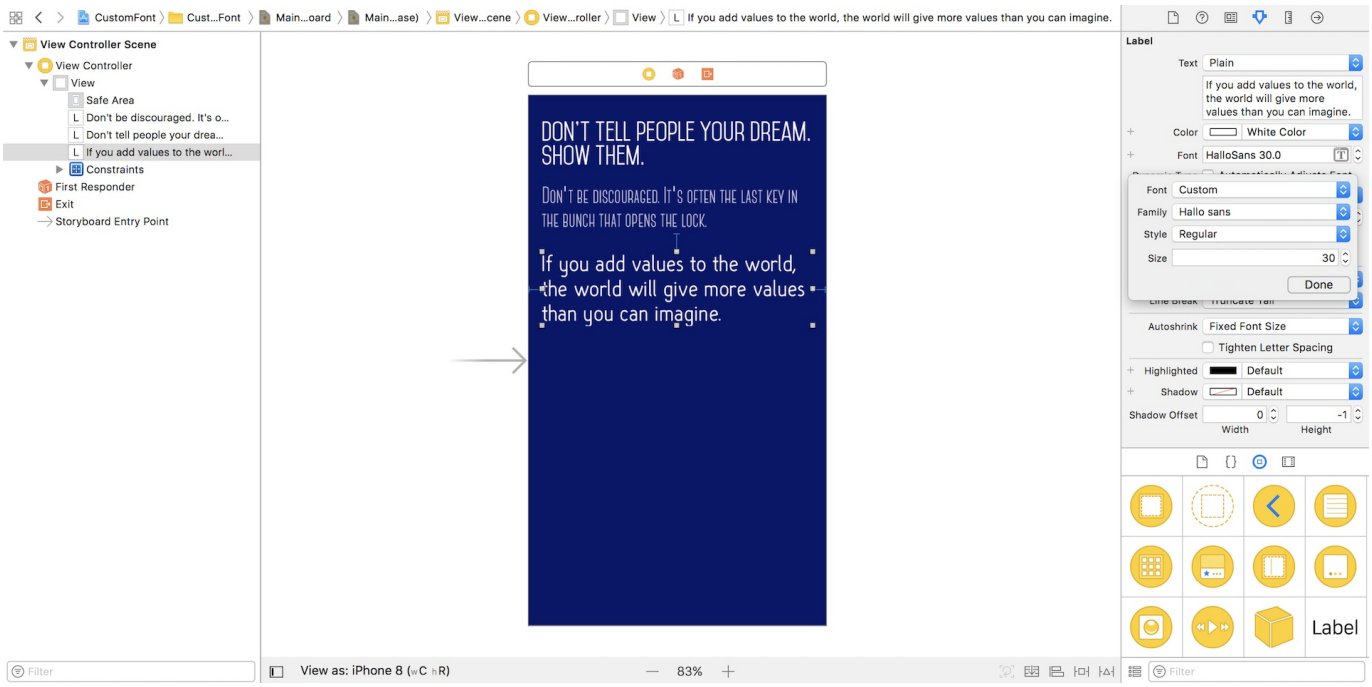


Figure 16.5. Using custom fonts in Interface Builder

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*.

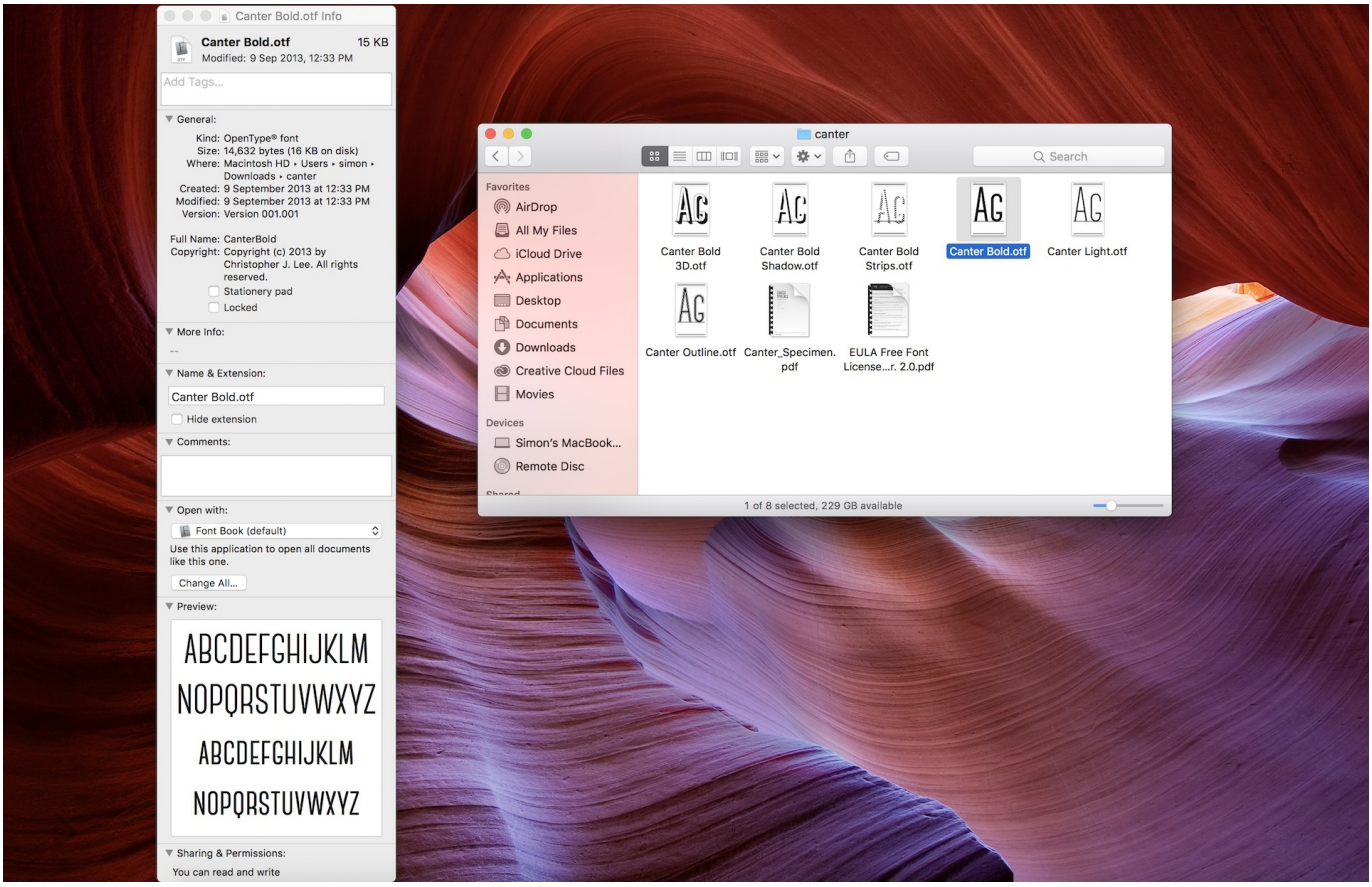


Figure 16.6. Finding out the font name

Working with Dynamic Type

Dynamic Type is not something new. It has been around since the release of iOS 7. With Dynamic Type, users are able to choose their own font size, as long as the app supports Dynamic Type.

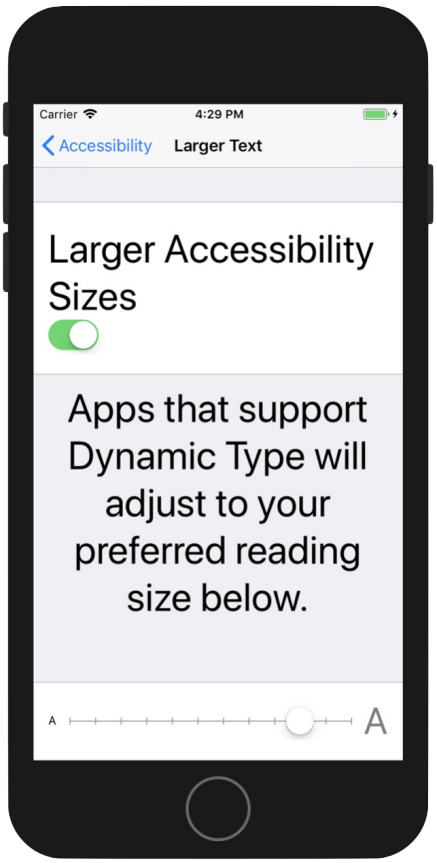


Figure 16.7. Enable large font size

So far, all the labels we worked with are of a fixed font size. Even if you go to *Settings* > *General* > *Accessibility* and enable *Larger Text*, you will not be able to enlarge the font size of the demo app.

Enabling Dynamic Type

To make your app work with Dynamic Type, you have to set the font of the labels to a text style (instead of a specific font). Similar to what we have done before, you can change the font style through Interface Builder or using code. If you go to `Main.storyboard`, select one of the labels. You can change its font in the Attributes inspector. Instead of using a custom font, choose a text style (say, Title 1).

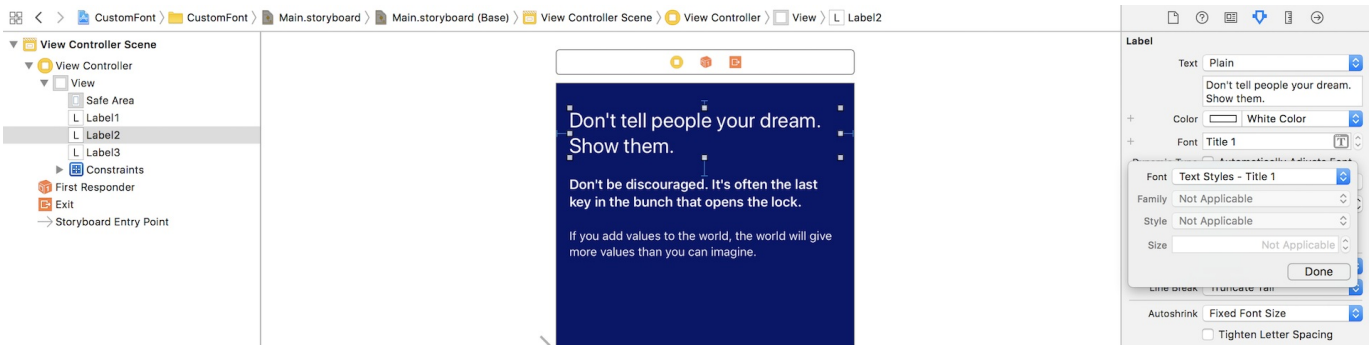


Figure 16.8. Changing the label to a text style

If you prefer to set the font programmatically, replace the code in the `viewDidLoad()` method of `ViewController.swift` with the following:

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

    label1.font = UIFont.preferredFont(forTextStyle: .title1)
    label2.font = UIFont.preferredFont(forTextStyle: .headline)
    label3.font = UIFont.preferredFont(forTextStyle: .subheadline)

}
```

The `preferredFont` method of `UIFont` accepts one of the following text style:

- `.largeTitle`
- `.title1`, `.title2`, `.title3`
- `.caption1`, `.caption2`
- `.headline`
- `.subheadline`
- `.body`
- `.callout`
- `.footnote`

Now open the simulator and go to Setting to enable larger text (see figure 16.7). Once configured, run the app to have a look. You will see the labels are enlarged.

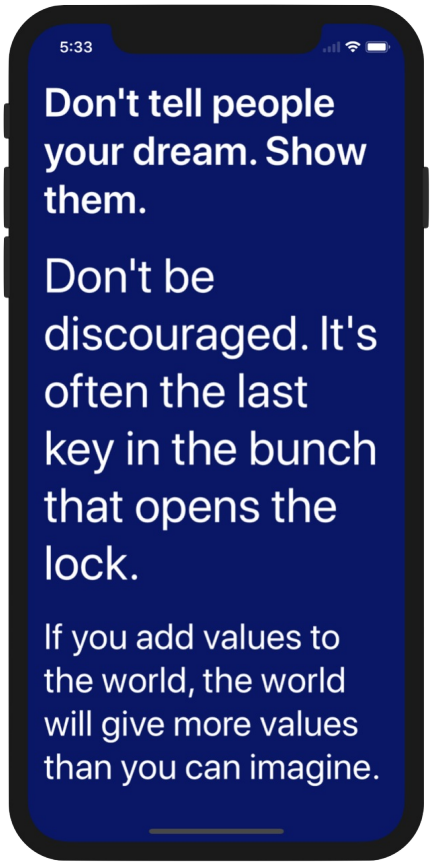


Figure 16.9. Dynamic Type in action

Adjust the Font Automatically

If you go back to Settings to adjust the preferred font size and reopen the demo app, the size of the font is unchanged. Right now, the size of the font keeps intact once the app is launched.

How can you enable the app to adjust the font size whenever the user changes the preferred font size in Settings?

If you use Interface Builder, select the label and go to the Attributes inspector. Tick the checkbox of the *Automatically Adjusts Font* option. The label will now adjust its font size automatically.

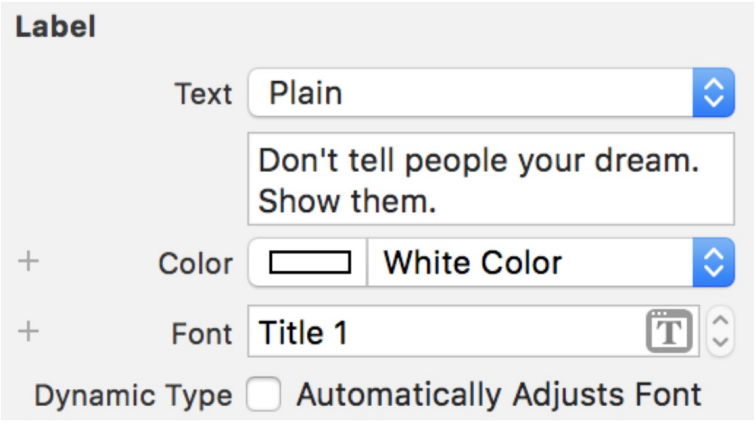


Figure 16.10. Enable the Automatically Adjusts Font option

Alternatively, you can set the `adjustsFontForContentSizeCategory` property of the label object to `true` so as to make sure the label responds to the text size changes.

```
label1.adjustsFontForContentSizeCategory = true
label2.adjustsFontForContentSizeCategory = true
label3.adjustsFontForContentSizeCategory = true
```

Using Custom Fonts for Text Styles

The default font of any text styles is set to San Francisco, which is the system font of iOS. What if you need to use a custom font? How can you use your own font and take advantage of Dynamic Type at the same time?

Since iOS 11 (or up), developers can scale any custom font to work with Dynamic Type by using a new class called `UIFontMetrics`. Before I explain what `UIFontMetrics` is and how you use it, let's think about what you need to define when using a custom font (say, Mohave) for Dynamic Type.

Apparently, you have to specify the font size for each of the text styles. Say, the `.body` text style should have the font size of 15 points, the `.headline` text style is 18 points, etc.

Remember this is just for the default content size. You will have to provide the font size of these text styles for each of the content size categories. Do you know how many content size categories iOS provides?

Go back to figure 16.7 and do the counting.

If you count it correctly, there are a total of 12 content size categories. Combining with 11 text styles, you will need to set a total of 132 different font sizes (12 content size categories x 11 text styles) in order to support Dynamic Type. That's tedious!

This is where `UIFontMetrics` comes into play to save you time from defining all these font metrics. Instead of specifying the font metrics by yourself, this new class lets you retrieve the font metrics of a specific text style. You can then reuse those metrics to scale the custom font. Here is a sample usage of scaling a custom font for the text style `.title1`:

```
if let customFont = UIFont(name: "Mohave-Italic", size: 28.0) {
    let fontMetrics = UIFontMetrics(forTextStyle: .title1)
    label.font = fontMetrics.scaledFont(for: customFont)
}
```

You can now modify the `viewDidLoad()` method to the following code snippet:

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

    if let customFont1 = UIFont(name: "Mohave-Italic", size: 28.0) {
        let fontMetrics = UIFontMetrics(forTextStyle: .title1)
        label1.font = fontMetrics.scaledFont(for: customFont1)
    }

    if let customFont2 = UIFont(name: "Hallo sans", size: 20.0) {
        let fontMetrics = UIFontMetrics(forTextStyle: .headline)
        label2.font = fontMetrics.scaledFont(for: customFont2)
    }

    if let customFont3 = UIFont(name: "CanterLight", size: 17.0) {
        let fontMetrics = UIFontMetrics(forTextStyle: .subheadline)
        label3.font = fontMetrics.scaledFont(for: customFont3)
    }

    label1.adjustsFontForContentSizeCategory = true
    label2.adjustsFontForContentSizeCategory = true
    label3.adjustsFontForContentSizeCategory = true

}
```

This is how you use custom fonts and make it work with Dynamic Type. Run the app to have a quick test. Also, remember to adjust the preferred font size in Settings to see the text size changes.

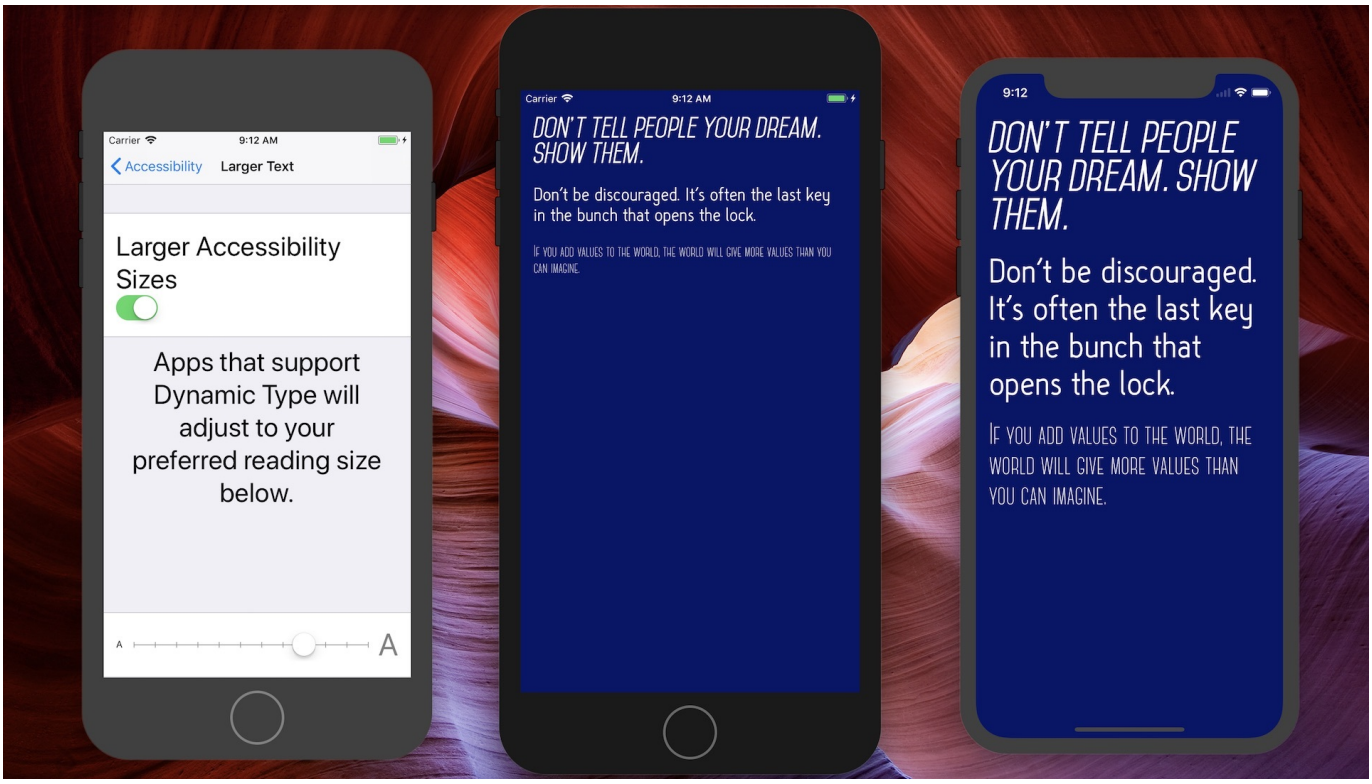


Figure 16.11. Using custom fonts with Dynamic Type

In the code above, we initialized the custom font object with a default font size. It is up to you to decide the font size at the default content size. However, you can always refer to Apple's default typeface for reference.

For the San Francisco typeface, Apple published its font metric used for different content size categories in the iOS Human Interface Guidelines. Figure 16.12 shows the font size of all the text styles at the *Large* content size.

Large (Default)

Style	Weight	Point size	Leading	Tracking
Large Title	Regular	34pt	41pt	11pt
Title 1	Regular	28pt	34pt	13pt
Title 2	Regular	22pt	28pt	16pt
Title 3	Regular	20pt	25pt	19pt
Headline	Semi-Bold	17pt	22pt	-24pt
Body	Regular	17pt	22pt	-24pt
Callout	Regular	16pt	21pt	-20pt
Subhead	Regular	15pt	20pt	-16pt
Footnote	Regular	13pt	18pt	-6pt
Caption 1	Regular	12pt	16pt	0pt
Caption 2	Regular	11pt	13pt	6pt

Figure 16.12. The font metrics Apple defines for its San Francisco typeface at the Large content size

Summary

Apple has put a lot of efforts to encourage developers to use Dynamic Types. With the introduction of `UIFontMetrics`, you can now easily scale custom fonts and make them work with Dynamic Type. When developing your apps, remember that it will reach a lot of users. Some users may prefer a small text size, some may prefer a large text size for comfortable reading. If your apps haven't utilized Dynamic Type, it is time to add it to your To-Do list.

For reference, you can download the complete project from http://www.appcoda.com/resources/swift5/CustomFont.zip.

Chapter 17
Working with AirDrop, UIActivityViewController and Uniform Type Identifiers

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.

Figure 17.1. Using AirDrop on iPhone

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 that 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 added the support of AirDrop sharing. In iOS 8 or later, the activity view controller further added 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 the screen. Here is the code snippet:

```
let objectsToShare = [fileURL]
let activityController = UIActivityViewController(activityItems: objectsToShare, a
pplicationActivities: nil)
present(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 = [UIActivity.ActivityType.postToWeibo, UIActivity.Activity
Type.message, UIActivity.ActivityType.postToTencentWeibo]
activityController.excludedActivityTypes = excludedActivities
```

You can use `UIActivityViewController` to share different types of data including `String`, `UIImage`, and `URL`. Not only you can use `URL` 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 http://www.appcoda.com/resources/swift5/AirdropDemoStarter.zip. After downloading the template, open it and have a quick look.

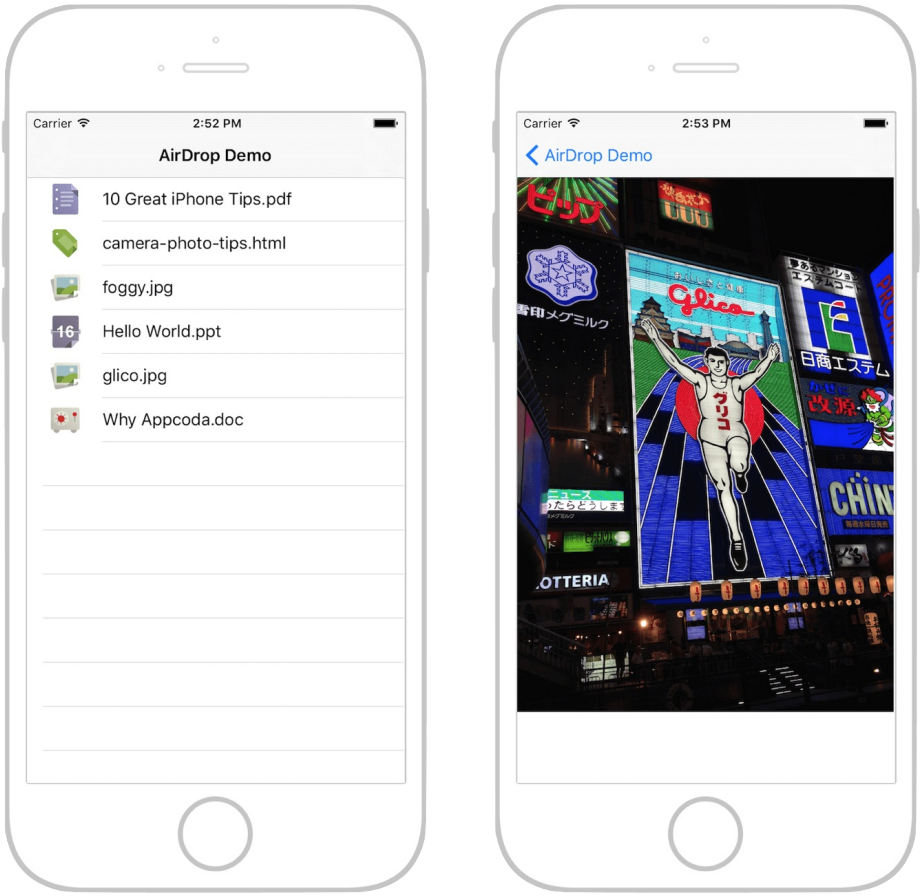


Figure 17.2. AirDrop demo app

The project template already includes the storyboard and the custom classes. The table view controller is associated with `AirDropTableViewController`, while the detail view controller is connected with `DetailViewController`. The `DetailViewController` object simply makes use of `WKWebView` to display the file content. What we are going to do is add a *Share* button in the detail view to activate AirDrop.

> **Note:** If you forgot how to use WKWebView, please refer to chapter 23 of the beginner book.

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:

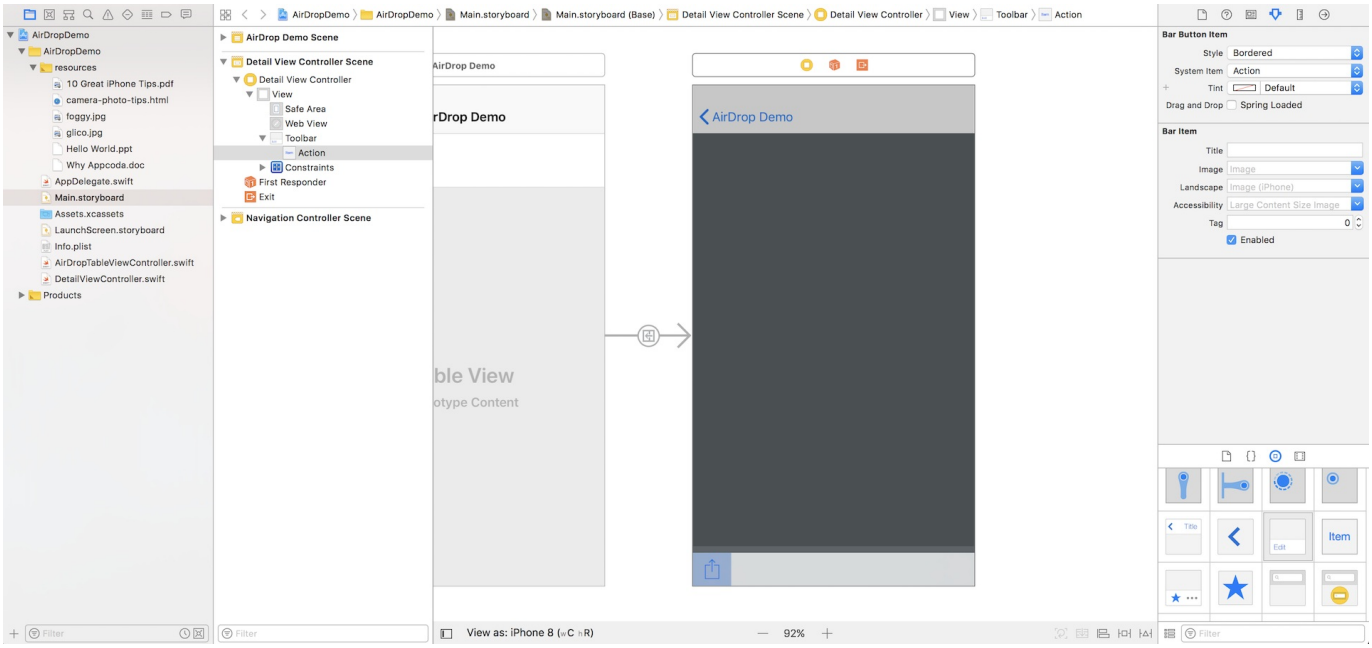


Figure 17.3. Adding a toolbar to the detail view controller

Next, you'll need to add some layout constraints for the toolbar, otherwise, it will not be properly displayed on some devices. Now, select the toolbar. In the auto layout bar, click the *Add new constraints* button to add some spacing constraints. Set the spacing value to `0` for the left, right and bottom sides. Click *Add 3 Constraints* to add the space constraints.

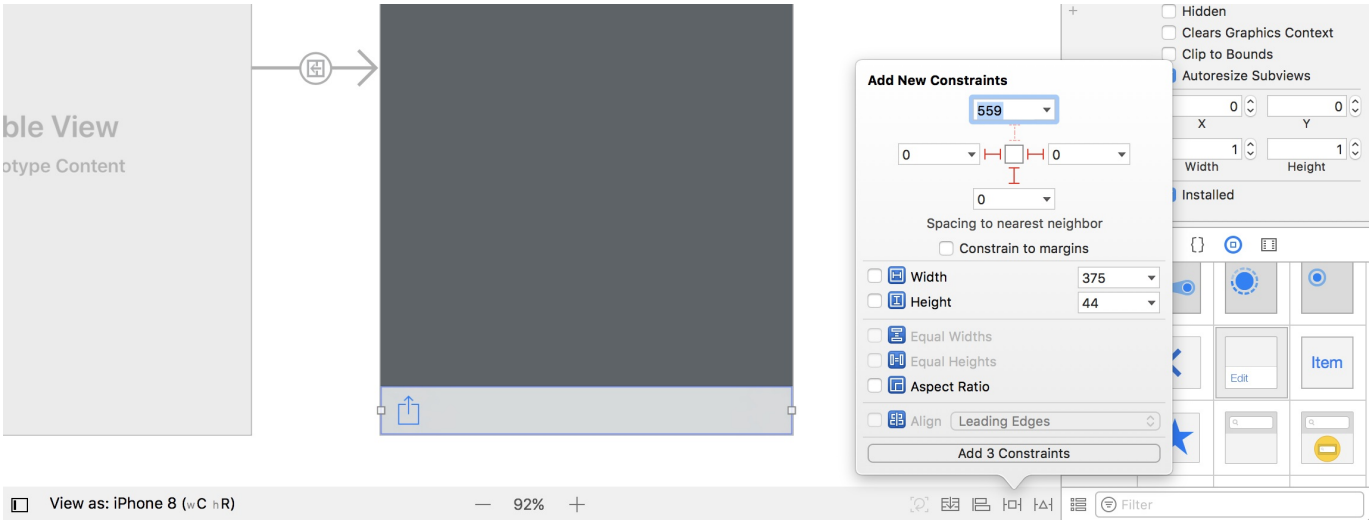


Figure 17.4. Adding some spacing constraints for the toolbar

The newly-added constraints ensure the toolbar is always displayed at the bottom part of the view controller. Now go back to `DetailViewController.swift` 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 `shareWithSender:` from the pop-up menu.

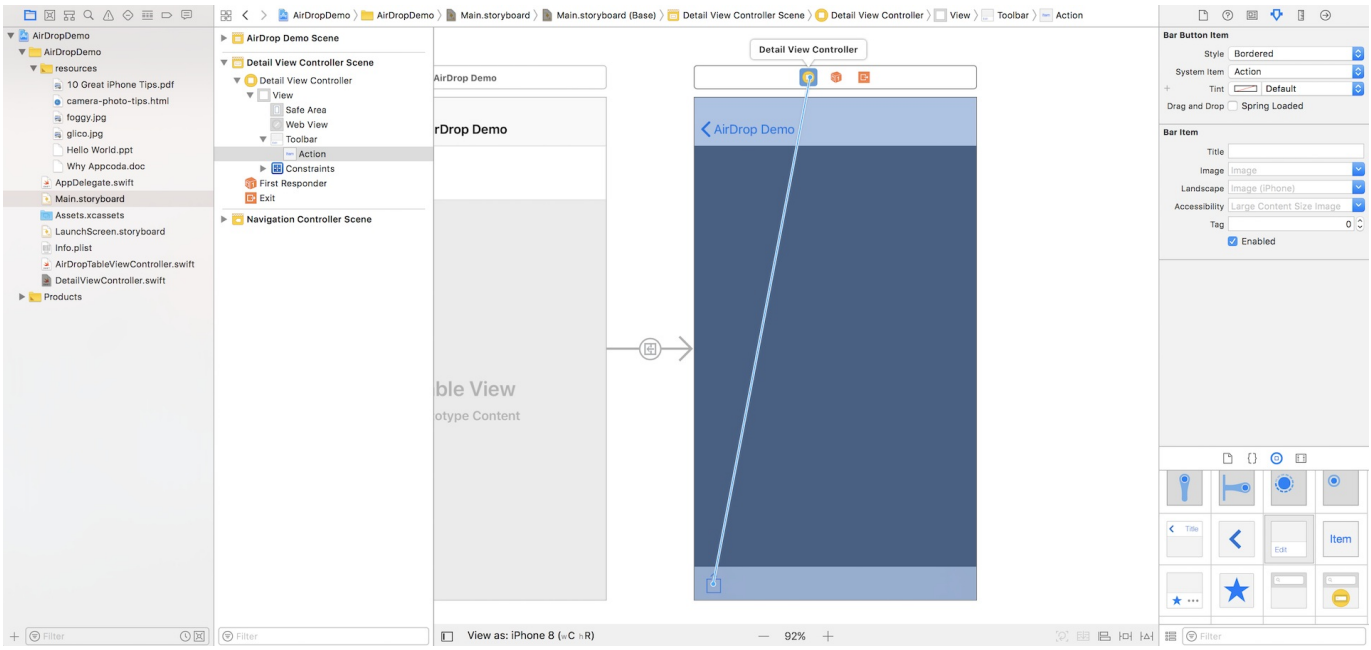


Figure 17.5. Connecting the toolbar item with the action method

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 `DetailViewController` class to the following:

```
@IBAction func share(sender: AnyObject) {
    if let fileURL = fileToURL(file: filename) {
        let objectsToShare = [fileURL]
        let activityController = UIActivityViewController(activityItems: objectsTo
Share, applicationActivities: nil)
        let excludedActivities = [UIActivity.ActivityType.postToFlickr, UIActivity.
ActivityType.postToWeibo, UIActivity.ActivityType.message, UIActivity.ActivityType
.mail, UIActivity.ActivityType.print, UIActivity.ActivityType.copyToPasteboard, UI
Activity.ActivityType.assignToContact, UIActivity.ActivityType.saveToCameraRoll, U
IActivity.ActivityType.addToReadingList, UIActivity.ActivityType.postToFlickr, UIA
ctivity.ActivityType.postToVimeo, UIActivity.ActivityType.postToTencentWeibo]

        activityController.excludedActivityTypes = excludedActivities

        if let popOverController = activityController.popoverPresentationControlle
r {
            popOverController.barButtonItem = actionButtonItem
        }

        present(activityController, animated: true, completion: nil)
    }
}
```

The code above 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 (e.g. print / assign to contact) and presents the controller on the screen. The tricky part is how you define the objects to share.

The `filename` property of `DetailViewController` 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) -> URL? {
    // Get the full path of the file
    let fileComponents = file.components(separatedBy: ".")

    if let filePath = Bundle.main.path(forResource: fileComponents[0], ofType: fil
eComponents[1]) {
        return URL(fileURLWithPath: filePath)
    }

    return nil
}
```

The code is very straightforward. For example, the image file `glico.jpg` will be transformed to:

```
file:///Users/simon/Library/Developer/CoreSimulator/Devices/7DC35502-54FD-447B-B10
F-2B7B0FD5BDDF/data/Containers/Bundle/Application/01827504-4247-4C81-9DE5-02BEAE94
C7E5/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 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.

Figure 17.6. Testing the AirDrop demo on a real device

Working with iPad

If you run the demo app on iPad and try to activate the Share action, you will end up with the following error:

```
2019-11-14 12:26:32.284152+0800 AirDropDemo[28474:5821534] *** Terminating app due
to uncaught exception 'NSGenericException', reason: 'Your application has presente
d a UIActivityViewController (<UIActivityViewController: 0x7f9d4985d400>). In its
current trait environment, the modalPresentationStyle of a UIActivityViewControlle
r with this style is UIModalPresentationPopover. You must provide location informa
tion for this popover through the view controller's popoverPresentationController.
 You must provide either a sourceView and sourceRect or a barButtonItem.  If this
information is not known when you present the view controller, you may provide it
in the UIPopoverPresentationControllerDelegate method -prepareForPopoverPresentati
on.'
```

On iPad, `UIActivityViewController` is presented as a *popover* instead of an action sheet. In this case, you have to specify either the source view or the source bar button item for the popover presentation controller. For our demo app, when the user taps the Share button, we want to present the popover from the button. To do that, create an outlet variable for the Share button in `DetailViewController.swift` and establish the connection in the Interface Builder:

```
@IBOutlet var actionButtonItem: UIBarButtonItem!
```

Next, go back to `DetailViewController.swift`. Insert the following code snippet right before calling `present(activityController, animated: true, completion: nil)` of the `share(sender:)` method:

```
if let popOverController = activityController.popoverPresentationController {
    popOverController.barButtonItem = actionButtonItem
}
```

When a device (e.g. iPad) uses *popover* for presenting `UIActivityViewController`, the `popoverPresentationController` property has a value. Therefore, we test if the property contains a value, and then set its `barButtonItem` property to the Share button.

Now if you run the demo app on iPad, you will have something like this:

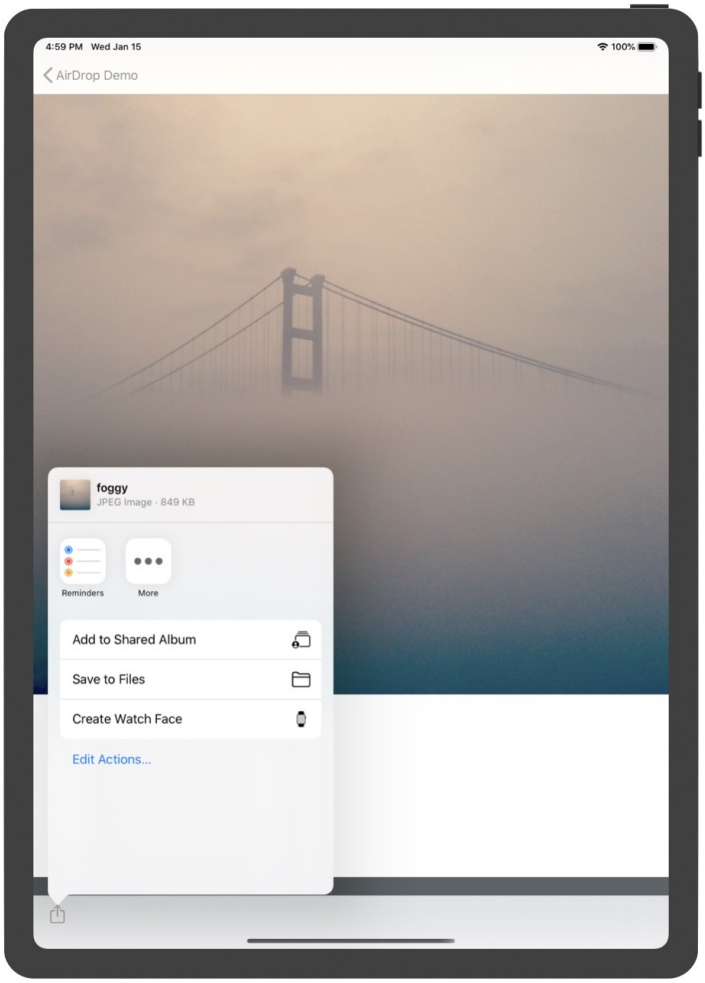


Figure 17.7. Share action on iPad

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:

https://developer.apple.com/library/content/documentation/Miscellaneous/Reference/UTIRef/Articles/System-DeclaredUniformTypeIdentifiers.html#//apple_ref/doc/uid/TP40009259-SW1

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.

Figure 17.8. List of supported applications

The system allows multiple apps to register the same UTI. In this case, iOS will prompt the 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.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/AirDropDemo.zip.

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 icons 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.

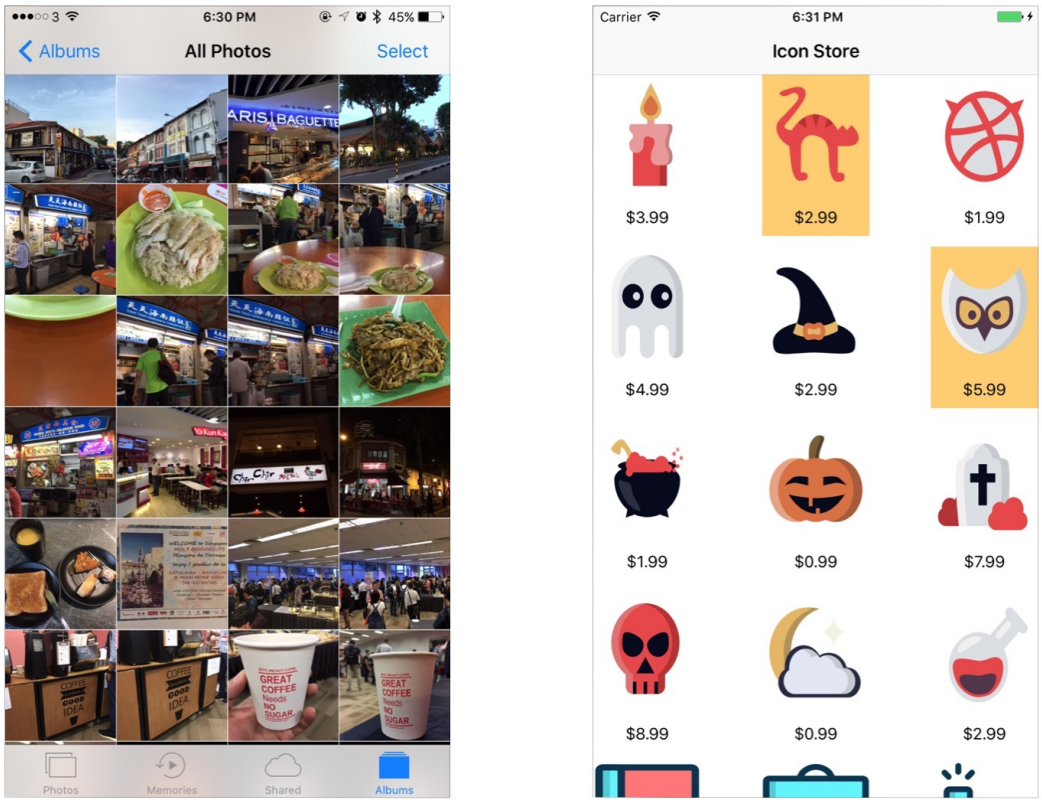


Figure 18.1. A sample usage of Collection Views (left: Photos app, right: our demo app)

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. What we're going to do is to build an icon store app. When a user launches the app, it displays a set of icons (with price included) in grid form.

Creating a New Project

First, fire up Xcode and create a new project using the *Single View Application* template. Name the project `CollectionViewDemo` 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.

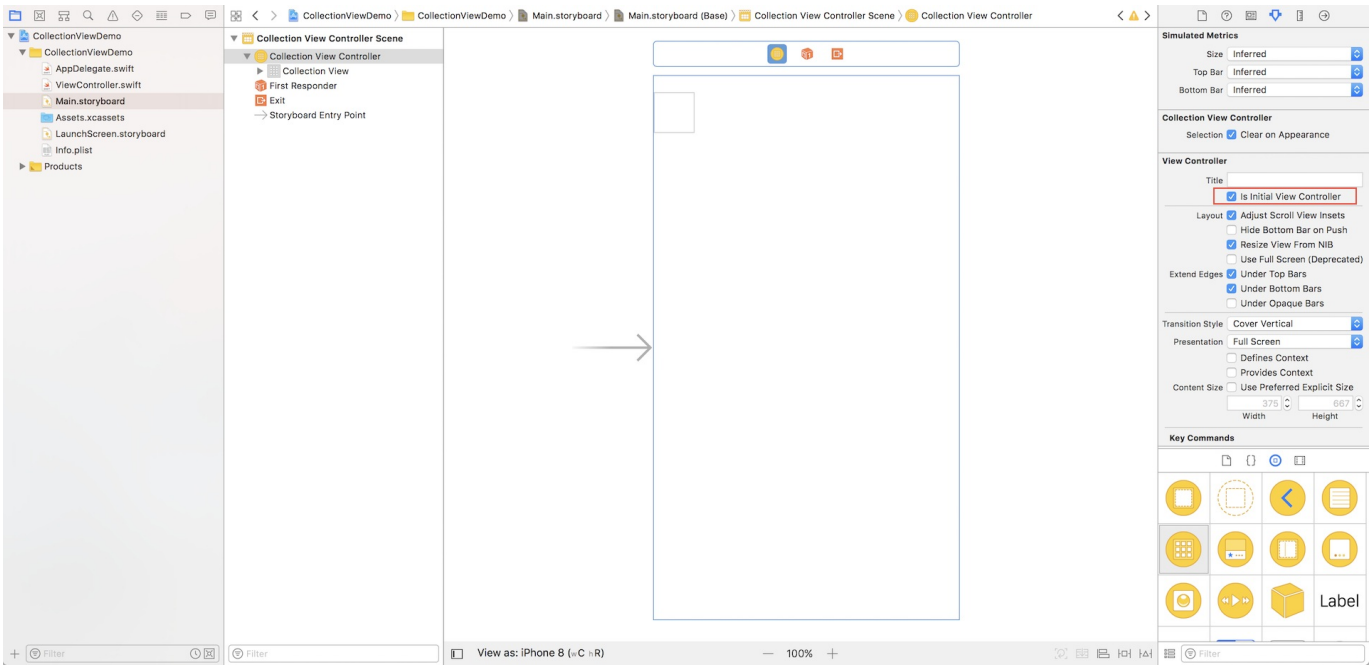


Figure 18.2. Adding a collection view controller in the storyboard

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 and `150` points respectively. Also, change the min spacing of both *for cells* and *for lines* to `10` points.

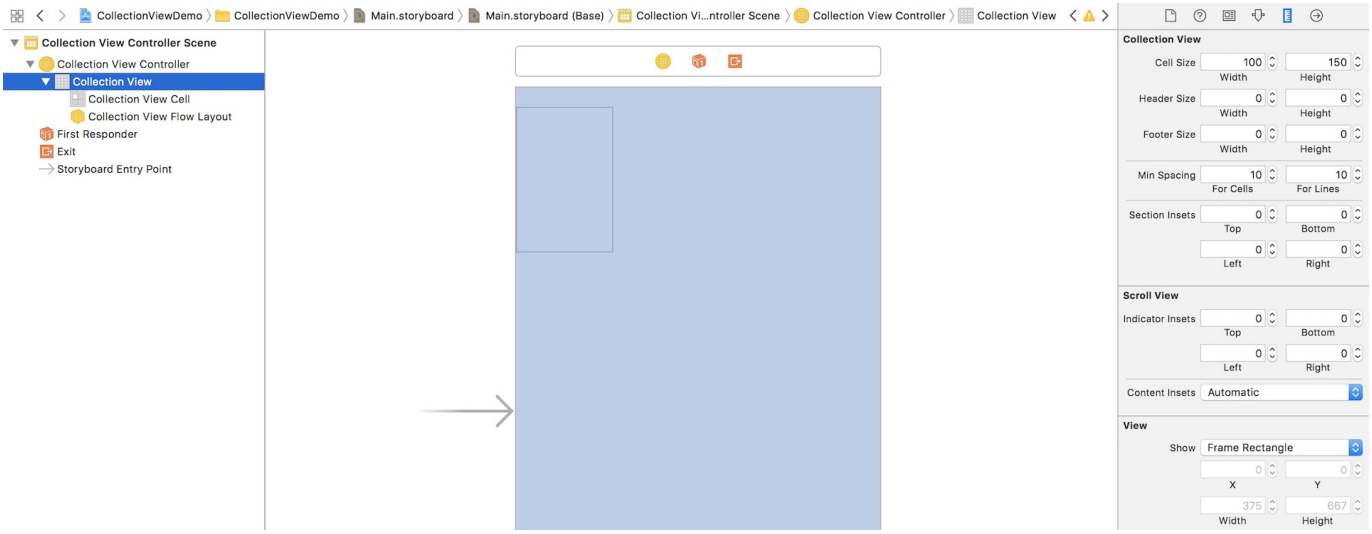


Figure 18.3. Changing the size of the collection view cell

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?

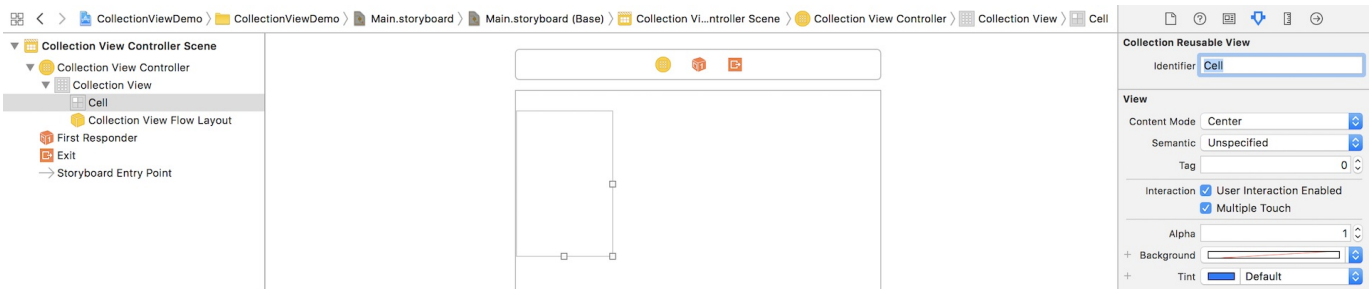


Figure 18.4. Setting the cell's identifier

Now drag an image view from the Object library to the cell. You then manually resizes the image view such that its width is `100` points and its height is `115` points. Alternatively, you can go to the Size inspector and set its size (see figure below).

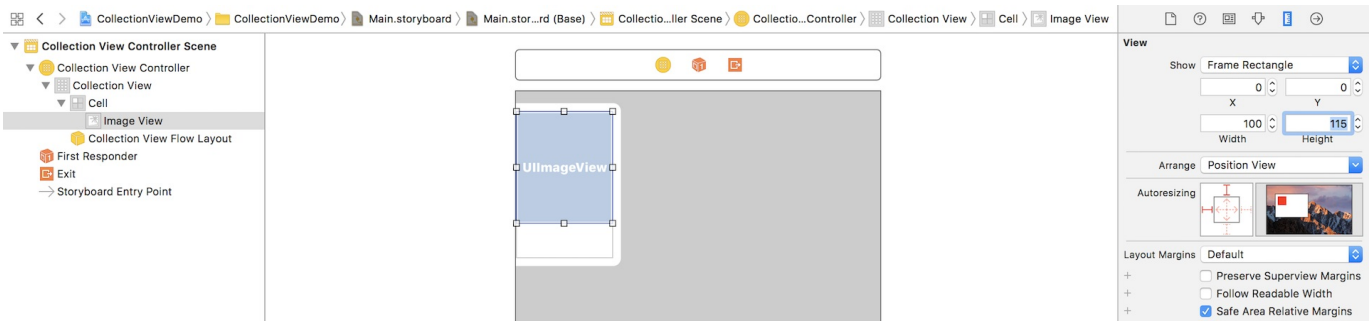


Figure 18.5. Adjusting the size of the image view

To display the price of an icon, we will add a label below the image view. Drag a label object from the Object library to the collection view cell. In the Size inspector, set X to `0`, Y to `115`, Width to `100`, and Height to `35`. In the Attributes inspector, change the alignment option to `center`, and the font size to `15` points. Your cell design should look similar to that in figure 18.6.

Figure 18.6. Adding a label to the collection view cell

As usual, you will need to add some layout constraints for the image view and the label. Select the image view, click the *Add New Constrants* button in the layout bar and add 4 spacing contraints (refer to the figure below for details).

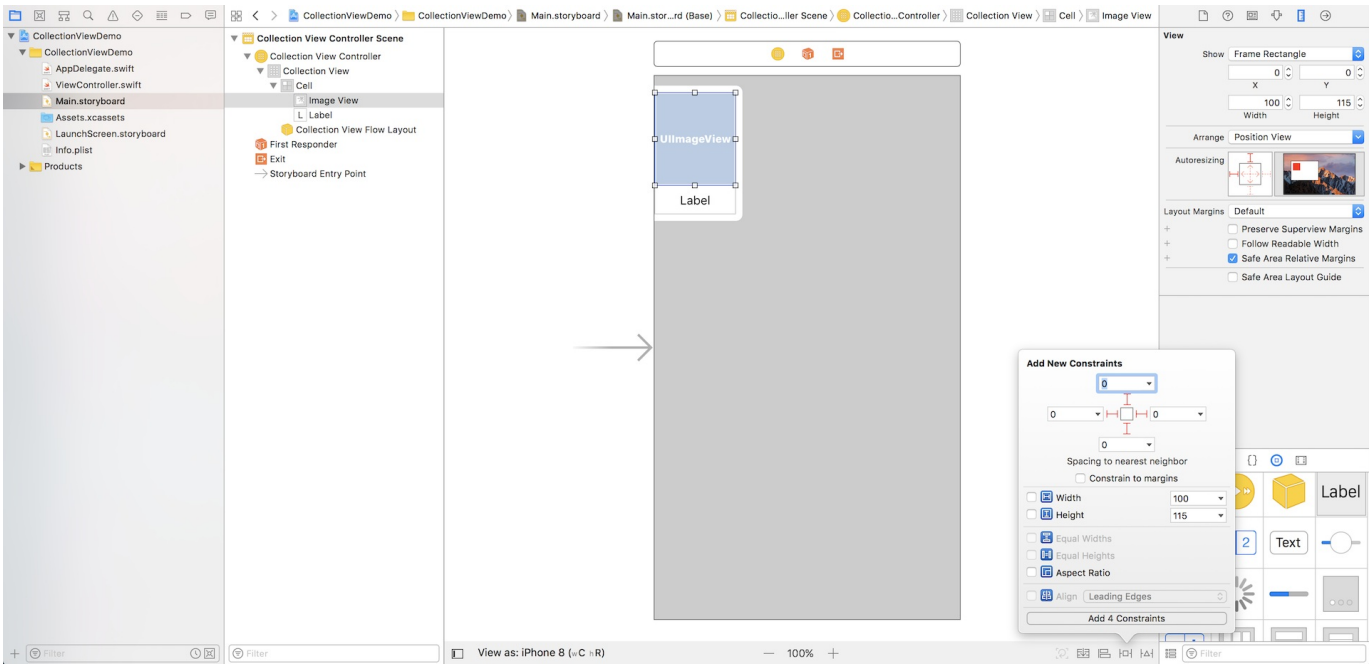


Figure 18.7. Adding spacing constraints for the image view

Next, select the label, and click the *Add New Constraints* button. Set the spacing value of the left, right and bottom sides to `0`. Also, check the *height* checkbox to limit the height of the label. Click `Add 4 Constraints` to add the constraints.

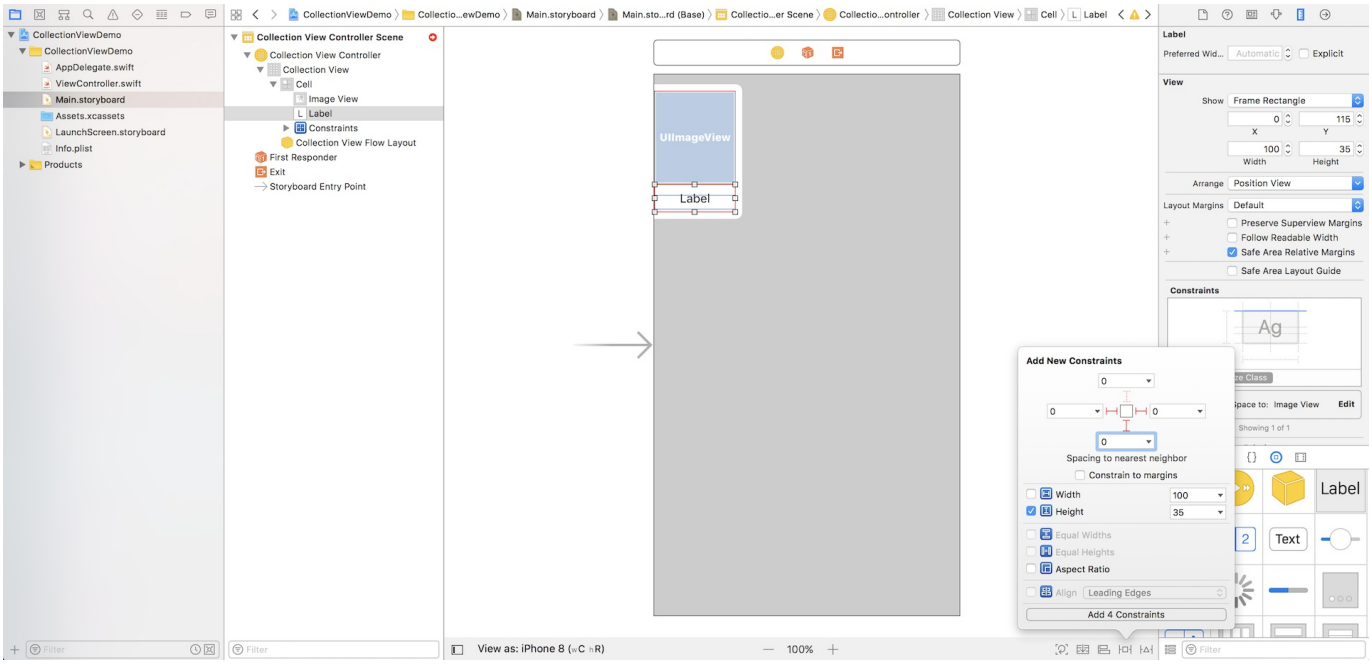


Figure 18.8. Adding spacing constraints for the image view

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 `Icon Store`.

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

First, 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.

Similar to how you implement the table view cell, we have to create a custom class for a custom collection view cell. Right click the *CollectionViewDemo* folder and select `New File...`. Create a new class using the Cocoa Touch Class template. Name the class `IconCollectionViewCell` and set the subclass to `UICollectionViewCell`.

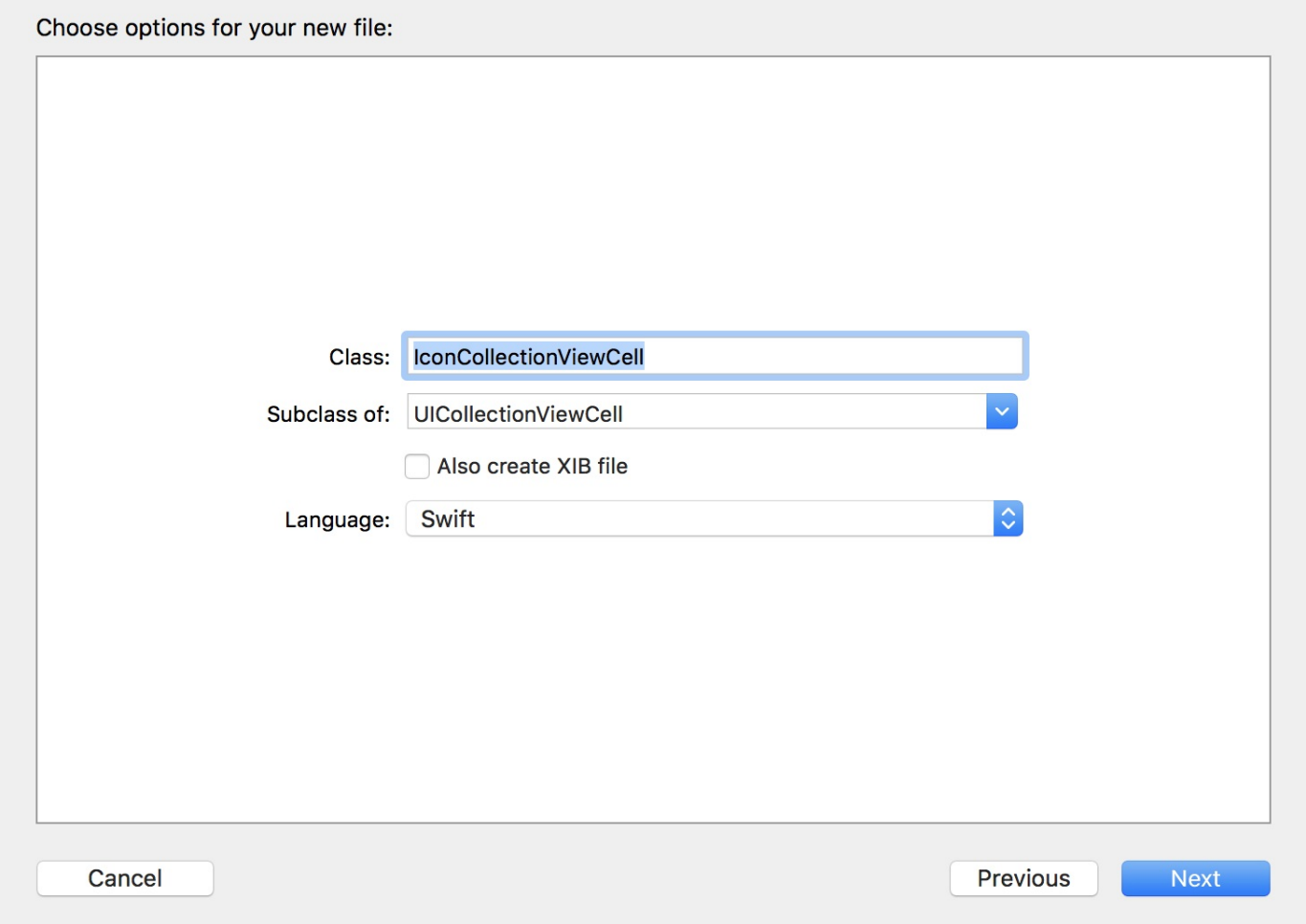


Figure 18.9. Creating a new class named IconCollectionViewCell

Repeat the process to create another class for the collection view controller. Name the new class `IconCollectionViewController` and set the subclass to `UICollectionViewController`.

Let's start with `IconCollectionViewCell.swift`. The cell has an image view and a label. So, we will create two outlet variables in the class.

Now open the `IconCollectionViewCell.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 IconCollectionViewCell: UICollectionViewCell {
    @IBOutlet weak var iconImageView: UIImageView!
    @IBOutlet weak var iconPriceLabel: UILabel!
}
```

Go back to storyboard and select the collection view cell. Under the Identity inspector, change the custom class to `IconCollectionViewCell`. Then right click the cell and connect the outlet variable with the image view.

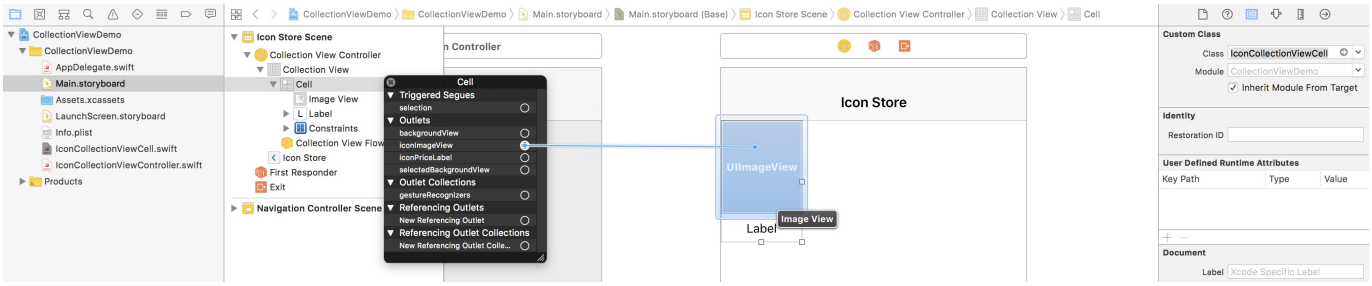


Figure 18.10. Connecting the image view with the corresponding outlet variable

Next, select the collection view controller. Under the Identity inspector, set the custom class to `IconCollectionViewController`.

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, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell`

First, download this image pack (http://www.appcoda.com/resources/swift4/IconStoreImage.zip), unzip it and add all the images to the image asset.

> **Note:** The icon images are courtesy of Tania Raskalova and Marin Begović.

Next, create a new Swift file and name it `Icon.swift`. In the file, we define an `Icon` structure with three properties:

- **name** - the image name of the icon
- **price** - the price of the icon
- **isFeatured** - indicates if the icon is featured on the store

Your `Icon.swift` file should look like this:

```
import Foundation

struct Icon {
    var name: String = ""
    var price: Double = 0.0
    var isFeatured: Bool = false

    init(name: String, price: Double, isFeatured: Bool) {
        self.name = name
        self.price = price
        self.isFeatured = isFeatured
    }
}
```

Now let's move on to the implementation of the `IconCollectionViewController` class. In the class, declare an `iconSet` array and initialize it with the set of icon images. For demo purpose in the later section, we have set the `isFeatured` property of some `Icon` objects to `true`.

```
private var iconSet: [Icon] = [ Icon(name: "candle", price: 3.99, isFeatured: false
),
                                Icon(name: "cat", price: 2.99, isFeatured: true),
                                Icon(name: "dribbble", price: 1.99, isFeatured: fa
lse),
                                Icon(name: "ghost", price: 4.99, isFeatured: false
),
                                Icon(name: "hat", price: 2.99, isFeatured: false),
                                Icon(name: "owl", price: 5.99, isFeatured: true),
                                Icon(name: "pot", price: 1.99, isFeatured: false),
                                Icon(name: "pumkin", price: 0.99, isFeatured: false
),
                                Icon(name: "rip", price: 7.99, isFeatured: false),
                                Icon(name: "skull", price: 8.99, isFeatured: false
),
                                Icon(name: "sky", price: 0.99, isFeatured: false),
                                Icon(name: "toxic", price: 2.99, isFeatured: false
),
                                Icon(name: "ic_book", price: 2.99, isFeatured: fal
se),
                                Icon(name: "ic_backpack", price: 3.99, isFeatured:
false),
                                Icon(name: "ic_camera", price: 4.99, isFeatured: f
alse),
                                Icon(name: "ic_coffee", price: 3.99, isFeatured: t
rue),
                                Icon(name: "ic_glasses", price: 3.99, isFeatured:
false),
                                Icon(name: "ic_ice_cream", price: 4.99, isFeatured
: false),
                                Icon(name: "ic_smoking_pipe", price: 6.99, isFeatu
red: false),
                                Icon(name: "ic_vespa", price: 9.99, isFeatured: fa
lse)]
```

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!.register(UICollectionViewCell.self, forCellWithReuseIdentifie
r: 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 numberOfSections(in collectionView: UICollectionView) -> Int {
    // Return the number of sections
    return 1
}

override func collectionView(_ collectionView: UICollectionView, numberOfItemsInSe
ction section: Int) -> Int {
    // Return the number of items
    return iconSet.count
}

override func collectionView(_ collectionView: UICollectionView, cellForItemAt ind
exPath: IndexPath) -> UICollectionViewCell {
    let cell = collectionView.dequeueReusableCell(withReuseIdentifier: reuseIdenti
fier, for: indexPath) as! IconCollectionViewCell

    // Configure the cell
    let icon = iconSet[indexPath.row]
    cell.iconImageView.image = UIImage(named: icon.name)
    cell.iconPriceLabel.text = "$\(icon.price)"

    return cell
}
```

The `numberOfSections(in:)` 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 icon images.

The `collectionView(_:cellForItemAt:)` 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 `dequeueReusableCell(withReuseIdentifier:for:)`

method will either automatically create a cell or return a cell from the re-use queue. The cell returned is downcast to the type `IconCollectionViewCell`. Finally, we get the corresponding image and assign it to the image view to display.

Now compile and run the app using the iPhone SE simulator. You should have a grid-based Icon Store app like this.

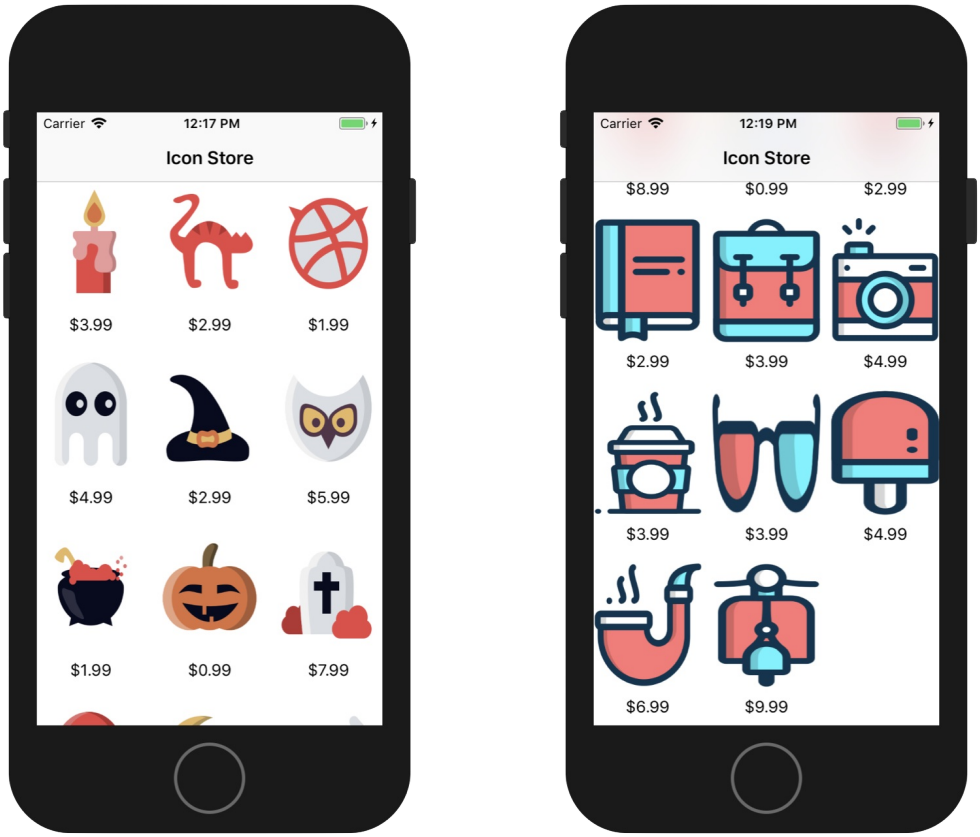


Figure 18.11. Displaying the icons in grid form

> **Quick note:** If you run the app on iPhone 8/8 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 app. What if you want to highlight some of the featured icons? 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 icon image. What we are going to do is use the background view to display a background image. In the image pack you downloaded earlier, it includes a file named *feature-bg.png*, which is the background image.

For those `Icon` objects that are featured, the `isFeatured` property is set to `true`. I want to highlight these icons with a bright color background.

Let's see how to do it.

Go back to the `IconCollectionViewController.swift` file. In the `collectionView(_:cellForItemAt:)` method, insert the following line of code before `return cell`:

```
cell.backgroundView = (icon.isFeatured) ? UIImageView(image: UIImage(named: "featu
re-bg")) : nil
```

We simply load the background image and set it as the background view of the collection view cell when the icon is featured. Now compile and run the app again. Your app now displays a background image for those featured icons.

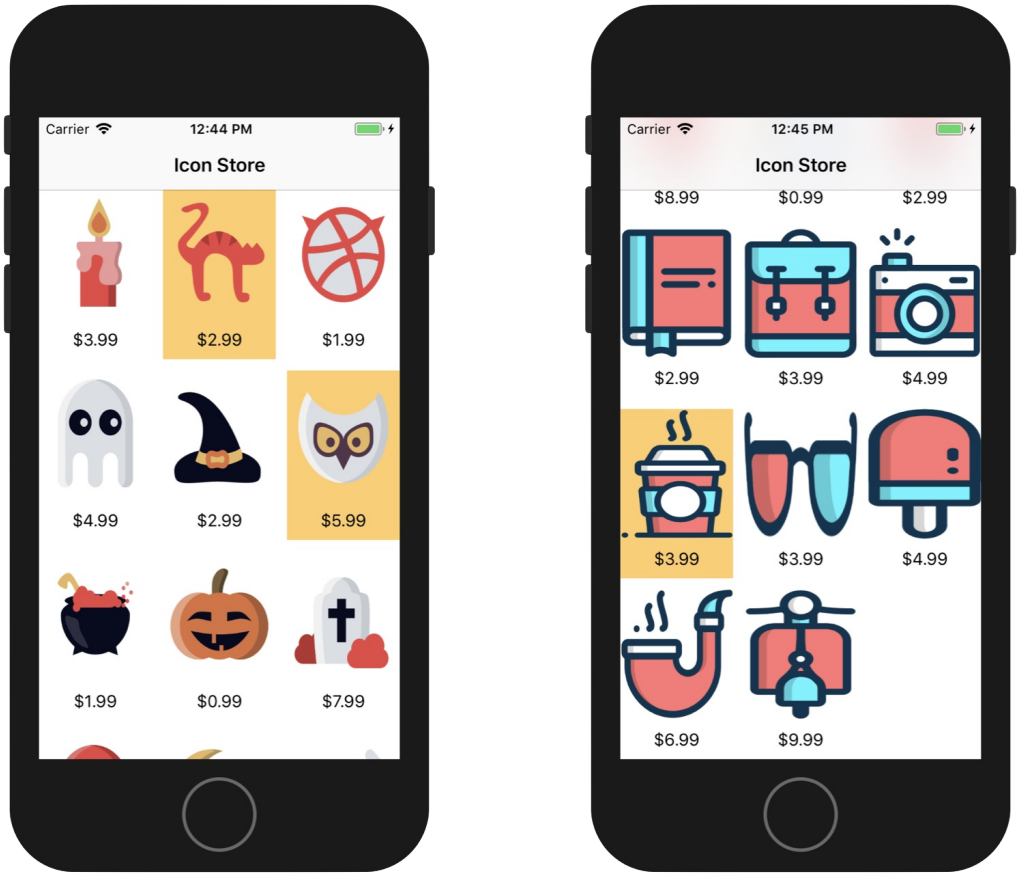


Figure 18.12. Highlighting those featured icons with a background image

For reference, you can download the Xcode project from http://www.appcoda.com/resources/swift5/CollectionViewDemo.zip.

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, especially 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 an icon, the app will bring up a modal view to display the icon in a

larger size.
- We'll also implement social sharing in the app in order to demonstrate multiple item selections. Users are allowed to select multiple icons and share them on Messages.

Let's first see how to implement the first feature to handle single selection. When the user taps any of the icons, the app will bring up a modal view to display a larger photo and its information like description and price. If you didn't go through the previous chapter, you can start by downloading the starter project from http://www.appcoda.com/resources/swift5/CollectionViewSelectionStarter.zip.

The starter project is very similar to the final project we built in the previous chapter. There are only a couple of changes for the `Icon` structure. I just changed the `Icon` structure a bit to store the name, image, and description of an icon. You can refer to `Icon.swift` to reveal the changes.

Designing the Detail View Controller

Before I showed you how to interact with the collection view, let's start by designing the detail view controller that is used to display the icon details.

Go to `Main.storyboard`, drag a View Controller from the Object library to the storyboard. Then add an Image View to it. In the Size inspector, set X to `0`, Y to `0`, width to `375` and height to `400`. Under the Attributes inspector, change the mode of the image view to `Aspect Fit`.

Next, drag a button and place it at the bottom of the view controller. Set its width to `375` points and height to `47` points. In the Attributes inspector, set the type to `System` and change its background color to yellow (or whatever color you prefer). The text color should be changed to white to give it some contrast.

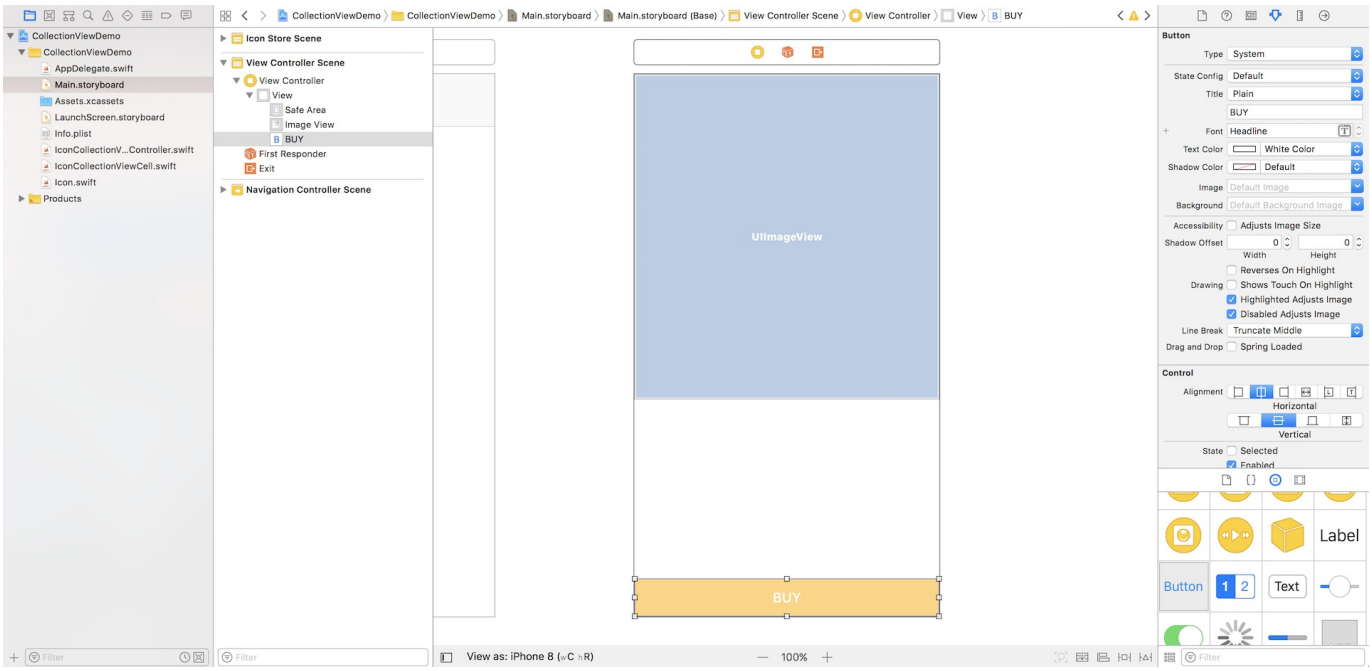


Figure 19.1. Designing the detail view controller

Now, let's add three labels for the name, description, and price of the icon. Place them in the white area of the detail view controller. You can use whatever font you like, but make sure you the alignment of the labels to *center*.

Lastly, drag another button object to the view, and place it near the top-right corner. This is the button for dismissing the view controller. Set its type to `System`, title to *blank*, and image to `close`. You will have to resize the button a little bit. I set its width and height to `30` points.

Your UI design should be very similar to figure 19.2.

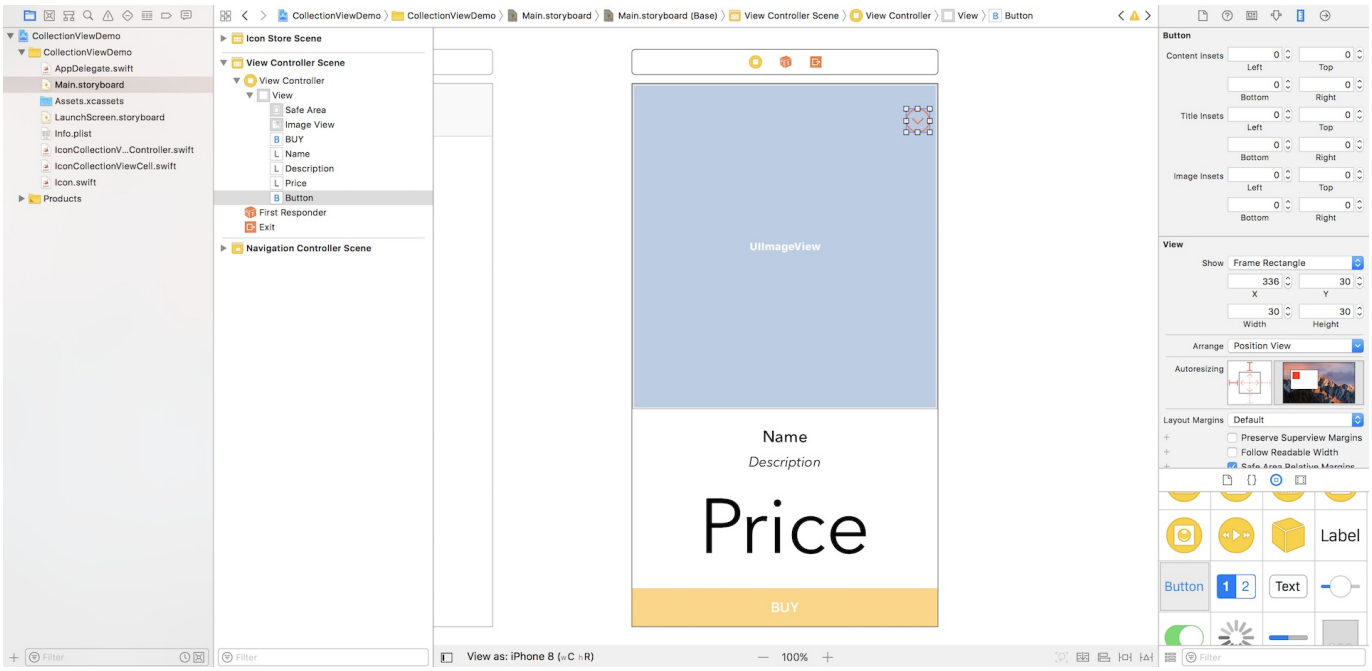


Figure 19.2. Adding three labels in the icon detail view controller

Now that we have completed the layout of the detail view controller, the next step is to add some layout constraints so that the view can fit different screen sizes.

Let's begin with the image view. In Interface Builder, select the image view, and click the `Add new constraints` button to add the spacing and size constraints. Make sure the `Constrain to margins` is unchecked, and set the spacing value of the top, left and right sides to `0`. The image should scale up/down without altering its aspect ratio. So check the *Aspect Ratio* option, and then click `Add 4 Constraints`. Then select the top spacing constraint. In the Size inspector, make sure the *Second Item* is set to `Safe Area.Top`. This ensures that the image will not be covered by the status bar.

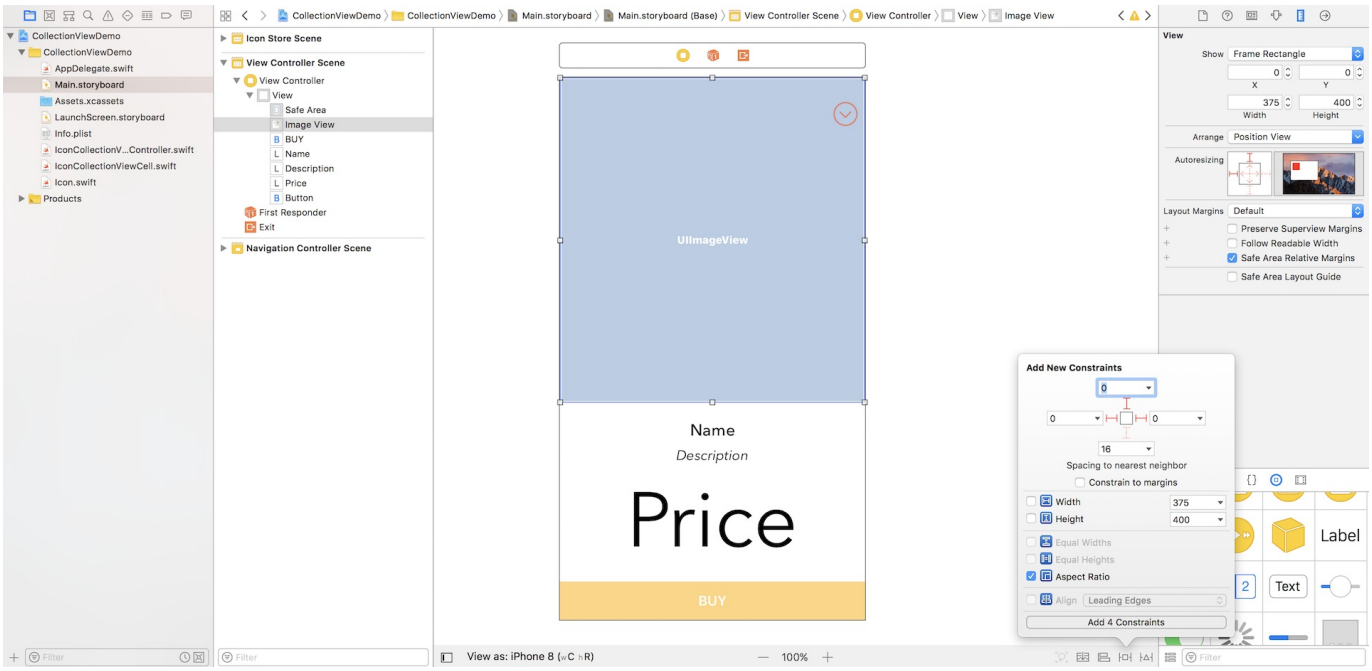


Figure 19.3. Defining layout constraints for the image view

Next, select the *close* button at the top-right corner. Click the `Add new constraints` button. Add the spacing constraints for the top and right sides. Also, check both width and height options so that the image size will keep intact regardless of the screen size.

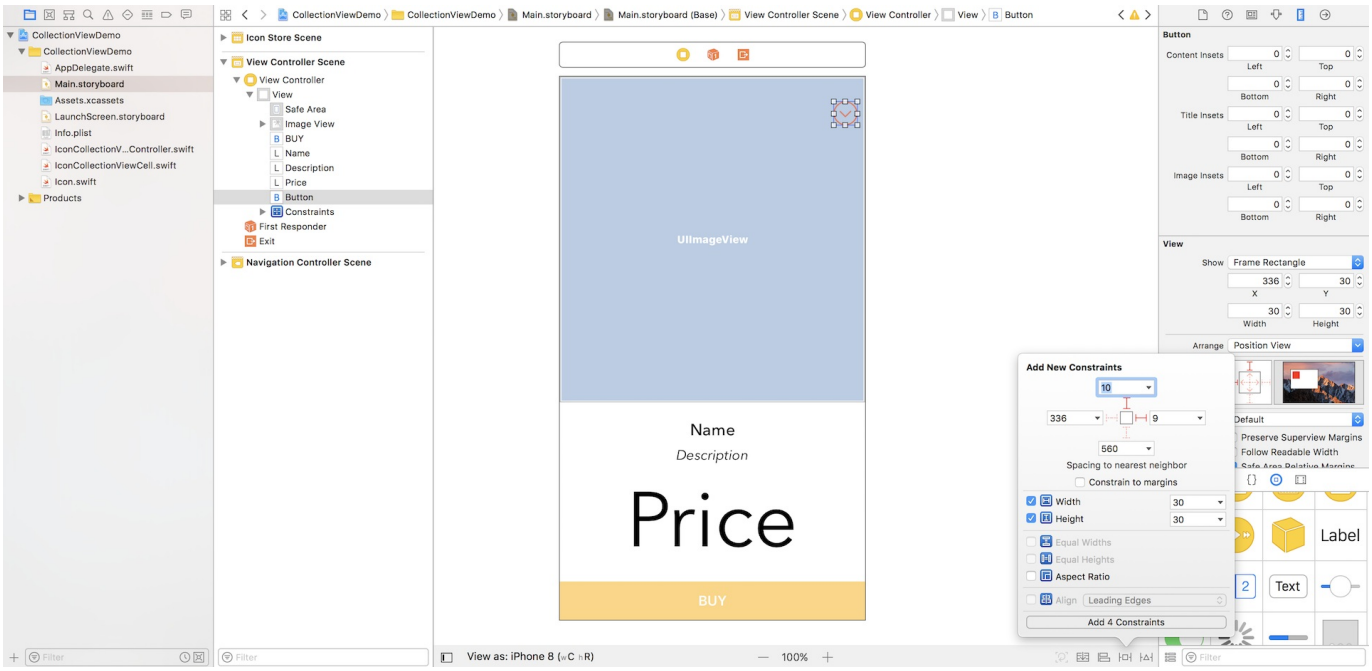


Figure 19.4. Defining layout constraints for the close button

Next, let's move on to the *Buy* button. We want it to stick to the bottom of the view, and keep its height unchanged. So select the button and click the `Add new constraints` button. Set the spacing value of the left, right and bottom sides to `0`. Check the height option to restrict its height, and click the `Add 4 Constraints` button to confirm.

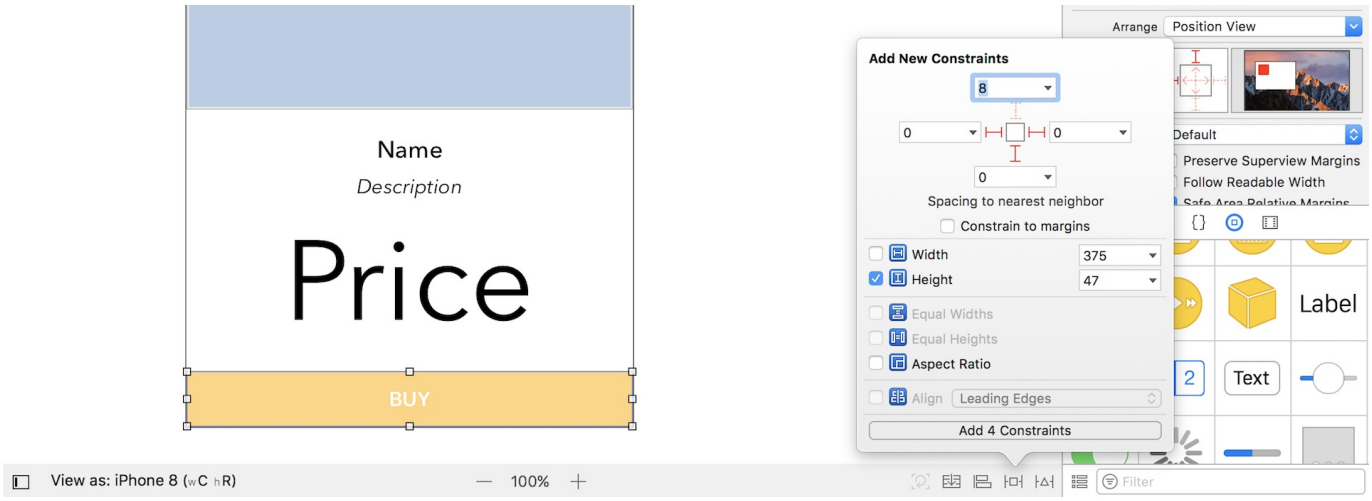


Figure 19.5. Defining layout constraints for the buy button

Now it comes to the labels. I prefer not to define the constraints of these labels one by one. Instead, I will use a stack view to embed them. Press and hold the command key, select the *Name*, *Description*, and *Price* labels. Then click the `Embed in stack` button in the layout bar to group them into a stack view. In the Attributes inspector, change the *Distribution* option of the stack view to `Fill Proportionally`.

Next, select the stack view and click the `Add new constraints` button. Define four spacing constraints for the stack view:

- Top side: 15 points
- Left/right side: 20 points
- Bottom side: 15 points

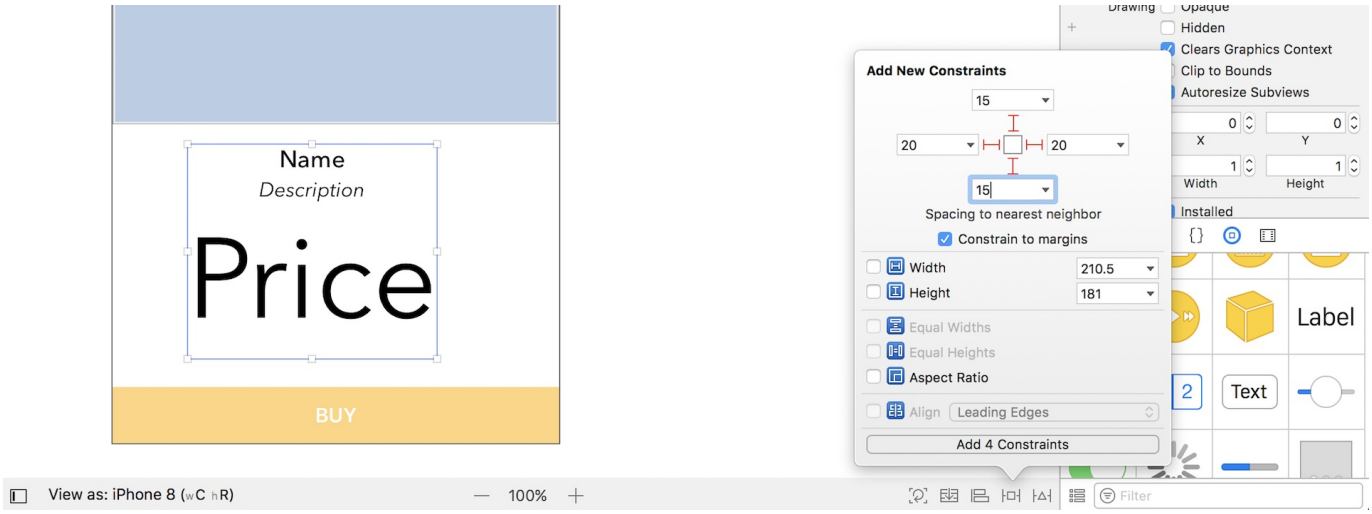


Figure 19.6. Defining layout constraints for the stack view

Once you add the constraints, Interface Builder shows you some layout issues. Select the bottom constraint of the stack view. Go to the Size inspector and change the Relation to `Greater Than or Equal` to fix the issue.

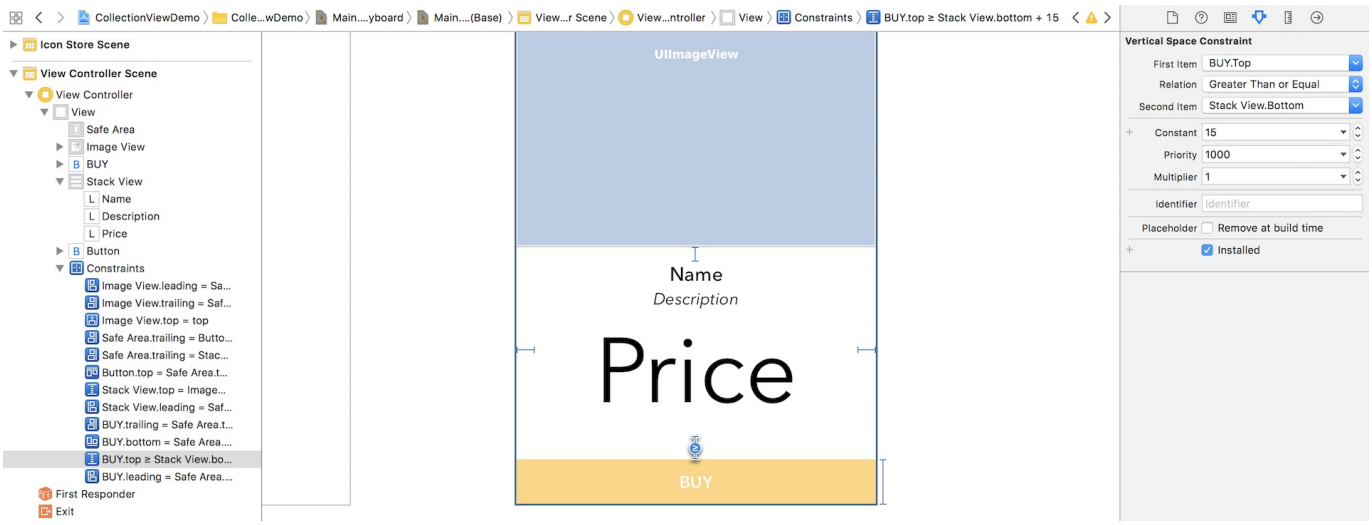


Figure 19.7. Changing the relation of the spacing constraint

Connecting the Controllers Using Segues

Cool! You've finished the UI design of the detail view. Let's see how to bring up the detail view controller when a user selects an icon in the collection view controller.

Since we want to display the view controller when a user taps any of the icons 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 view controller we just added. Select `Present Modally` for the style and set the segue identifier to `showIconDetail`.

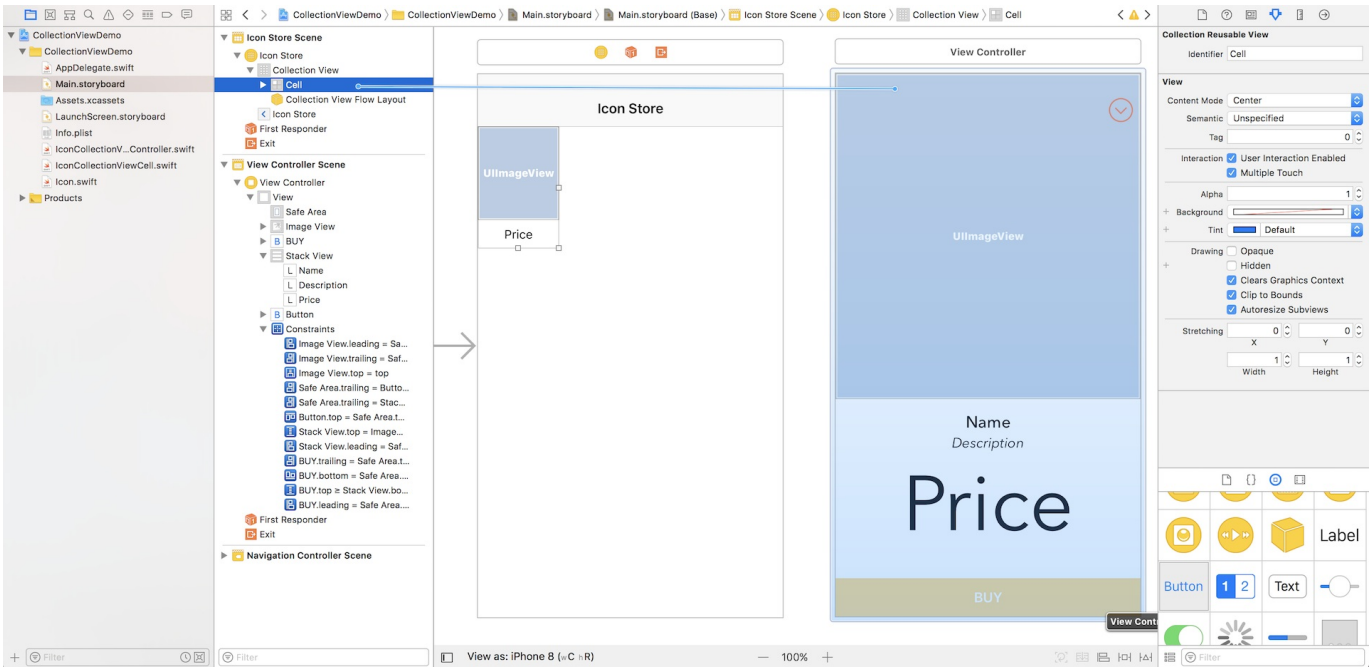


Figure 19.8. Connecting the collection view cell with the detail view controller

When the user taps the *Close* button in the detail view controller, the controller will be dismissed. In order to do that, we will add an unwind segue. In `IconCollectionViewController.swift`, insert the following unwind segue method:

```
@IBAction func unwindToHome(segue: UIStoryboardSegue) {
}
```

Go back to the storyboard. Control-drag from the *Close* button to the Exit icon of the scene dock. Select `unwindToHomeWithSegue:` 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.

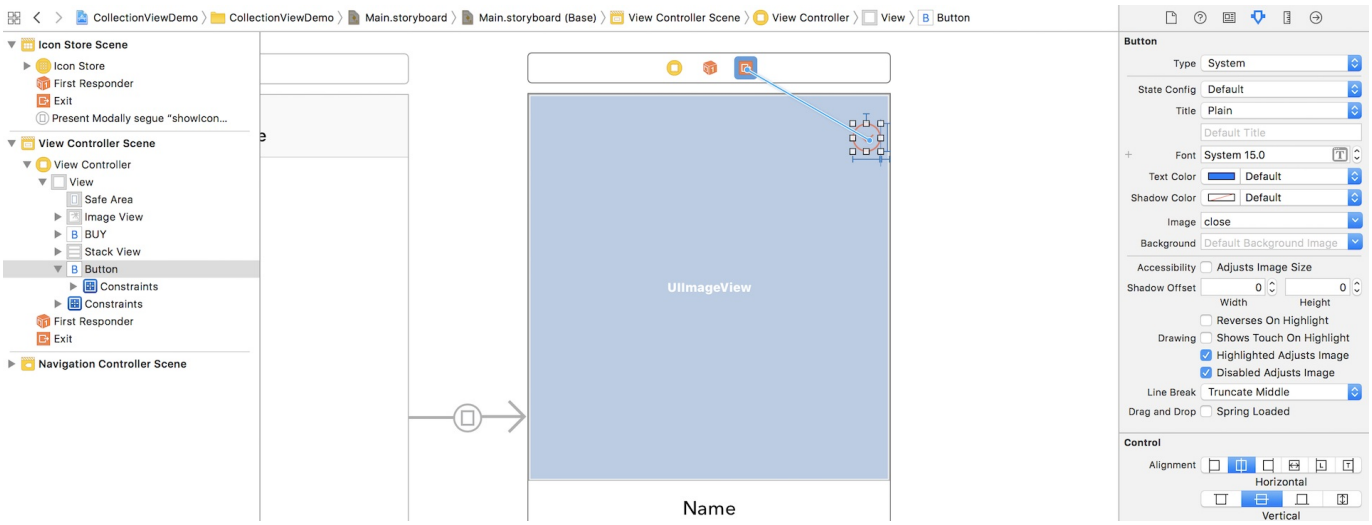


Figure 19.9. Connecting the close button with an unwind segue

If you compile and run the app, you'll end up with an empty view when selecting any of the icons. Tapping the *Close* button will dismiss the view.

Creating a Custom Class for the Detail View Controller

Since we haven't written any code, the modal view controller knows nothing about the selected icon. We will create a custom class for the detail view controller and see how we can pass the selected icon to the detail view.

Create a new class and name it `IconDetailViewController`. Make it a subclass of `UIViewController`. Once created, add the following outlet variable in the `IconDetailViewController` class, and define a variable for the selected icon:

```
@IBOutlet var nameLabel: UILabel! {
    didSet {
        nameLabel.text = icon?.name
    }
}
@IBOutlet var descriptionLabel: UILabel! {
    didSet {
        descriptionLabel.text = icon?.description
        descriptionLabel.numberOfLines = 0
    }
}
@IBOutlet var iconImageView: UIImageView! {
    didSet {
        iconImageView.image = UIImage(named: icon?.imageName ?? "")
    }
}
@IBOutlet var priceLabel: UILabel! {
    didSet {
        if let icon = icon {
            priceLabel.text = "$\(icon.price)"
        }
    }
}

var icon: Icon?
```

In the above code, we use `didSet` to initialize the title of the labels and the image of the image view. You can do the same kind of initialization in the `viewDidLoad` method. But I prefer to use `didSet` as the code is more organised and readable.

Now go back to `Main.storyboard`. Select the detail view controller and set the custom class to `IconDetailViewController` in the Identity inspector. Then establish the connections for the outlet variables:

- `nameLabel`: Connect it to the *Name* label
- `descriptionLabel`: Connect it to the *Description* label
- `iconImageView`: Connect it to the image view
- `priceLabel`: Connect it to the *Price* label

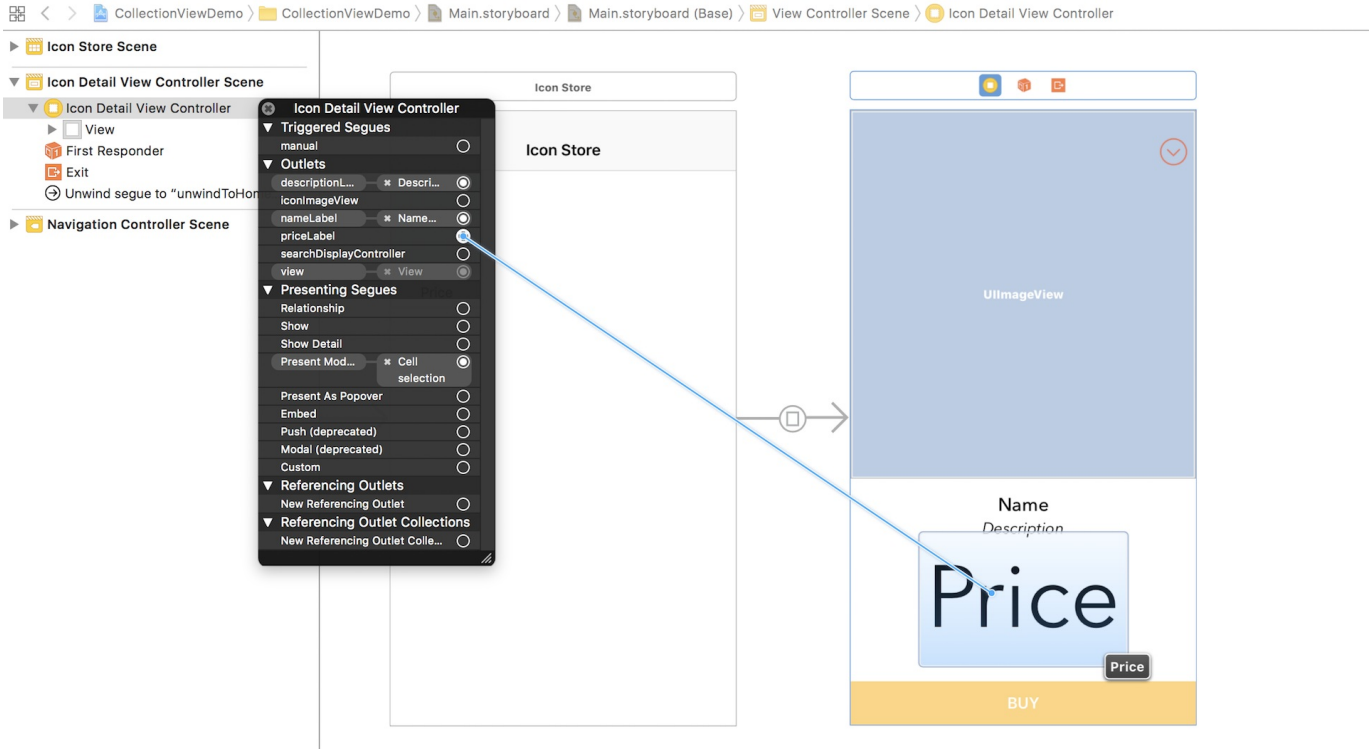


Figure 19.10. Establishing a connection between the outlet variable and the UI elements

Data Passing Between Controllers

In order to let other controllers pass the selected icon, we already added an `icon` property in the `IconDetailViewController` class:

```
var icon: Icon?
```

The question is: *How can we identify the selected item of the collection view and pass the selected icon to the IconDetailViewController?*

If you understand how data passing works via a segue, you know we must implement the `prepare(for:sender:)` method in the `IconCollectionViewController` class. Select `IconCollectionViewController.swift` and add the following code:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "showIconDetail" {
        if let indexPaths = collectionView?.indexPathsForSelectedItems {
            let destinationController = segue.destination as! IconDetailViewContro
ller
            destinationController.icon = iconSet[indexPaths[0].row]
            collectionView?.deselectItem(at: 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 icon, and pass it to the detail 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 icons. You should see a modal view showing the details of the icon.

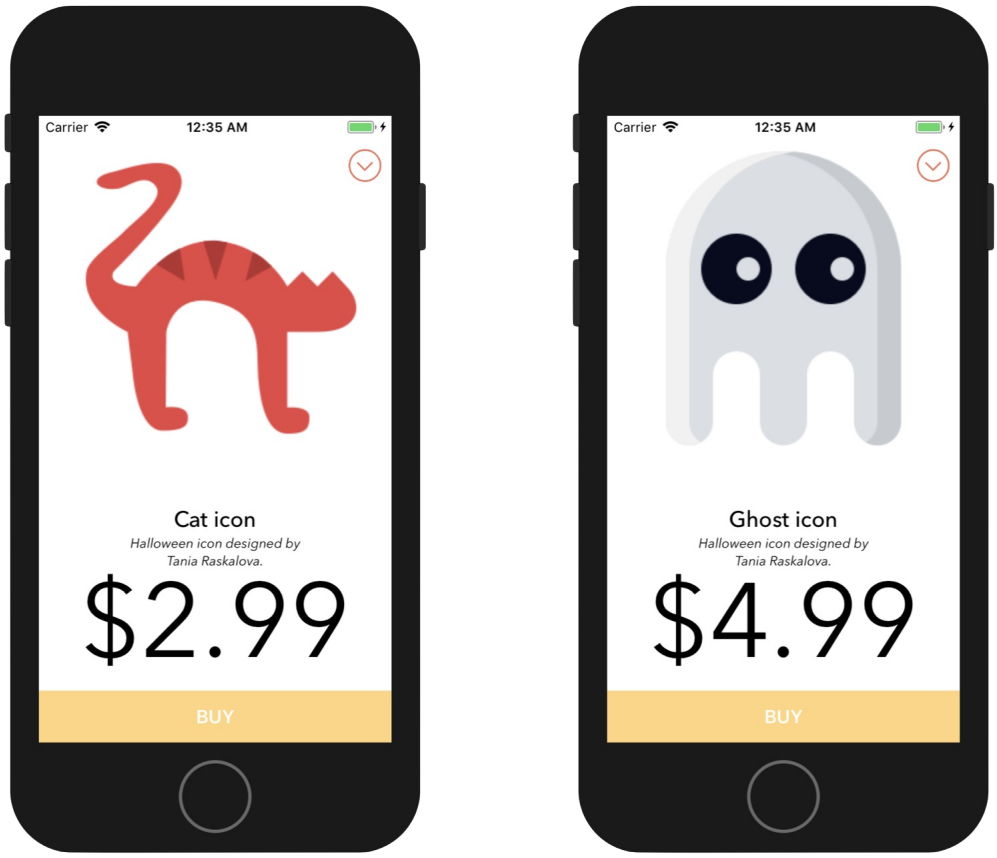


Figure 19.11. Sample details view of the icons

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 icons and share them by bringing up the activity controller:

- A user taps the *Share* button in the navigation bar. Once the sharing starts, the button title is automatically changed to *Done*.
- The user selects the icons to share.

- After selection, the user taps the *Done* button. The app will take a snapshot of the selected icon and then bring up the activity controller for sharing the icons using AirDrop, Messages or whatever service available.

We'll first add the *Share* button in the navigation bar of Icon Collection View Controller. Go to `Main.storyboard`, drag a Bar Button Item from the Object library, and put it in the navigation bar of Icon Collection View Controller. Set the title to `Share`.

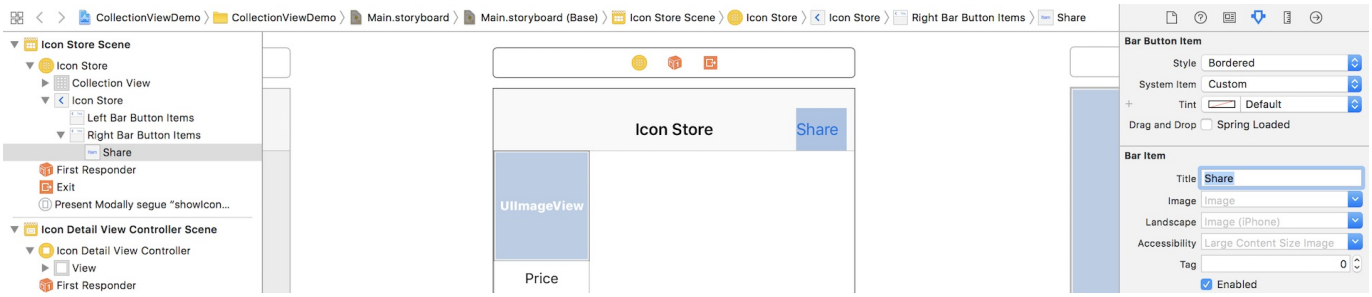


Figure 19.12. Add a bar button item to the navigation bar

In `IconCollectionViewController.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 *action* button and the `shareButtonTapped` method. Also, associate it with the `shareButton` outlet.

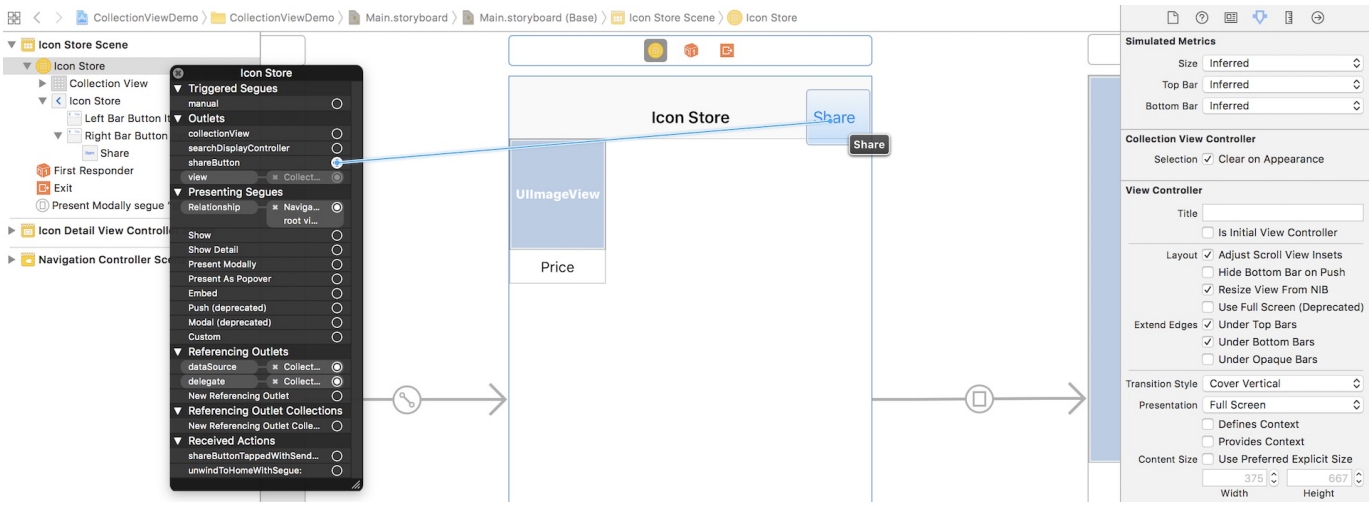


Figure 19.13. Establish a connection between the action button and the action method

The demo app now offers two modes: *single selection* and *multiple selections*. When a user taps the *action* button, the app goes into multiple selection mode. This allows users to select multiple icons for sharing. To support multiple selection mode, we'll add two variables in the `IconCollectionViewController` 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.
- `selectedIcons` – this is an array of tuples to store the selected icons. For each tuple, it stores the selected icon and the corresponding snapshot, which is an instance of `UIImage`.

Your code should look like this:

```
private var shareEnabled = false
private var selectedIcons: [(icon: Icon, snapshot: UIImage)] = []
```

Let's first start with the snapshot. How can you take a snapshot of a cell?

A collection view cell is essentially a subclass of `UIView`. To empower a cell with the snapshot capability, let's create an extension for the `UIView` class. In the project navigator, right-click `CollectionViewDemo` to create a new group named `Extension`. Then

right-click the *Extension* folder again to create a new Swift file. Name it `UIView+Snapshot.swift`. Once the file is created, replace its content with the following:

```
import UIKit

extension UIView {
    var snapshot: UIImage? {
        var image: UIImage? = nil
        UIGraphicsBeginImageContext(bounds.size)
        if let context = UIGraphicsGetCurrentContext() {
            self.layer.render(in: context)
            image = UIGraphicsGetImageFromCurrentImageContext()
        }
        UIGraphicsEndImageContext()

        return image
    }
}
```

We define a computed property named `snapshot`. In the code, we creates a bitmap-based graphics context with the view's size. Then we render the view's content over that graphics context and retrieve an image from it (i.e. `UIGraphicsGetImageFromCurrentImageContext()`). This is how we take the snapshot of a view or a collection view cell.

With the implementation of snapshots ready, let's continue to develop the *Share* feature.

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(_:didSelectItemAt:)` method will be called. We will implement this method and add the selected items into the `selectedIcons` array. Insert the following method in the `IconCollectionViewController` class:

```
override func collectionView(_ collectionView: UICollectionView, didSelectItemAt i
ndexPath: IndexPath) {
    // Check if the sharing mode is enabled, otherwise, just leave this method
    guard shareEnabled else {
        return
    }

    // Determine the selected items by using the indexPath and take a snapshot
    let selectedIcon = iconSet[indexPath.row]
    if let snapshot = collectionView.cellForItem(at: indexPath)?.snapshot {
        // Add the selected item into the array
        selectedIcons.append((icon: selectedIcon, snapshot: snapshot))
    }

}
```

The code above is pretty straightforward. We find out the index of the selected cell by accessing the `row` property of the index path. Then we put the corresponding icon object and its snapshot in the `selectedIcons` array.

Next, how can we highlight the selected cell?

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 *icon-selected.png* file in the starter project. Edit the `collectionView(_:cellForItemAt:)` method and insert the following line of code:

```
cell.selectedBackgroundView = UIImageView(image: UIImage(named: "icon-selected"))
```

Now when a user selects an icon, the selected cell will be highlighted.

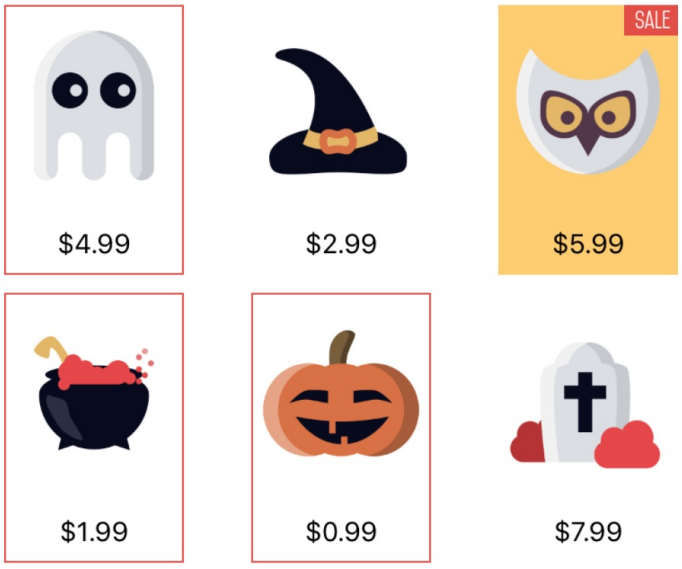


Figure 19.14. The selected cell is highlighted with a frame

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 `selectedIcons` array.

Insert the following code in the `IconCollectionViewController` class:

```
override func collectionView(_ collectionView: UICollectionView, didDeselectItemAt
 indexPath: IndexPath) {

    // Check if the sharing mode is enabled, otherwise, just leave this method
    guard shareEnabled else {
        return
    }

    let deSelectedIcon = iconSet[indexPath.row]

    // Find the index of the deselected icon. Here we use the index method and pas
s it
    // a closure. In the closure, we compare the name of the deselected icon with
all
    // the items in the selected icons array. If we find a match, the index method
 will
    // return us the index for later removal.
    if let index = selectedIcons.firstIndex(where: { $0.icon.name == deSelectedIco
n.name }) {
        selectedIcons.remove(at: index)
    }
}
```

When an item in a collection view is deselected, the `collectionView(_:didDeselectItemAt:)` method of the `UICollectionViewDelegate` protocol is called. In the method, we identify the deselected icon, and then remove it from the `selectedIcons` array.

Next, we'll move onto the implementation of the `shareButtonTapped` method. The method is called when a user taps the *Share* button. Update the method to the following code:

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

    guard shareEnabled else {
        // Change shareEnabled to YES and change the button text to Done
        shareEnabled = true
        collectionView?.allowsMultipleSelection = true
        shareButton.title = "Done"
        shareButton.style = UIBarButtonItem.Style.plain
```

```
            return
        }

        // Make sure the user has selected at least one icon
        guard selectedIcons.count > 0 else {
            return
        }

        // Get the snapshots of the selected icons
        let snapshots = selectedIcons.map { $0.snapshot }

        // Create an activity view controller for sharing
        let activityController = UIActivityViewController(activityItems: snapshots, ap
plicationActivities: nil)

        activityController.completionWithItemsHandler = { (activityType, completed, re
turnedItem, error) in

            // Deselect all selected items
            if let indexPaths = self.collectionView?.indexPathsForSelectedItems {
                for indexPath in indexPaths  {
                    self.collectionView?.deselectItem(at: indexPath, animated: false)
                }
            }

            // Remove all items from selectedIcons array
            self.selectedIcons.removeAll(keepingCapacity: true)

            // Change the sharing mode to NO
            self.shareEnabled = false
            self.collectionView?.allowsMultipleSelection = false
            self.shareButton.title = "Share"
            self.shareButton.style = UIBarButtonItem.Style.plain
        }

        present(activityController, animated: true, completion: nil)

    }
```

Let's take a look at the above code line by line.

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 *Done*.

When the app is in sharing mode (i.e. `shareEnabled` is set to `true`), we check if the user has selected at least one icon. If no icon is selected, we will not perform the sharing action.

Assuming the user has selected at least one icon, we'll bring up the activity view controller. I have covered this type of controller in chapter 17. You can refer to that chapter if you do not know how to use it. In brief, we pass an array of the snapshots to the controller for sharing.

The `completionWithItemsHandler` property takes in a closure which will be executed after the activity view controller is dismissed. The code in the closure is used to perform cleanup and revert the share button back to the original state.

Finally, we call up the `present(_:animated:completion:)` method to display the activity view controller on screen.

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 icons - 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 `shouldPerformSegue(withIdentifier:sender:)` method allows you to control the performance of a segue. Insert the following code snippet in `IconCollectionViewController.swift`:

```
override func shouldPerformSegue(withIdentifier identifier: String, sender: Any?)
-> Bool {
    if identifier == "showIconDetail" {
        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 icons and tap the *Done* button to share them on Messages or any available app.

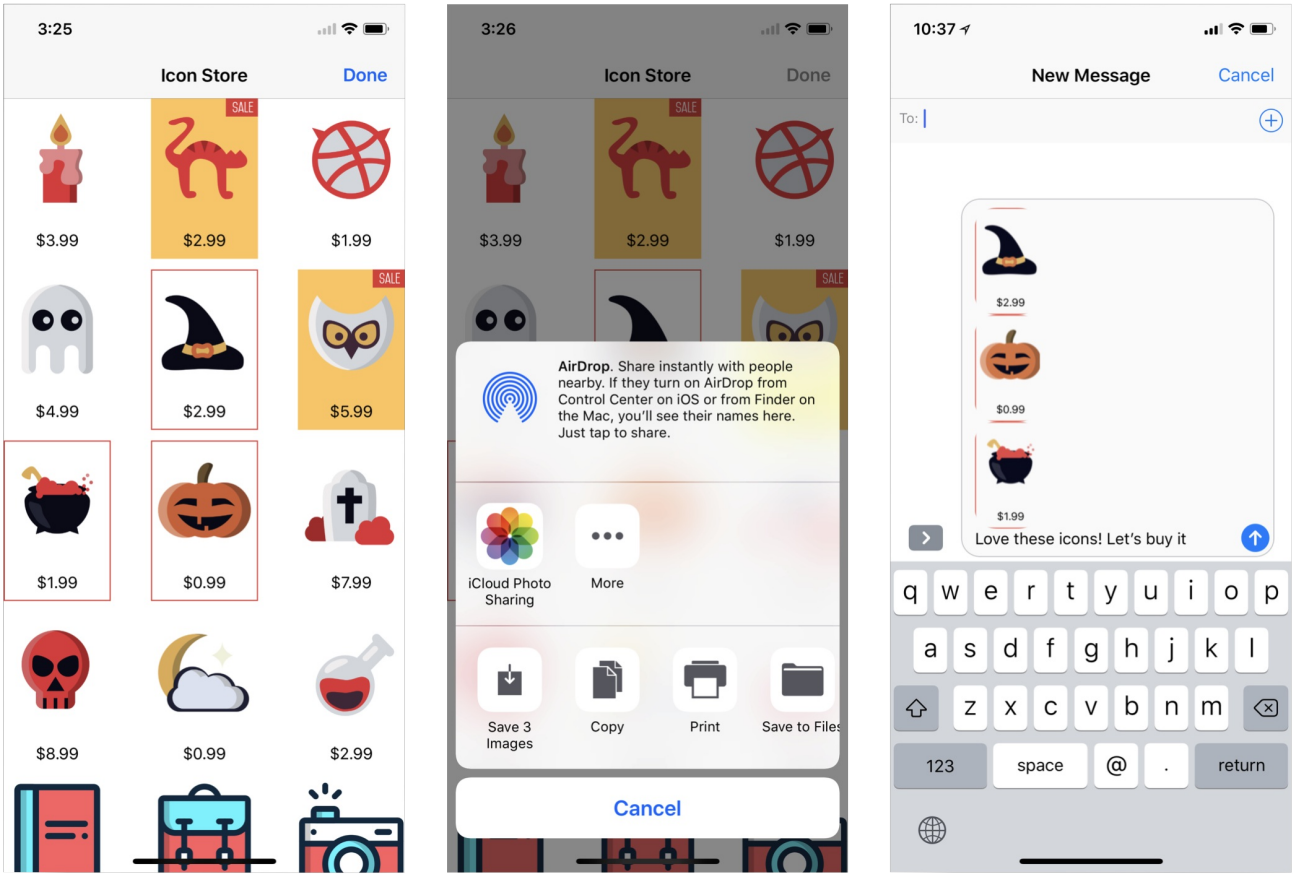


Figure 19.15. Sharing the icons over Twitter

For reference, you can download the Xcode project from http://www.appcoda.com/resources/swift5/CollectionViewSelection.zip.

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 SE. But if you run the app on iPhone 8/8 Plus/11, your screen should look like the screenshot shown below. The icons are displayed in grid format but with a large space between items.

The screen of iPhone 8 or iPhone 11 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.

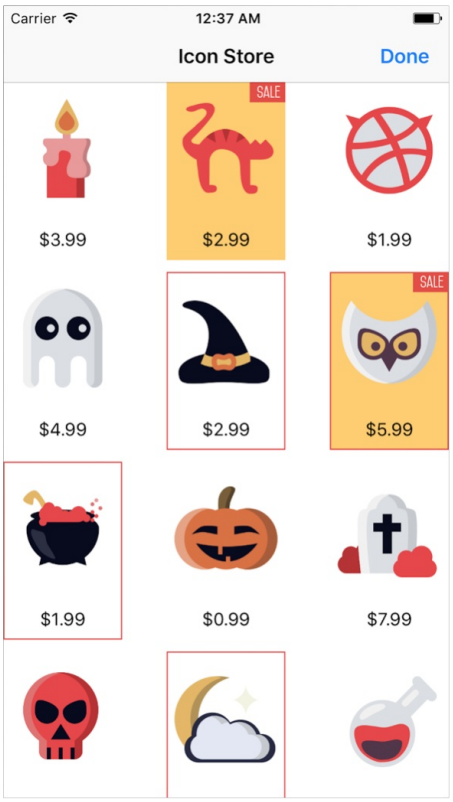


Figure 20.1. Running the collection view demo on iPhone 8 Plus

So how can we fix the issue? As mentioned in the first chapter of the book, iOS comes with a 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 http://www.appcoda.com/resources/swift5/DoodleFunStarter.zip. I have included a set of Doodle images (provided by the team at RoundIcons) and prebuilt the storyboard for you.

The starter project supports universal devices as I am going to show you how to create a collection view that adapts to different screen sizes. If you open the `Main.storyboard` file, you will find an empty view controller, embedded in a navigation controller. This is our starting point. We're going to design the collection view.

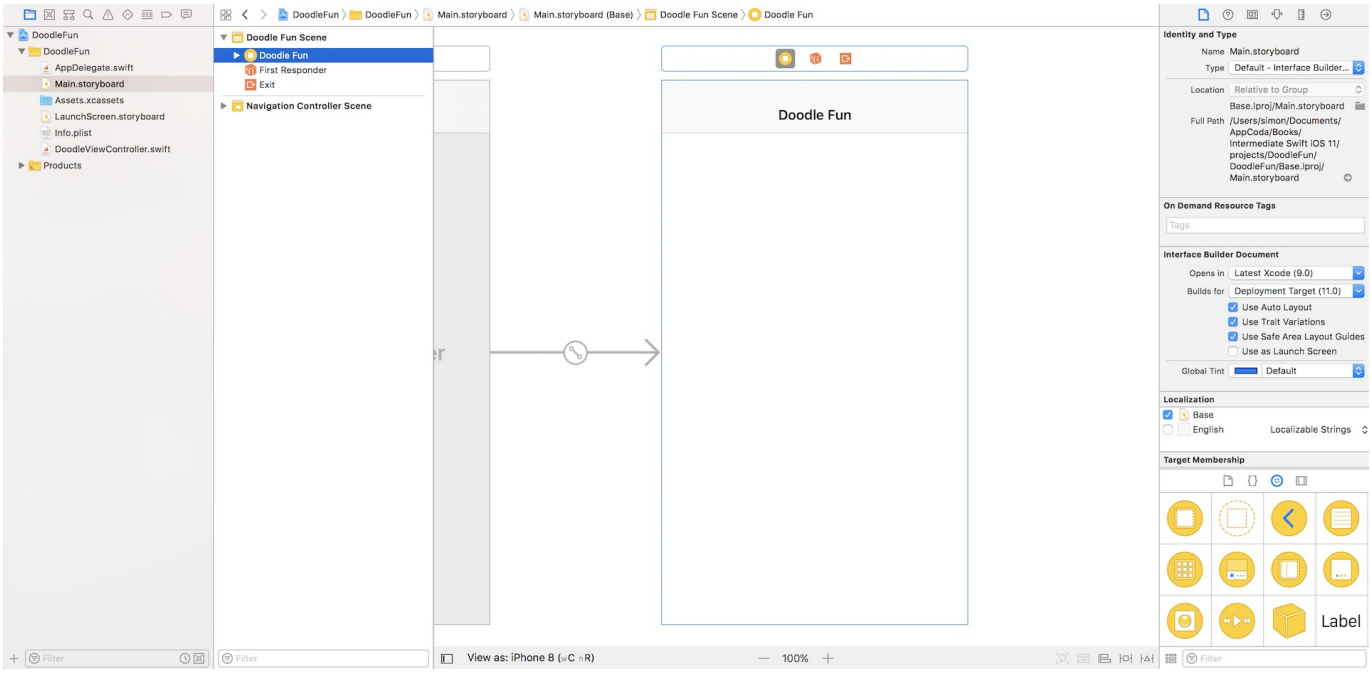


Figure 20.2. Creating a new Xcode project

Drag a Collection View object from the Object library to the View Controller. Resize it (375x667) 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. If everything is configured correctly, your screen should look like figure 20.3.

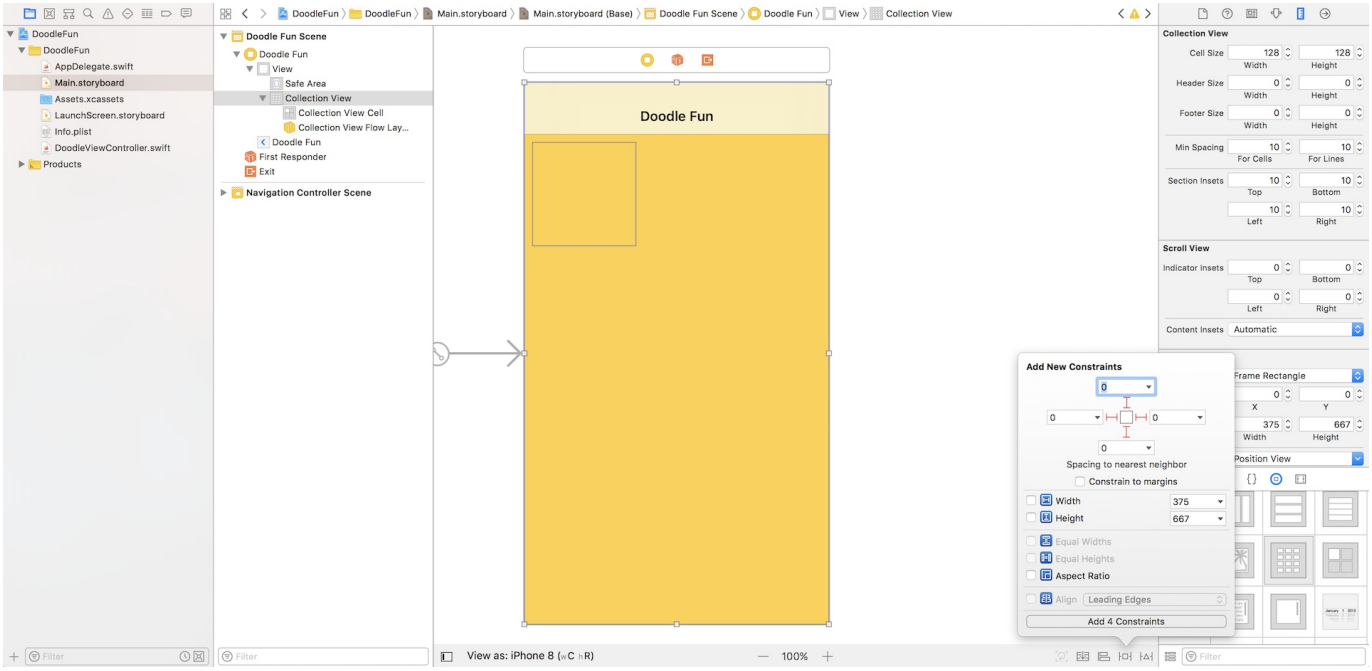


Figure 20.3. Creating a collection view in the view controller

Because we are using a Collection View instead of Collection View Controller, we have to deal with auto layout constraints on our own. Click the `Add New Constraints` button in the layout bar. Define four spacing constraints for the top, left, right and bottom sides of the collection view. Make sure you uncheck the `Constrain to margins` and set the spacing values for all sides to `0`. For the spacing constraint of the bottom side, please set its *Second Item* to `Superview.Bottom` instead of `Safe Area.bottom`. I want to extend the collection view to full screen on iPhone X.

Next, add an image view to the cell for displaying an image. Select the collection view cell and set its identifier to `Cell` under the Attributes inspector. Again, you will need to add a few layout constraints for the image view. Click the `Add New Constraints` button, and

define the spacing constraints for the top, left, right and bottom sides of the image view.

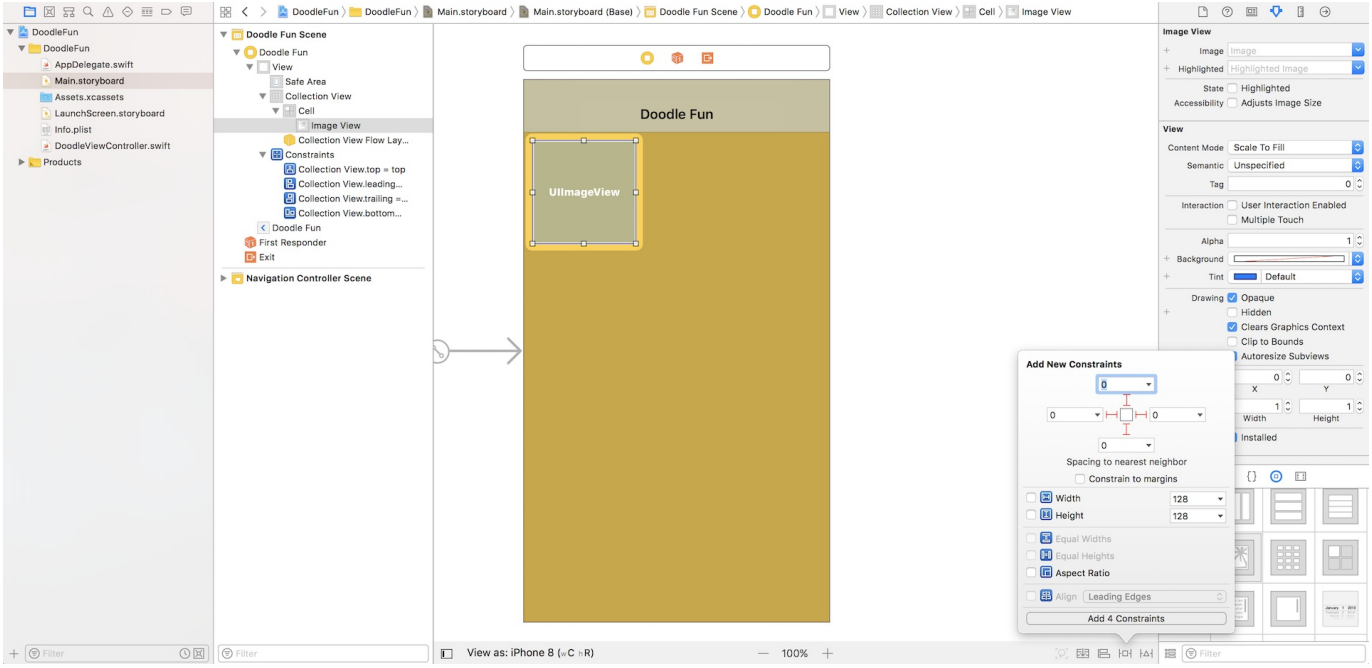


Figure 20.4. Adding an image view to the cell

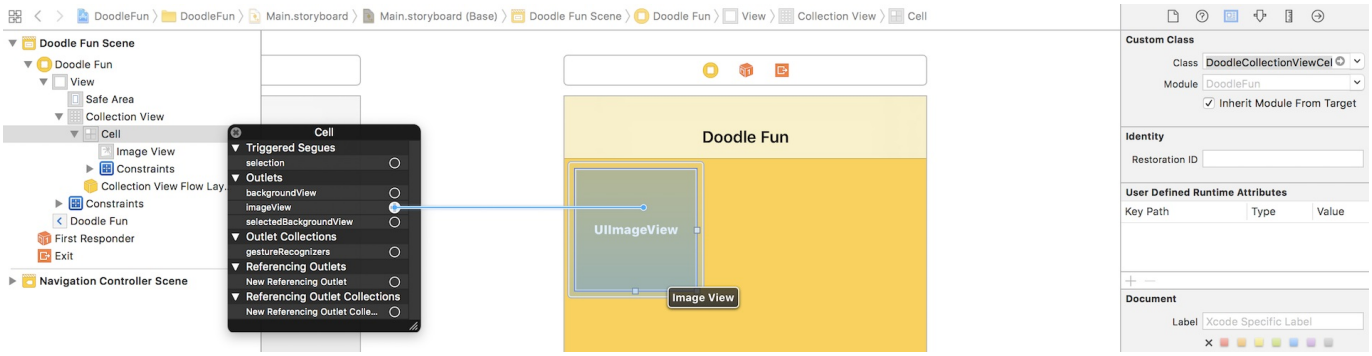
Diving into the Code

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 `DoodleCollectionViewCell` and set it as a subclass of `UICollectionViewCell`.

Once the file was created, declare an outlet variable for the image view:

```
class DoodleCollectionViewCell: UICollectionViewCell {
    @IBOutlet var imageView: UIImageView!
}
```

Switch to the storyboard. Select the collection view cell and change its custom class (under the Identity inspector) to `DoodleCollectionViewCell`. Then right-click the cell and connect the `imageView` outlet variable with the image view.



The `DoodleViewController` class is now associated with the view controller in the storyboard. As we want to present a set of images using the collection view, declare an array for the images and an outlet variable for the collection view:

```
private var doodleImages = ["DoodleIcons-1", "DoodleIcons-2", "DoodleIcons-3", "Do
odleIcons-4", "DoodleIcons-5", "DoodleIcons-6", "DoodleIcons-7", "DoodleIcons-8",
"DoodleIcons-9", "DoodleIcons-10", "DoodleIcons-11", "DoodleIcons-12", "DoodleIcon
s-13", "DoodleIcons-14", "DoodleIcons-15", "DoodleIcons-16", "DoodleIcons-17", "Do
odleIcons-18", "DoodleIcons-19", "DoodleIcons-20"]

@IBOutlet var collectionView: UICollectionView!
```

To populate data in the collection view, we have to adopt both the `UICollectionViewDataSource` and `UICollectionViewDelegate` protocols. In particular, here are the two methods we have to deal with:

- `collectionView(_:numberOfItemsInSection:)`
- `func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell`

Okay, let's implement the methods using an extension like this:

```
extension DoodleViewController: UICollectionViewDelegate, UICollectionViewDataSour
ce {

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

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPat
h: IndexPath) -> UICollectionViewCell {

        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "Cell",
for: indexPath) as! DoodleCollectionViewCell
        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. You can right-click `Collection View` to bring up a popover menu. And then drag from the circle of dataSource/delegate to the Doodle Fun view controller to establish the connection.

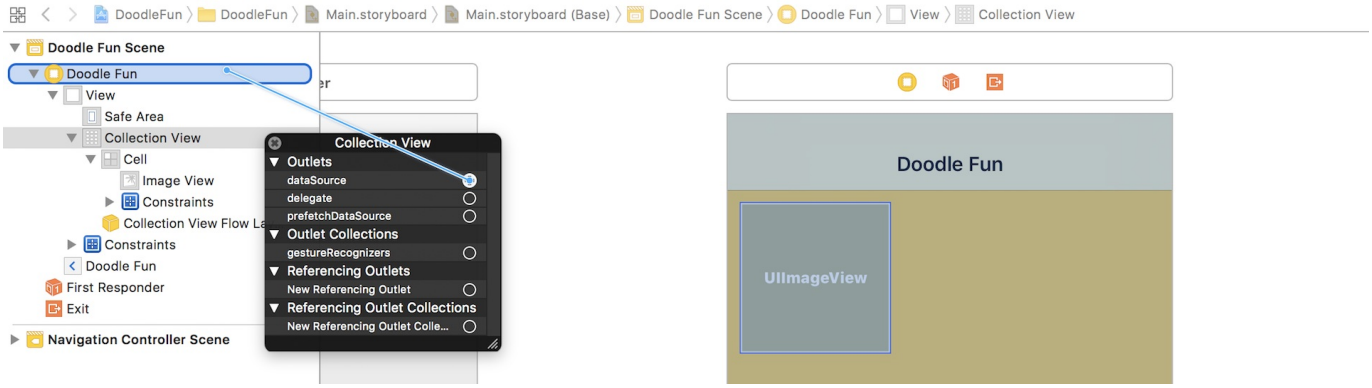


Figure 20.6. Associating the data source and delegate

If you do not want to use Interface Builder to establish the dataSource and delegate connections, you can write it in code like this:

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

    collectionView.delegate = self
    collectionView.dataSource = self
}
```

That's it! We're ready to test the app. Compile and run the app on iPhone SE 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 8, 8 Plus and iPhone X.

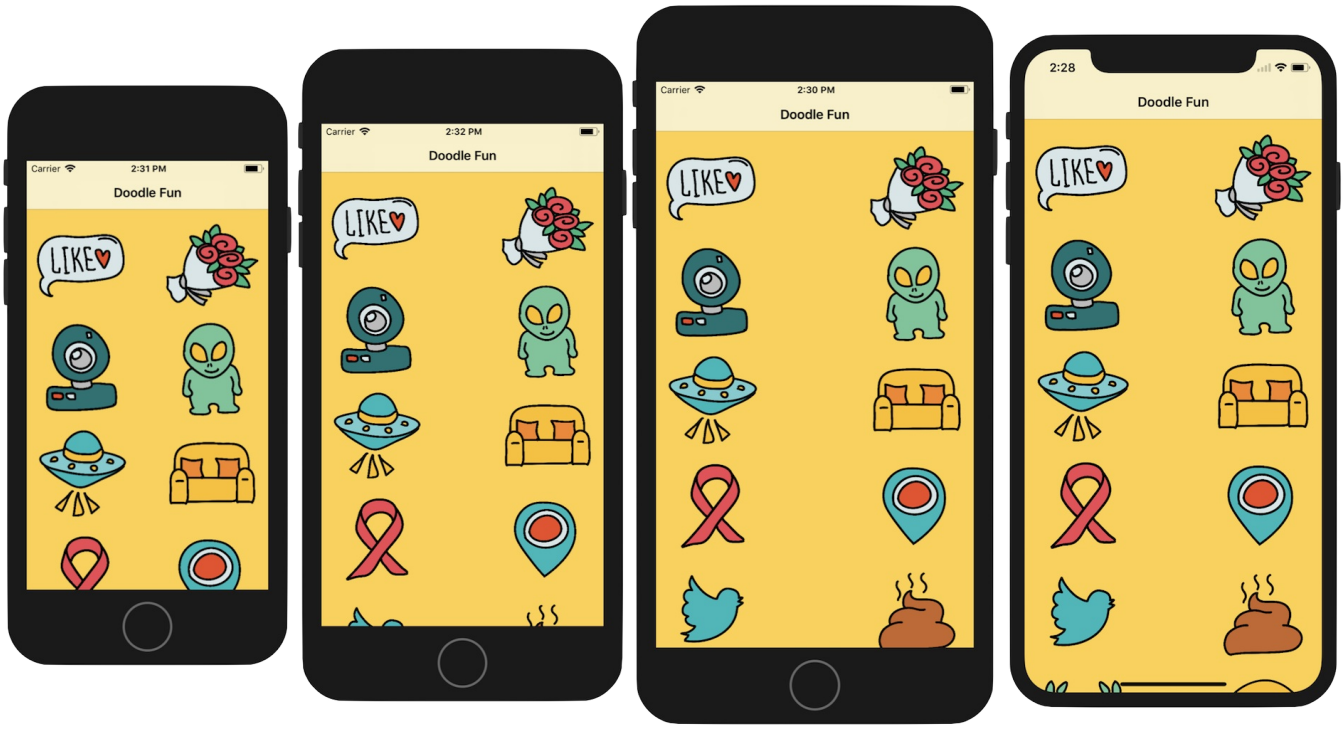


Figure 20.7. Doodle Fun app running on devices with different screen sizes

`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 8 (or iPhone X) and iPhone 8 Plus is 370 points and 414 points respectively. If you do a simple calculation for the iPhone 8 Plus (e.g. [414 - 20 (margin) - 20 (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 devices with 4.7-inch, 5.5-inch or 5.8-inch screen size? 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.current
let orientation = device.orientation
let isPhone = (device.userInterfaceIdiom == UIUserInterfaceIdiom.phone) ? true : f
alse

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:

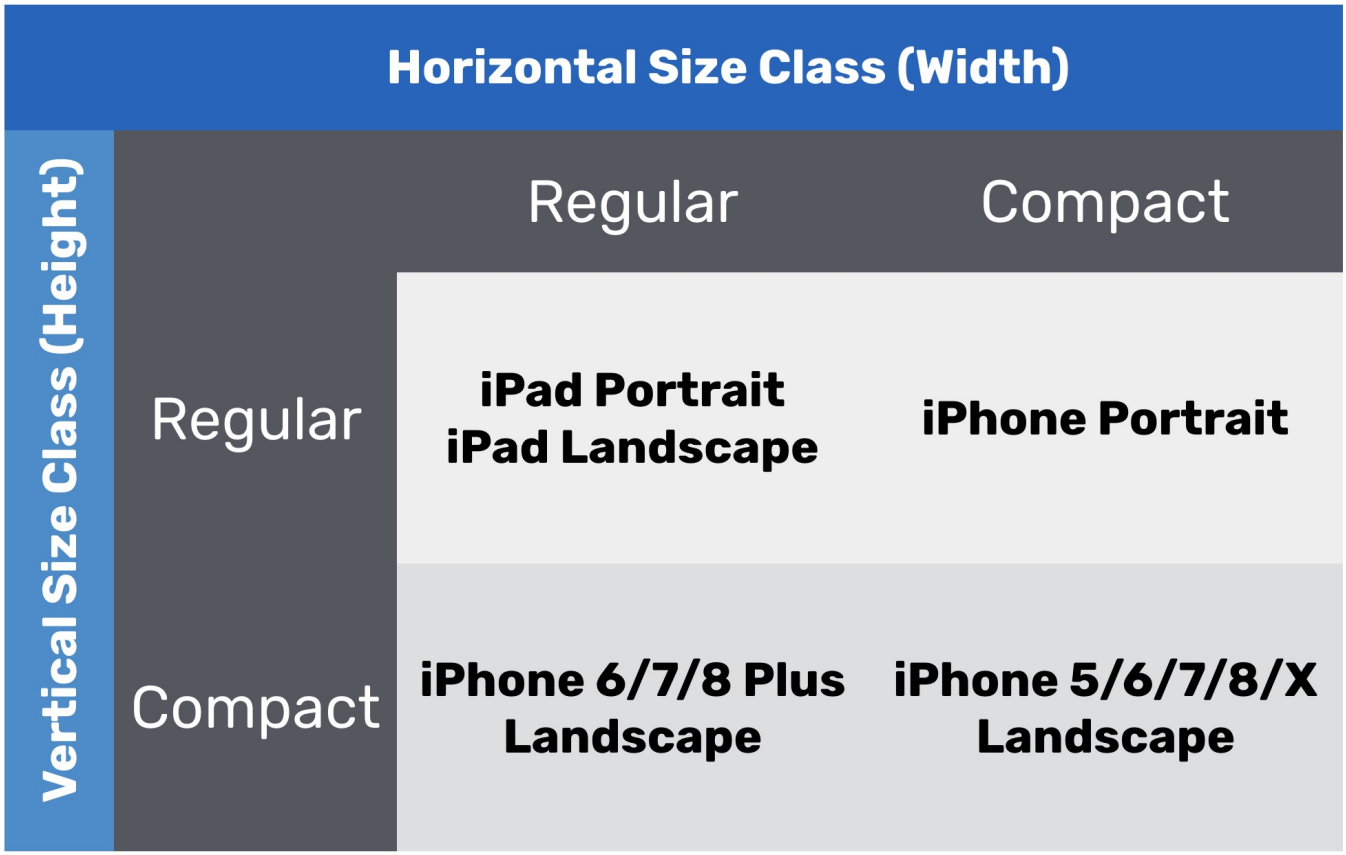


Figure 20.8. Size classes

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 8 Plus:

```
<UITraitCollection: 0x60c0002e2500; _UITraitNameUserInterfaceIdiom = Phone, _UITra
itNameDisplayScale = 3.000000, _UITraitNameDisplayGamut = P3, _UITraitNameHorizont
alSizeClass = Compact, _UITraitNameVerticalSizeClass = Regular, _UITraitNameTouchL
evel = 0, _UITraitNameInteractionModel = 1, _UITraitNameUserInterfaceStyle = 1, _U
ITraitNameUserInterfaceLayoutDirection = 0, _UITraitNameForceTouchCapability = 2,
_UITraitNamePreferredContentSizeCategory = UICTContentSizeCategoryXXL, _UITraitNam
eDisplayCornerRadius = 0.000000>
```

From the above information, you discover that the device is an iPhone which has 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 collections, you should understand how to determine the current size class of a device. Now it's time to make the collection view adaptive.

By default, a collection view uses `UICollectionViewFlowLayout` to organize the items into a grid. `UICollectionViewFlowLayout` actually adopts the `UICollectionViewDelegateFlowLayout` protocol and has implemented some of the required methods. If you look into the documentation of the protocol, it provides an optional method for specifying the size of a cell:

```
optional func collectionView(_ collectionView: UICollectionView, layout collection
ViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize
```

To resize a cell at runtime, all you need to do is adopt the protocol in `DoodleViewController` and implement the method to return the exact size of the cell for different screen sizes.

Now create another extension to adopt the `UICollectionViewDelegateFlowLayout` protocol. Because we only want to alter the cell size for iPhones in portrait mode, we will implement the method like this:

```
extension DoodleViewController: UICollectionViewDelegateFlowLayout {
    func collectionView(_ collectionView: UICollectionView, layout collectionViewL
ayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {

        let sideSize = (traitCollection.horizontalSizeClass == .compact && traitCo
llection.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 a device with a 4.7/5.5/5.8-inch screen. It should look much better now.

Figure 20.9. The collection view is now adaptive

Respond to the Change of Size Class

Did you try to test the app in landscape mode? When you turn the iPhone sideways, the cell size is 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 `DoodleViewController` class:

```
override func viewWillTransition(to size: CGSize, with coordinator: UIViewControll
erTransitionCoordinator) {

    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.

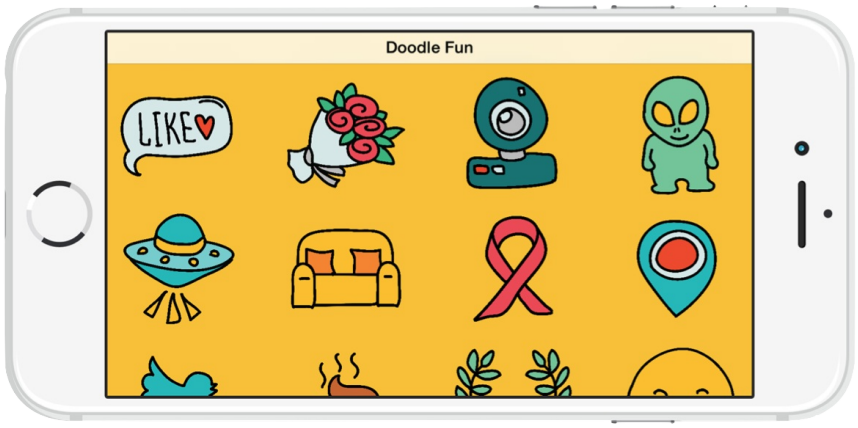


Figure 20.10. Doodle Fun app in landscape mode

For reference, you can download the complete project from http://www.appcoda.com/resources/swift5/DoodleFun.zip.

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.heig
ht)
```

Figure 20.11. Doodle Fun app now displays all images without scrolling

With the total area and the 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 http://www.appcoda.com/resources/swift5/DoodleFunExercise.zip.

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 shows some of the extension points available 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
- **iMessage** - Provides an extension for the Messages app such as stickers and iMessage applications.
- **Notifications** - Provides customizations for local and push notifications

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.

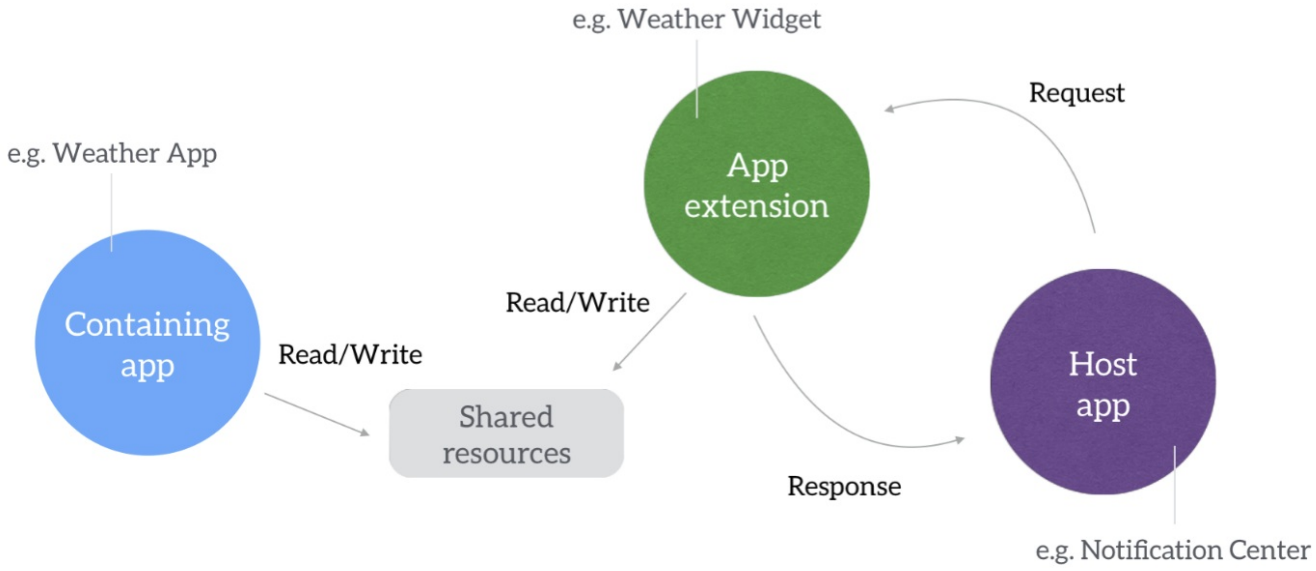


Figure 21.1. How app extensions work

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:completionHandler:` or a shared data container like the use of `UserDefaults` 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 http://www.appcoda.com/resources/swift5/WeatherDemo.zip. 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 set to *Paris, France*. The app, however, provides a setting screen for altering the default location. It relies on a free API provided by openweather.org to aggregate weather information. The API returns weather data for 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.

Figure 21.2. Weather app demo

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.

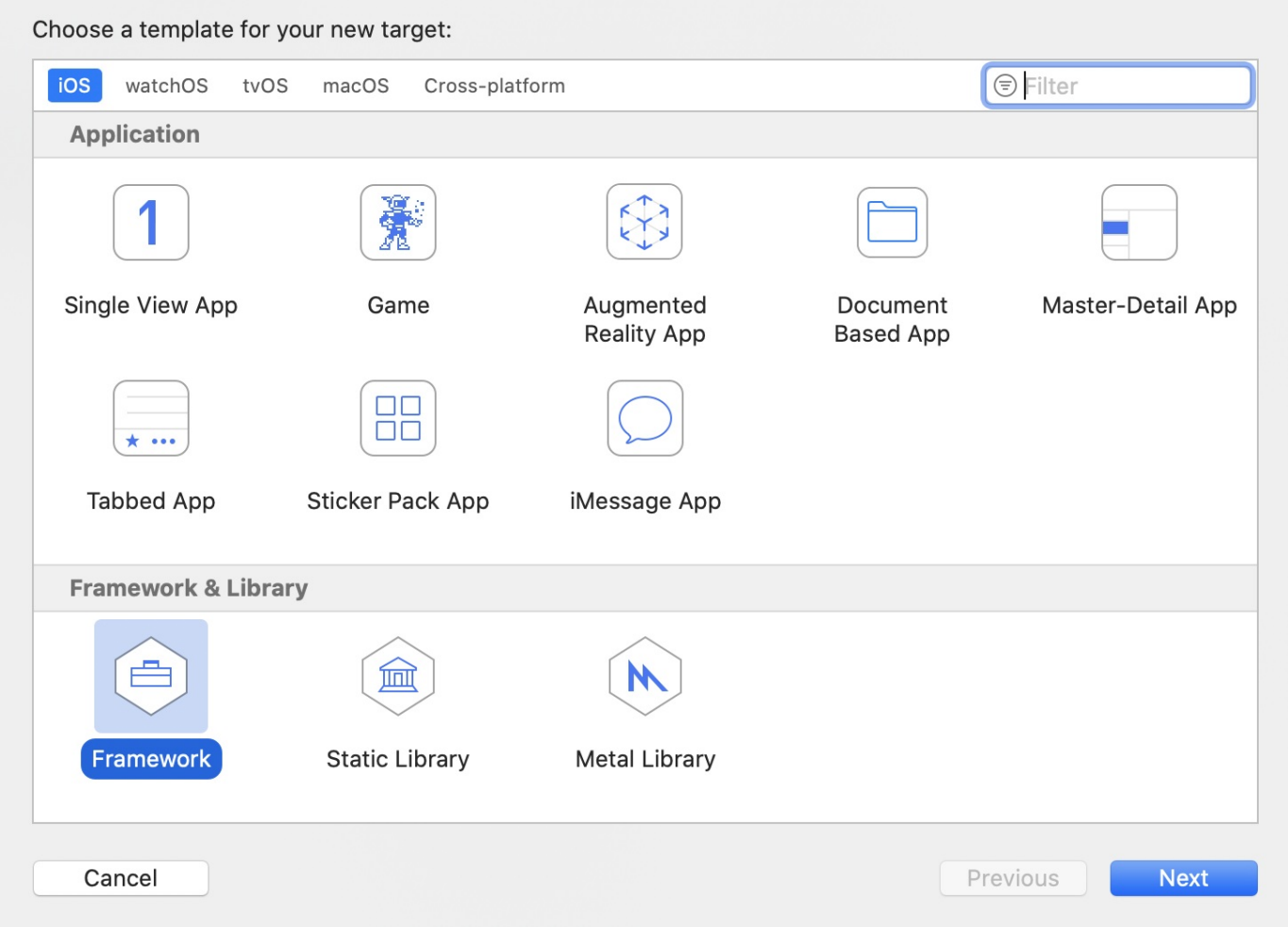
For example, both the weather app and the weather extension are required to use the `WeatherService` class to retrieve the latest weather information. You can replicate the files in both targets. But this is not a good practice. When developing an app or an extension, you should always consider code 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 the *iOS* tab. Scroll down and select `Framework` under Framework & Library.



Set its name to `WeatherInfoKit` and check that the language is `Swift`. Leave the rest of the options as they are and click `Finish`.

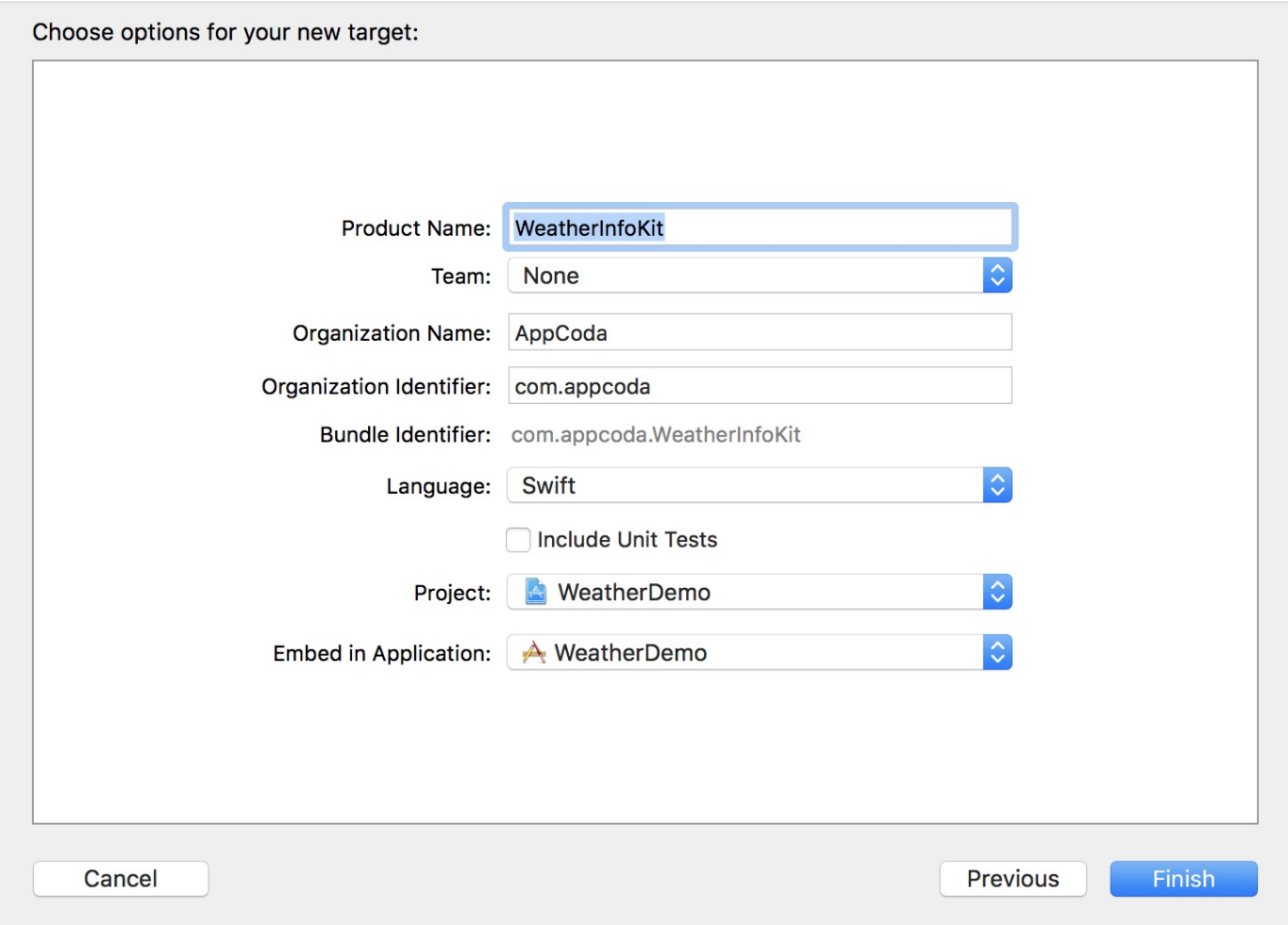


Figure 21.4. Set the product name to WeatherInfoKit

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`. Make sure the version of the target in the Deployment Info section is set to `11.0` or up because this Xcode project is set to support iOS 11.0 (or up).

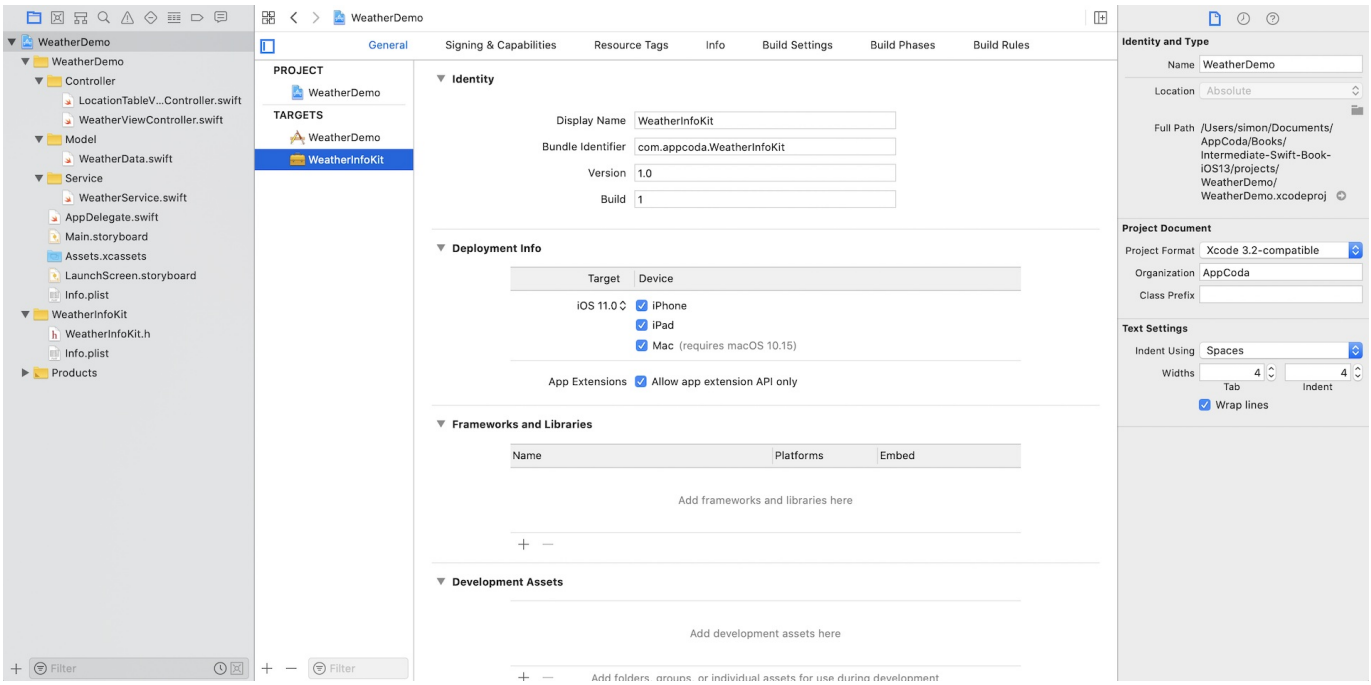


Figure 21.5. Enable the Allow App Extension API only option

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` structure 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.

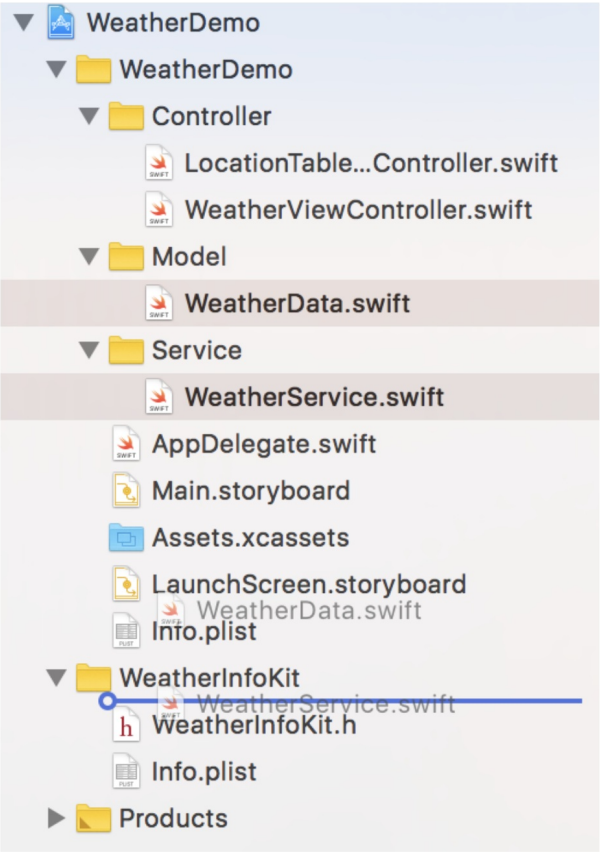


Figure 21.6. Move WeatherService.swift and WeatherData.swift to WeatherInfoKit

Now both `WeatherService.swift` and `WeatherData.swift` should be a part of the `WeatherInfoKit` target. Xcode has changed the file's target membership for you. But to play safe, let's make sure the setting is correct. Select the `WeatherService.swift` file from

the Project Navigator. Then open the File Inspector and verify the file target in the Target Membership section. The `WeatherInfoKit` option should be ticked. Repeat the process for the `WeatherData.swift` file.

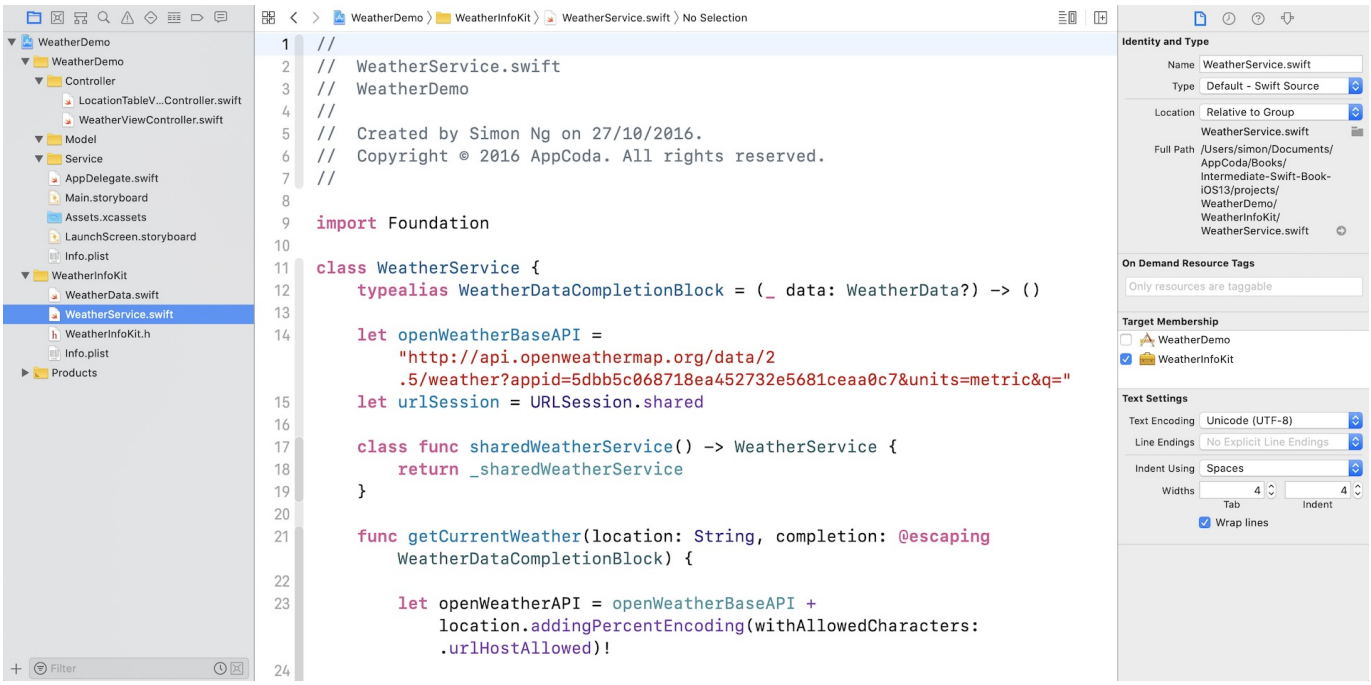


Figure 21.7. Verifying the target of the files

Because the `WeatherService` and `WeatherData` classes were removed from the `WeatherDemo` target, you'll end up with an error in `WeatherViewController.swift`.

Starting from Swift 3, the language provides five access levels for entities in your code: *open*, *public*, *internal*, *file-private* and *private*. For details of each access level, you can refer to Apple's official Swift documentation.

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 `WeatherViewController` 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.

So, open `WeatherData.swift` and add the `public` access modifier to the class declaration:

```
public struct WeatherData {

    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
=5dbb5c068718ea452732e5681ceaa0c7&units=metric&q="
    let urlSession = URLSession.shared

    public class func sharedWeatherService() -> WeatherService {
        return _sharedWeatherService
    }

    public func getCurrentWeather(location:String, completion: @escaping WeatherDa
taCompletionBlock) {
        ...
    }

}
```

After doing this, however, the errors in `WeatherViewController.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* and then choose *Today Extension*. Click `Next` to proceed.

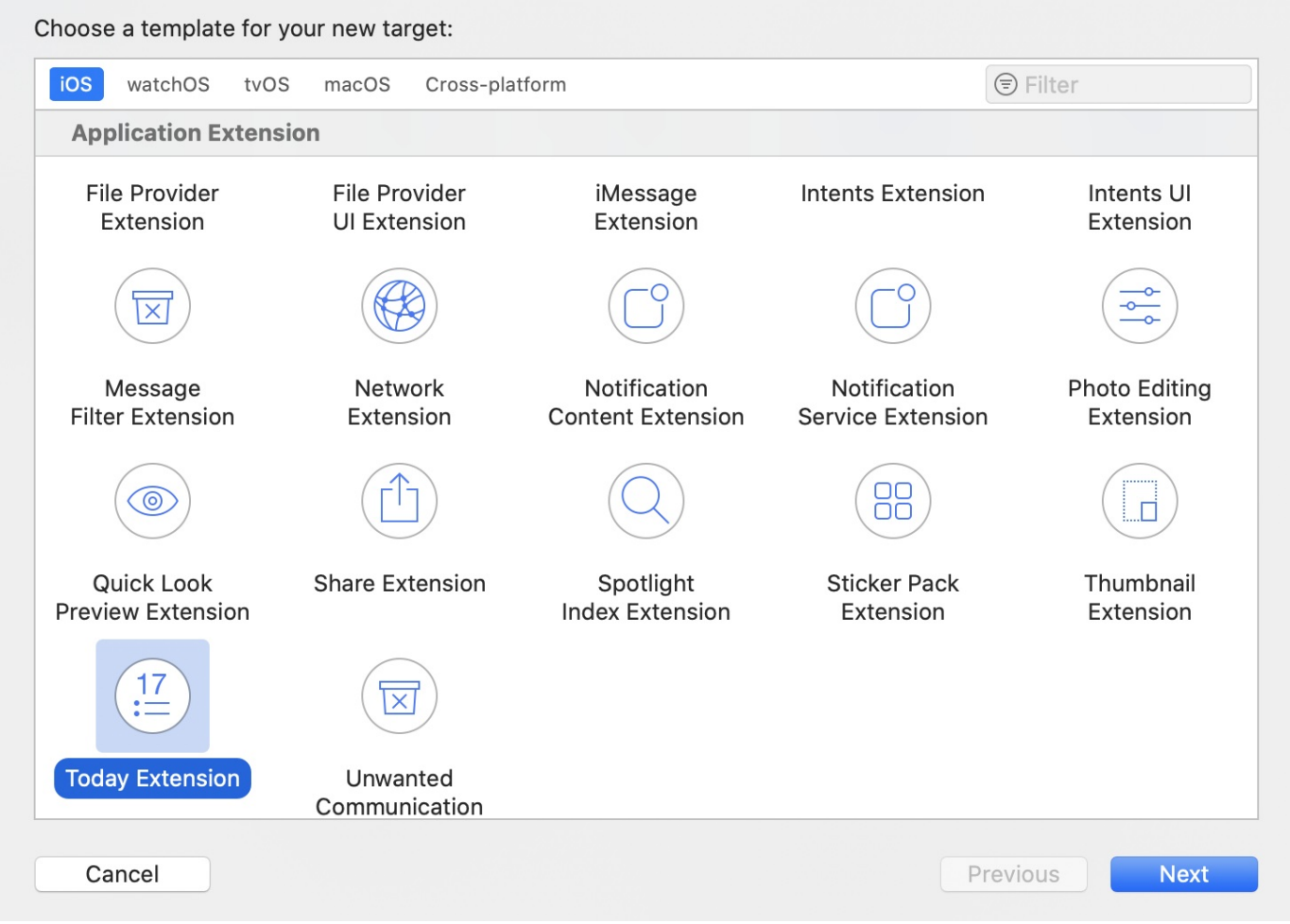


Figure 21.8. Choosing the Today Extension template

Set the Product Name to `Weather Widget` and leave the rest of the settings as they are. Click `Finish`.

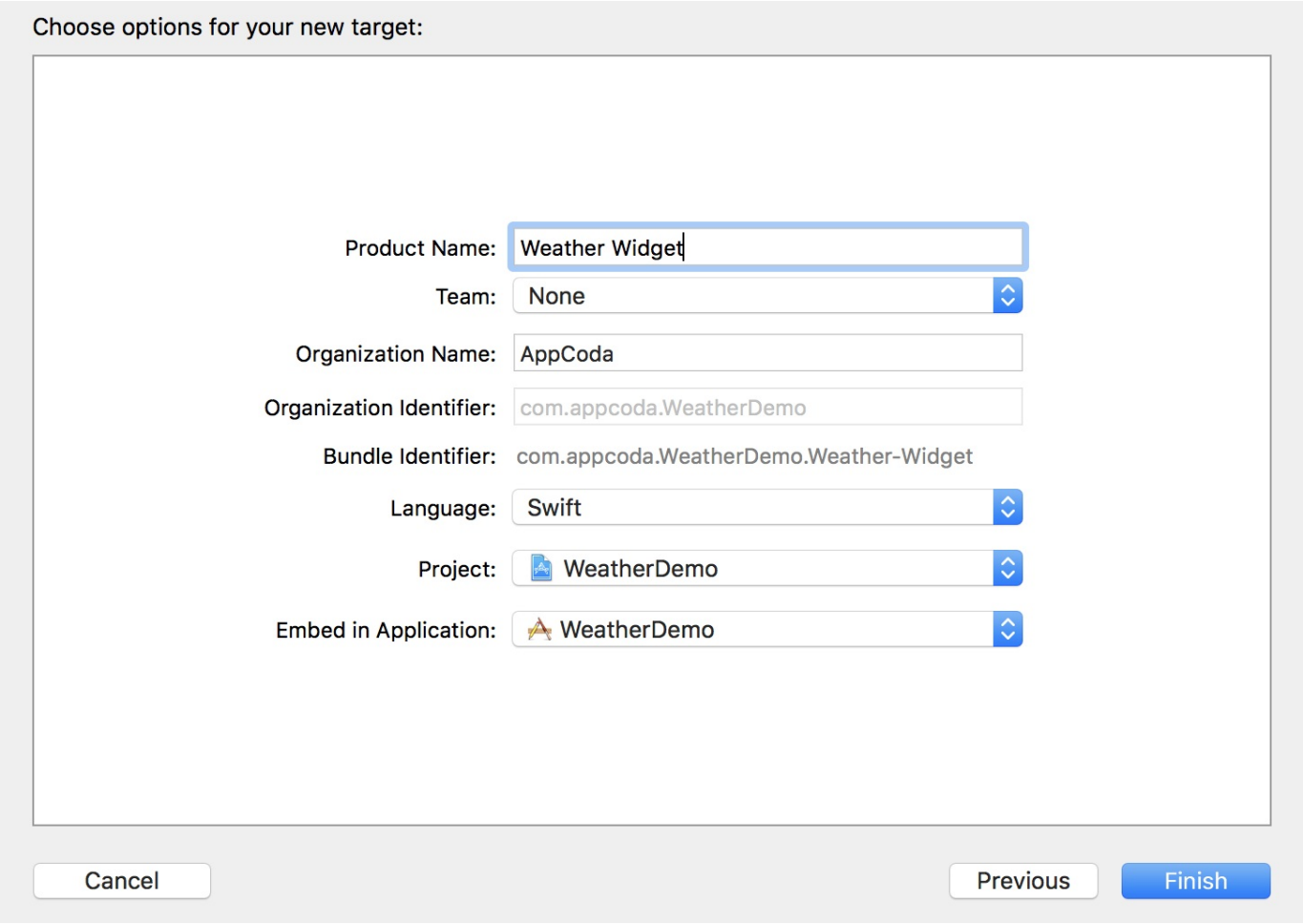


Figure 21.9. Filling in to the product name for the weather widget

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`. Make sure that the deployment target is set to `11.0` or up. Then on the General tab, press the `+` button under *Frameworks and Libraries*. Select `WeatherInfoKit.framework` and press `Add`.

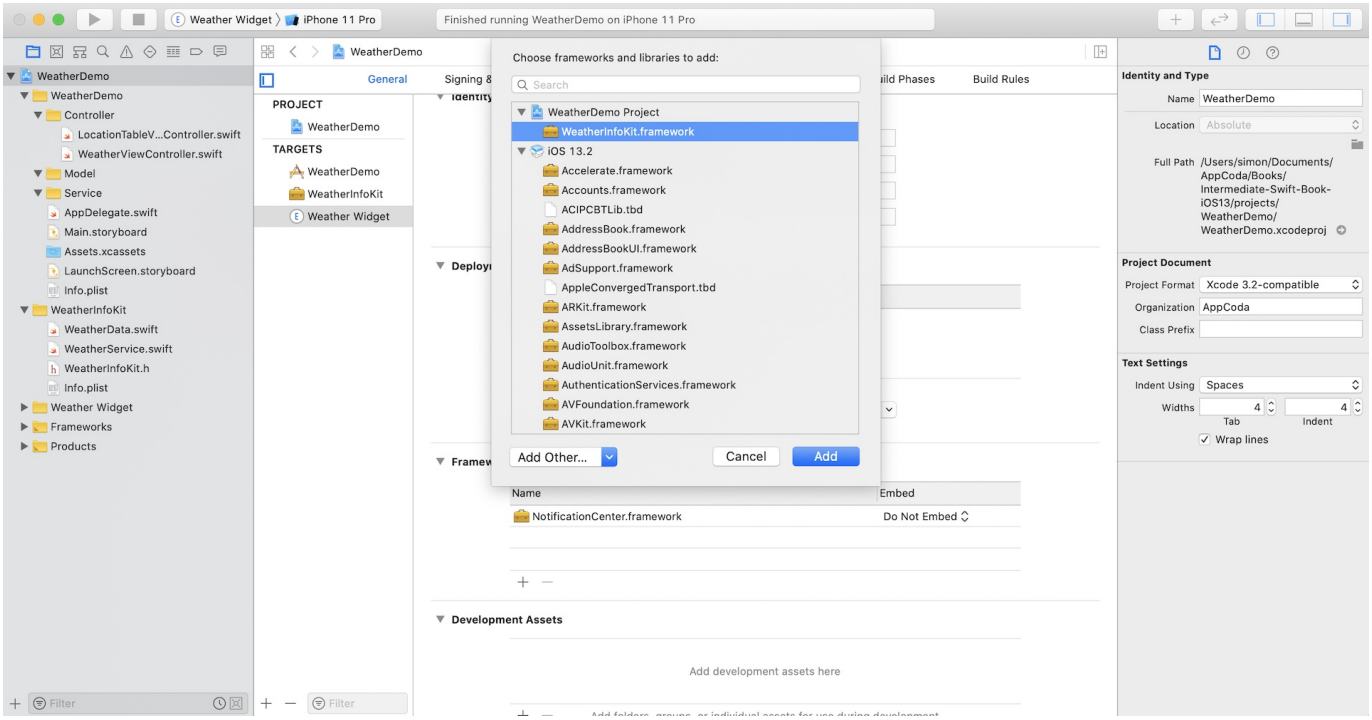


Figure 21.10. Adding the WeatherInfoKit.framework to the Weather Widget target

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.

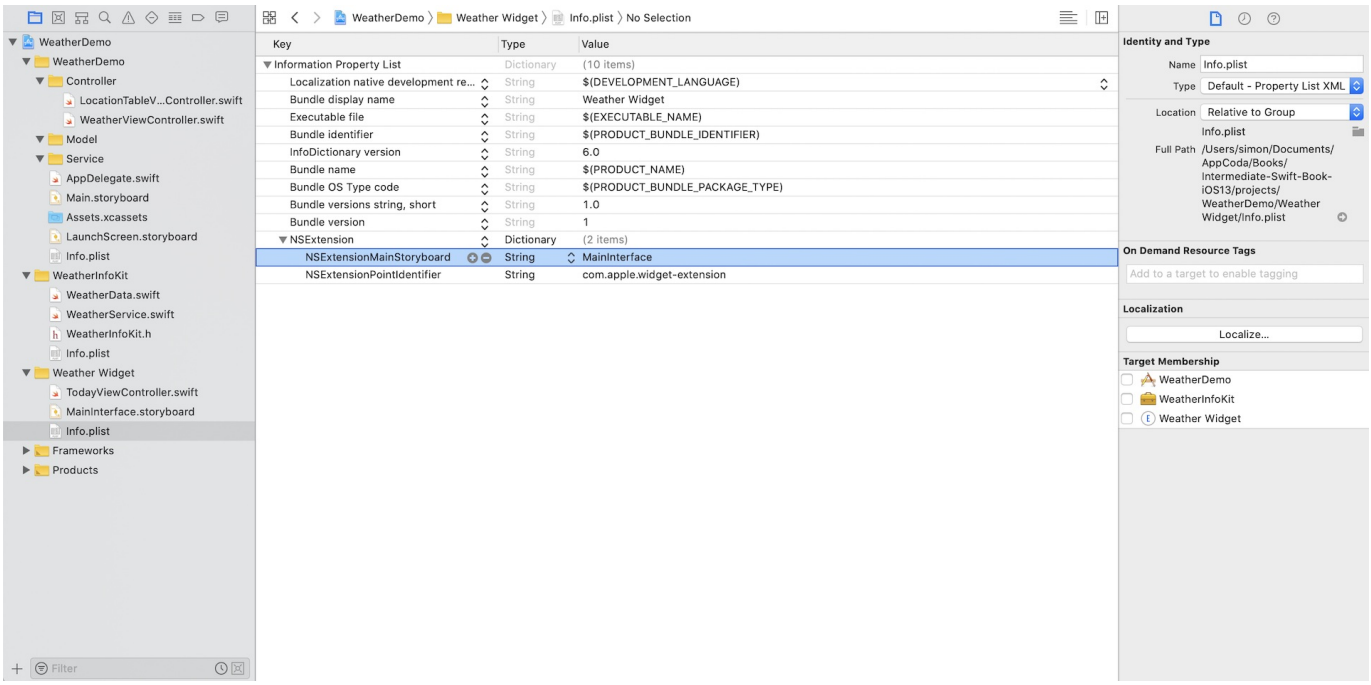


Figure 21.11. The NSExtension key in Info.plist

Open `MainInterface.storyboard`. You'll see a simple view with a Hello World label.

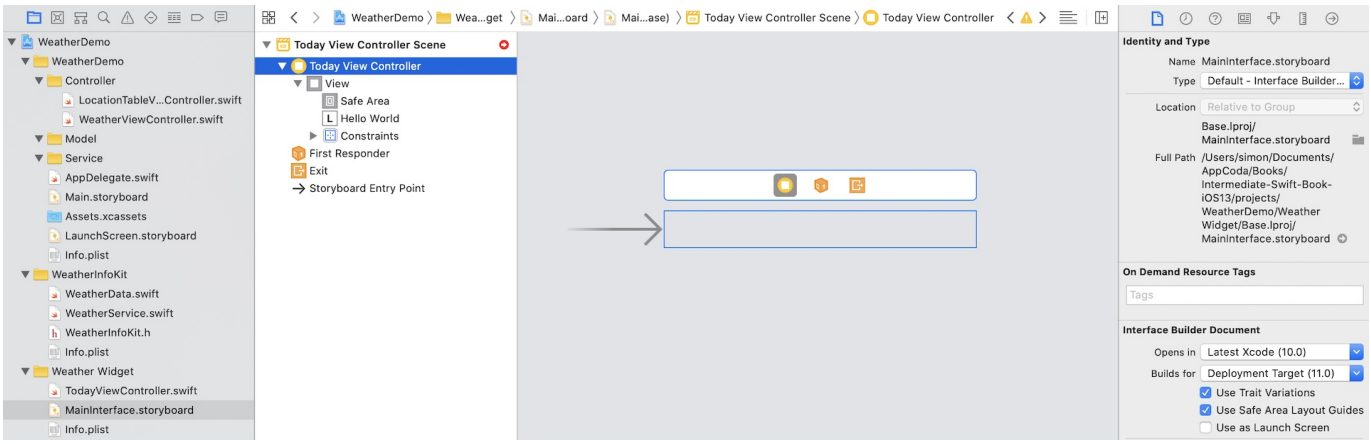


Figure 21.12. The default UI of MainInterface.storyboard

Let's have a quick test before redesigning the widget. To run the extension, make sure you select the `Weather Widget` scheme in Xcode's toolbar and hit Run.

Figure 21.13. Select the Weather Widget scheme to run the widget

The simulator will automatically launch the weather widget in Notification Center. In case you can't see the widget, stop the app and run it again.

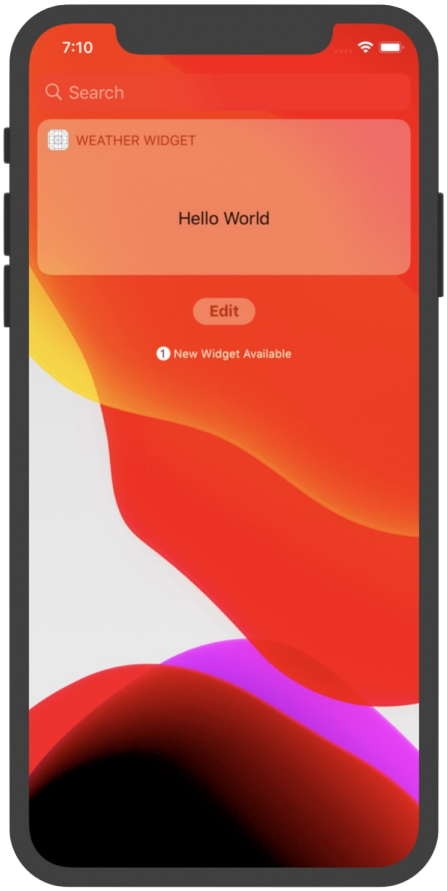


Figure 21.14. The default widget showing a Hello World message

To display the weather data in Today view, we have to redesign the Today View Controller in storyboard like this:

Figure 21.15. Designing the weather widget layout

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 `0` and define auto layout constraints such that it fits multiple screen resolutions.

Now, go to the `TodayViewController.swift` file and add the following import statement:

```
import WeatherInfoKit
```

Next, declare three outlet variables and a couple of properties for storing the location in `TodayViewController.swift`:

```
@IBOutlet var cityLabel: UILabel!
@IBOutlet var weatherLabel: UILabel!
@IBOutlet var temperatureLabel: UILabel!

private var city = "Paris"
private var country = "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.

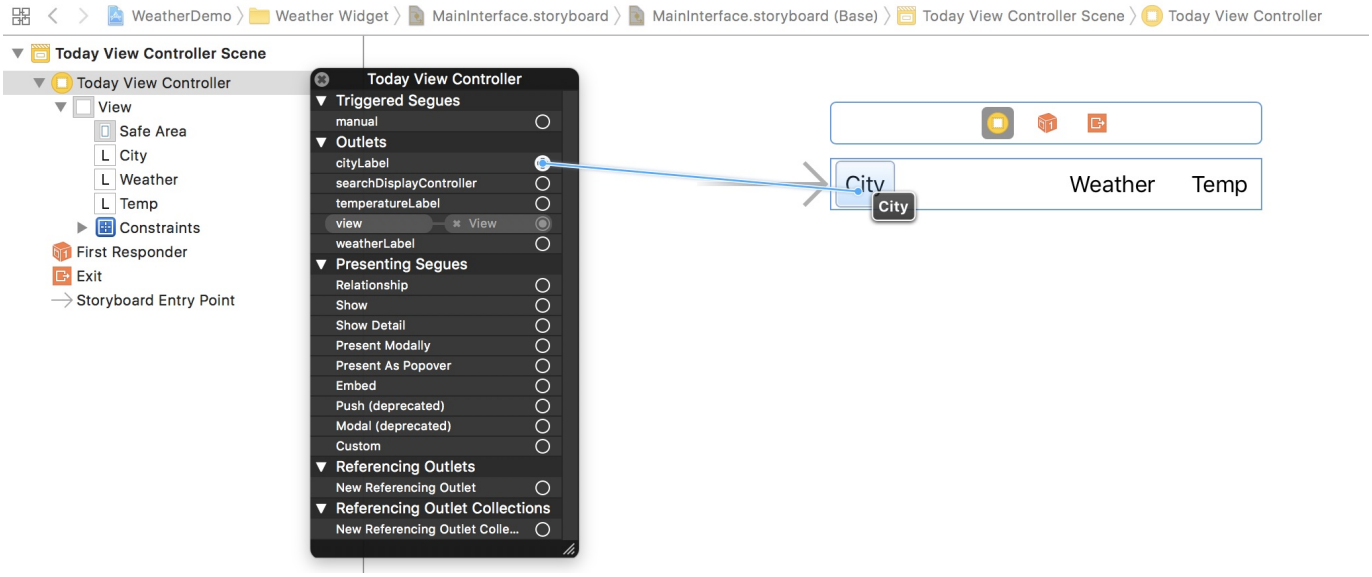


Figure 21.16. Connecting the outlet variables

To present the weather information in the widget's view controller, update the `viewDidLoad` method in `TodayViewController.swift`:

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

    cityLabel.text = "\(city), \(country)"

    // Invoke weather service to get the weather data
    WeatherService.sharedWeatherService().getCurrentWeather(location: city) { (dat
a) -> () in
        OperationQueue.main.addOperation({ () -> Void in
            if let weatherData = data {
                self.weatherLabel.text = weatherData.weather.capitalized
                self.temperatureLabel.text = String(format: "%d", weatherData.temp
erature) + "\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 `widgetPerformUpdate(completionHandler:)` method:

```
func widgetPerformUpdate(completionHandler: (@escaping (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 = city

    WeatherService.sharedWeatherService().getCurrentWeather(location: city) { (dat
a) -> () in
        guard let weatherData = data else {
            completionHandler(NCUpdateResult.noData)
            return
        }

        print(weatherData.weather)
        print(weatherData.temperature)

        OperationQueue.main.addOperation({ () -> Void in
            self.weatherLabel.text = weatherData.weather.capitalized
            self.temperatureLabel.text = String(format: "%d", weatherData.temperat
ure) + "\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 can update 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 the simulator. You will end up with this exception in the console:

```
Error Domain=NSURLErrorDomain Code=-1022 "The resource could not be loaded because
 the App Transport Security policy requires the use of a secure connection." UserI
nfo={NSUnderlyingError=0x60800024dcb0 {Error Domain=kCFErrorDomainCFNetwork Code=-
1022 "(null)"}, NSErrorFailingURLStringKey=http://api.openweathermap.org/data/2.5/
weather?appid=5dbb5c068718ea452732e5681ceaa0c7&units=metric&q=Paris,%20France, NSE
rrorFailingURLKey=http://api.openweathermap.org/data/2.5/weather?appid=5dbb5c06871
8ea452732e5681ceaa0c7&units=metric&q=Paris,%20France, NSLocalizedDescription=The r
esource could not be loaded because the App Transport Security policy requires the
 use of a secure connection.}
```

App Transport Security was 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 `App Transport Security Settings`. Then add the `Allow Arbitrary Loads` key with the type `Boolean`. By setting the key to `YES`, you explicitly disable App Transport Security.

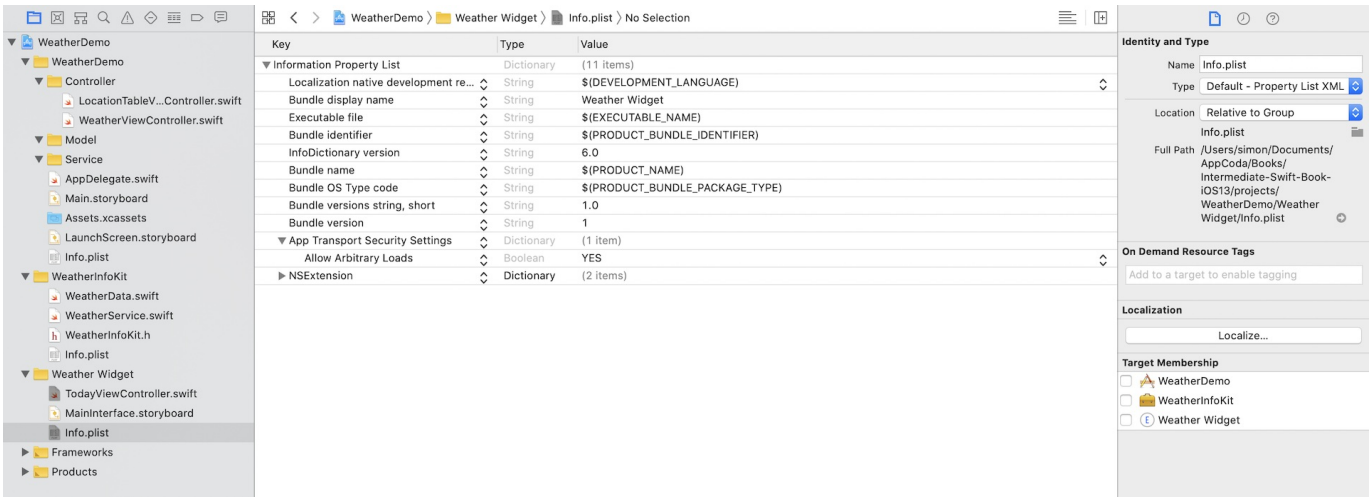


Figure 21.17. Disable ATS by setting Allow Arbitrary Loads to YES

Now run the app again. It should be able to load the widget. The weather widget should look like this:

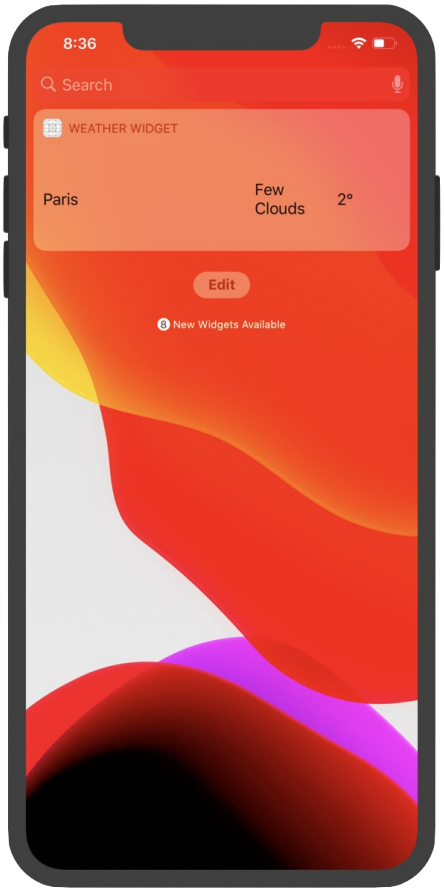


Figure 21.18. Weather widget in simulator

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 `UserDefaults`. 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 *Signing & Capabilities* tab. Click the "+ Capability" button and choose "App Groups". Note that you need a developer account to use this function. Next, 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.weatherappdemo`. Don't just copy & paste the name. You should use another name for your app.

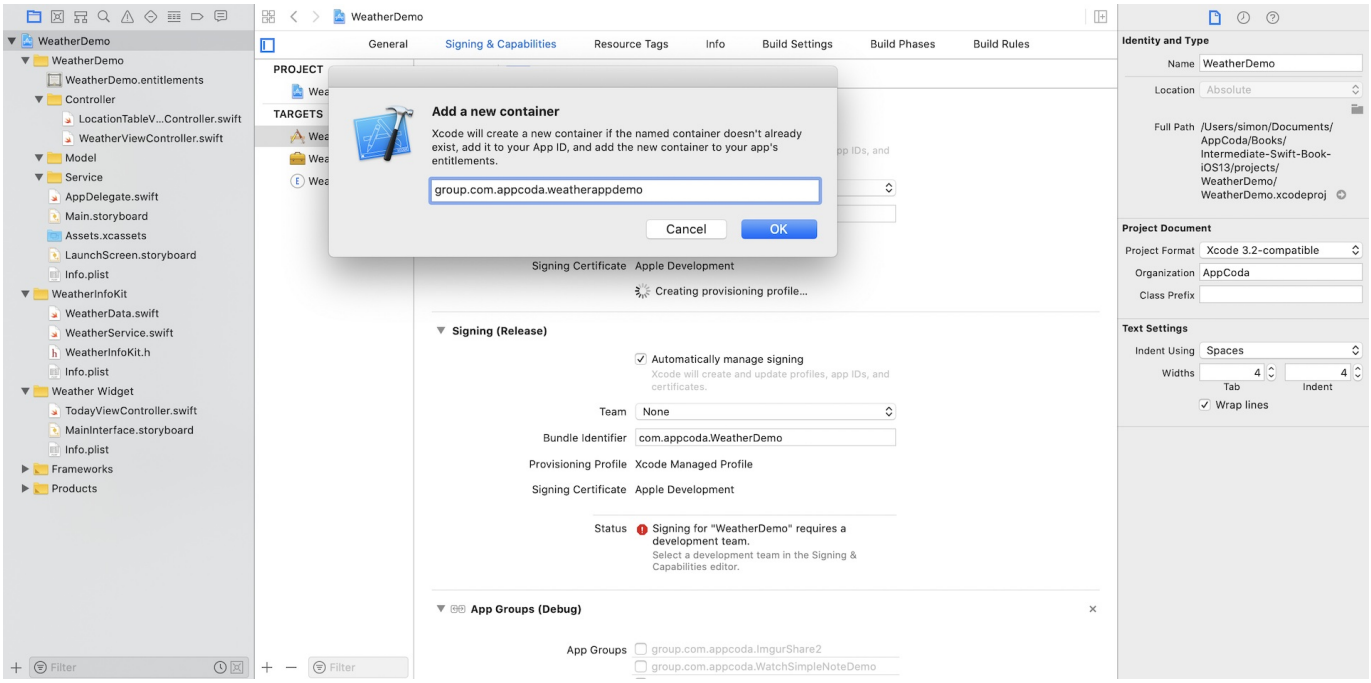


Figure 21.19. Adding a new container for App Groups

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 `UserDefaults` 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.weatherappdemo")!
```

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, didSelectRowAt indexPath: IndexP
ath) {
    let cell = tableView.cellForRow(at: indexPath)
    cell?.accessoryType = .checkmark
    if let location = cell?.textLabel?.text {
        selectedLocation = location

        defaults.setValue(selectedCity, forKey: "city")
        defaults.setValue(selectedCountry, forKey: "country")
    }

    tableView.reloadData()
}
```

We only add a couple lines of code in the `if let` block to save the selected city and country to the defaults. Next, open `TodayViewController.swift` and add the following variable:

```
var defaults = UserDefaults(suiteName: "group.com.appcoda.weatherappdemo")!
```

In the `widgetPerformUpdate(completionHandler:)` method, insert the following lines of code at the beginning:

```
// Get the location from defaults
if let currentCity = defaults.value(forKey: "city") as? String,
    let currentCountry = defaults.value(forKey: "country") as? String {

    city = currentCity
    country = currentCountry
}
```

Here we simply retrieve the city and country from `UserDefaults`, which is the location set by the user.

Also, change the following line of code from:

```
cityLabel.text = city
```

To:

```
cityLabel.text = "\(city), \(country)"
```

This is to display both city and country in the city label.

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.

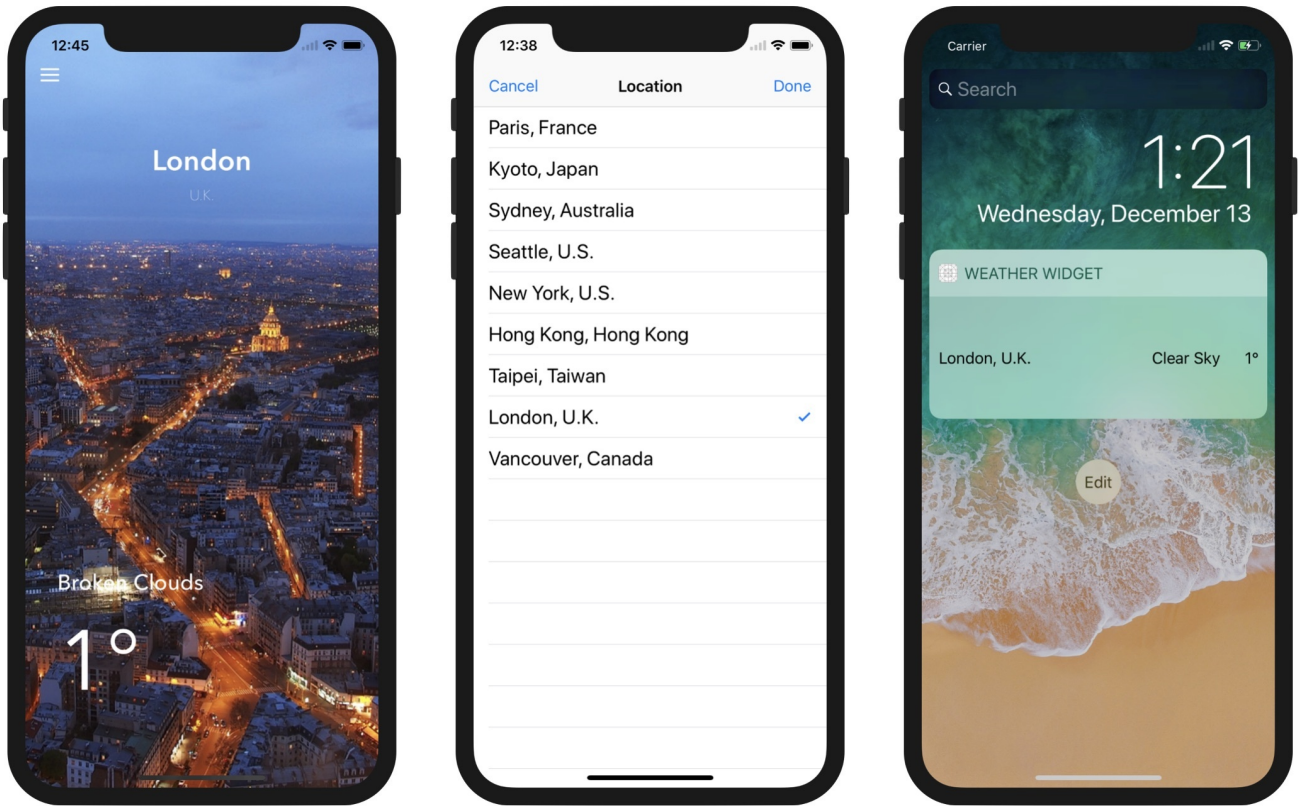


Figure 21.20. The location of the weather widget is now in-sync with that of the weather demo app

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/WeatherDemoFinal.zip.

Chapter 22
Building Slide Out Sidebar Menus and Working with Objective-C Libraries

In this chapter, I will show you how to 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.

In recent years, there were 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, 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. This is also one of the reasons I chose this library. 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 the 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 the 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.

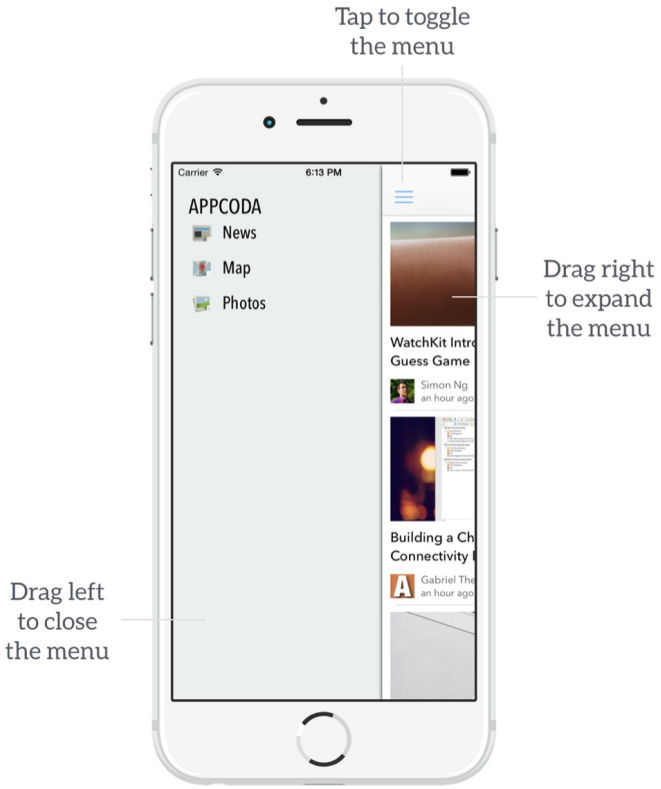


Figure 22.2. The demo app

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 http://www.appcoda.com/resources/swift5/SidebarMenuStarter.zip.

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.

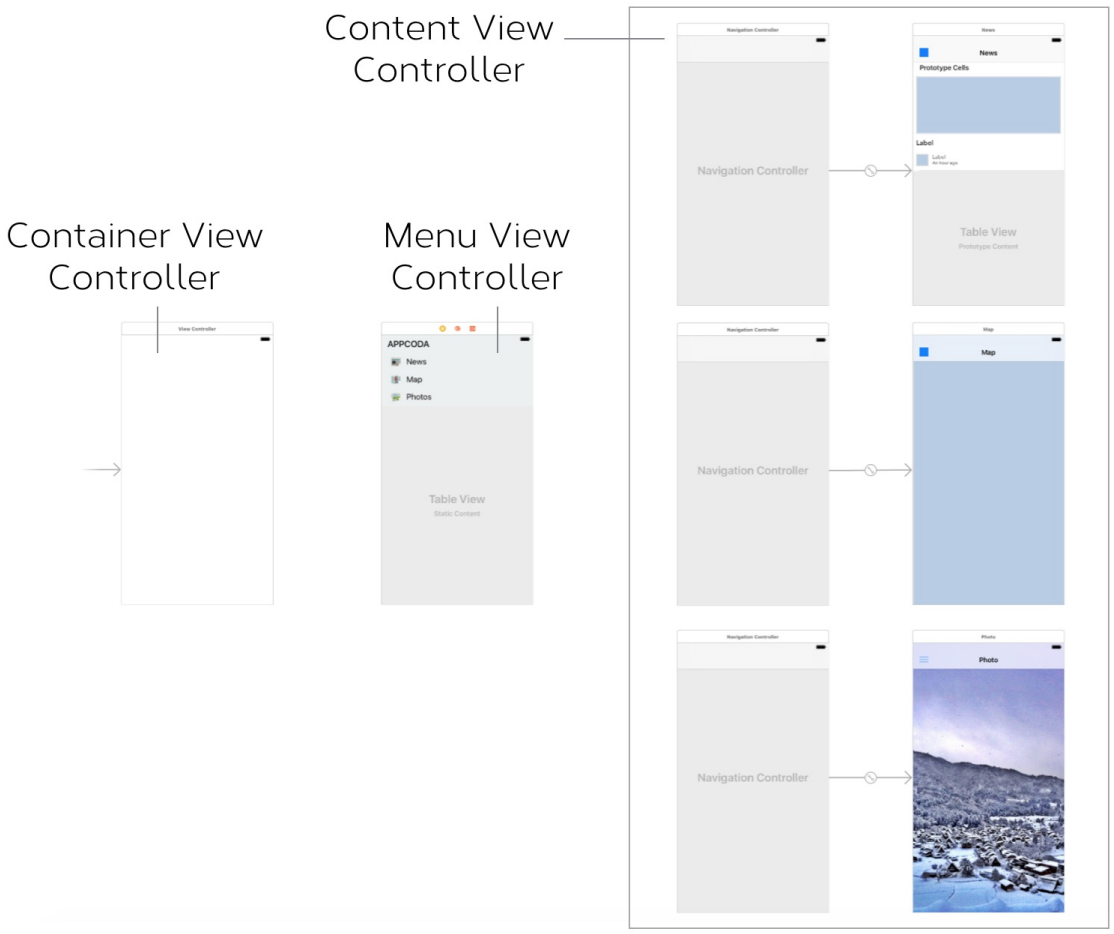


Figure 22.2. A sample storyboard for using SWRevealViewController

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 icons).

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.

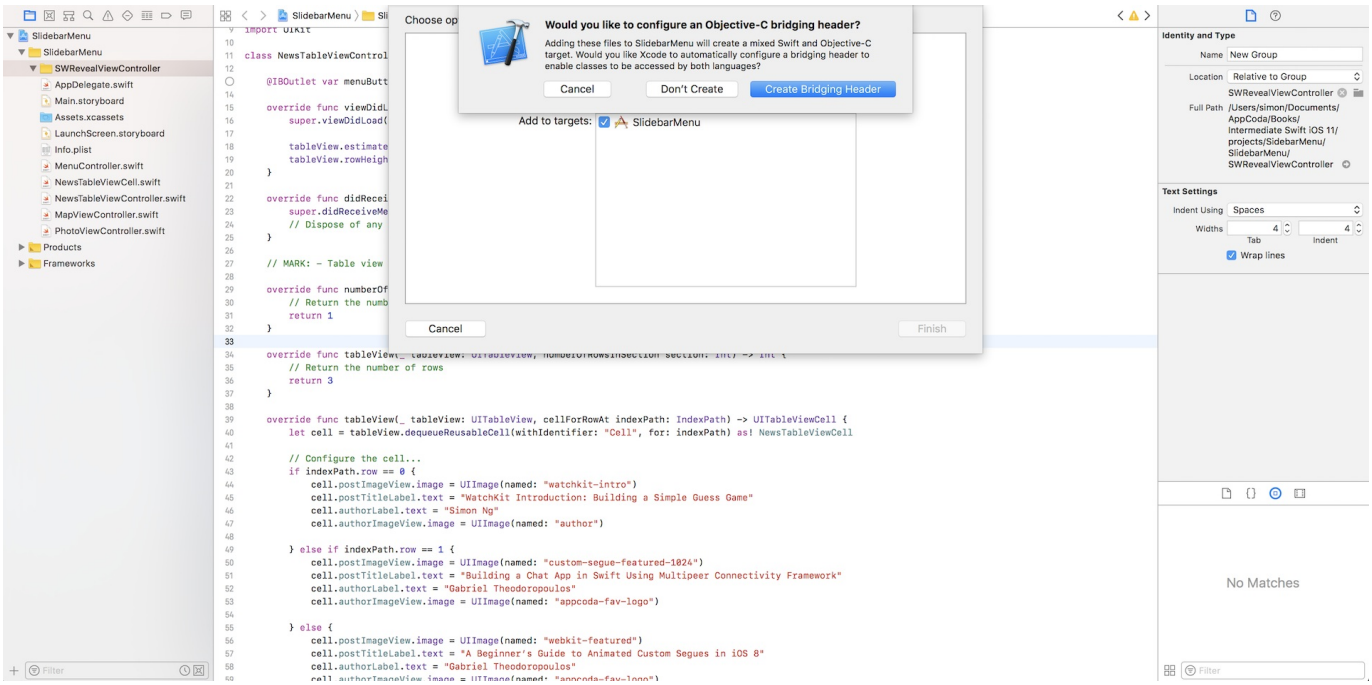


Figure 22.3. Adding the SWRevealViewController files to the project

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`.

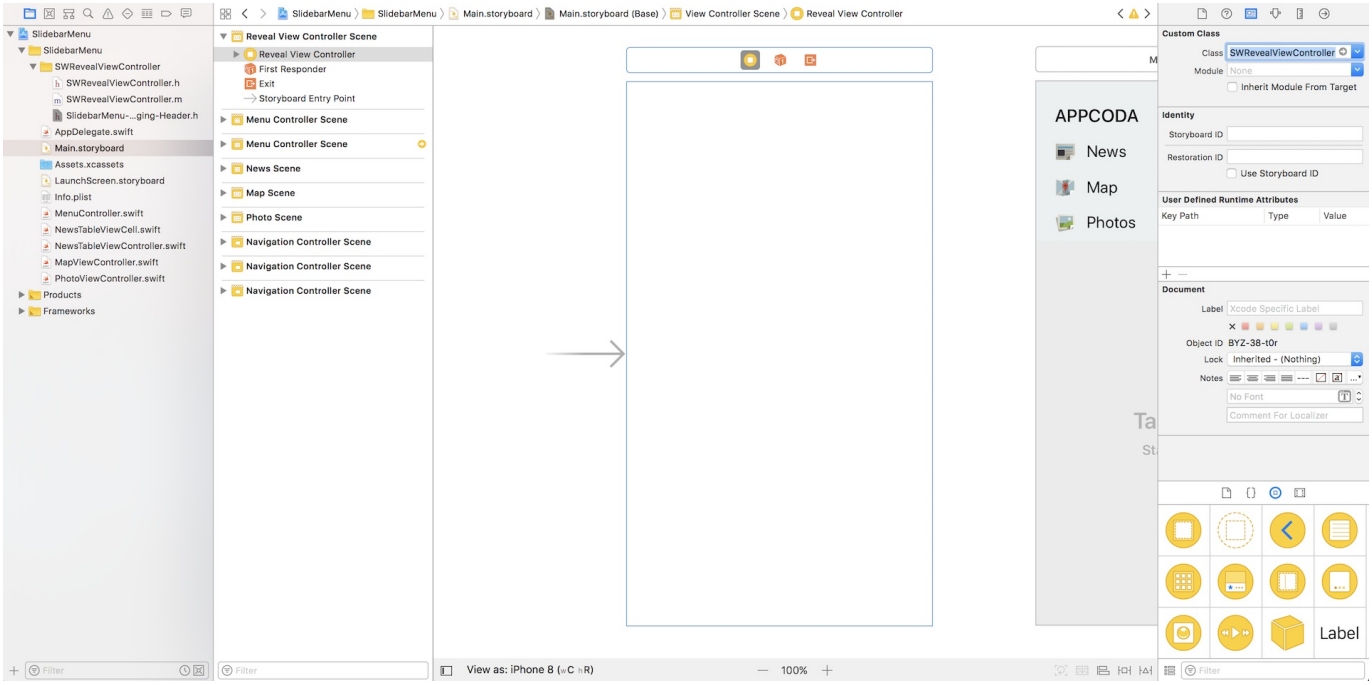


Figure 22.4. Using SWRevealViewController in storyboard

Next, control-drag from the Reveal View Controller to the Menu Controller. After releasing the button, you will see a context menu for segue selection. In this case, select `reveal view controller set segue`.

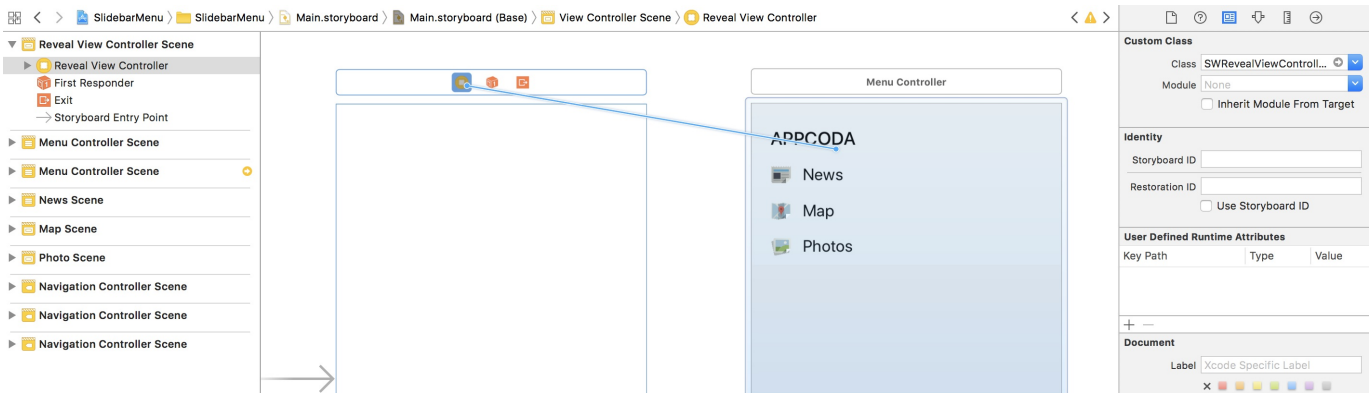


Figure 22.5. Control-drag from the Reveal View Controller to the Menu Controller

This defines a custom segue known as `SWRevealViewControllerSetSegue`. Select this segue and change its identifier to `sw_rear` under the Identity inspector. By setting the identifier, you tell `SWRevealViewController` that the menu 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.

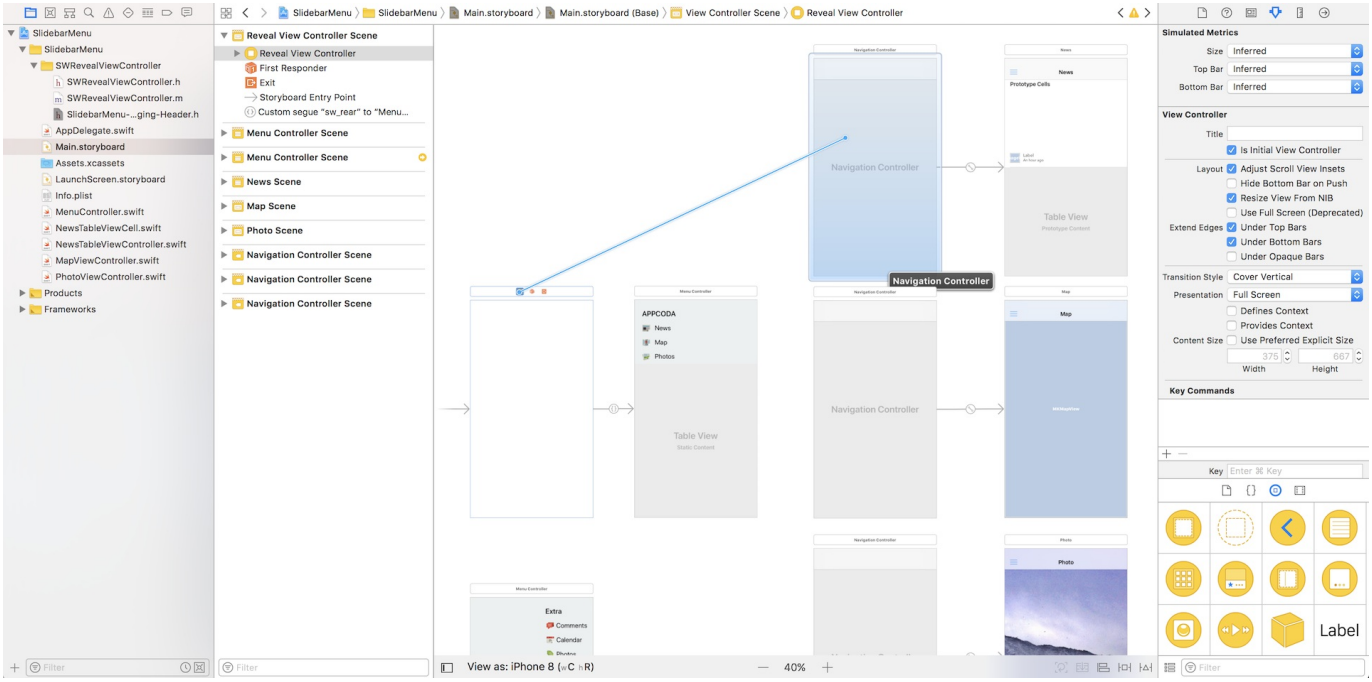


Figure 22.6. Connecting the Reveal View Controller with the navigation controller

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. However, 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 an action method yet.

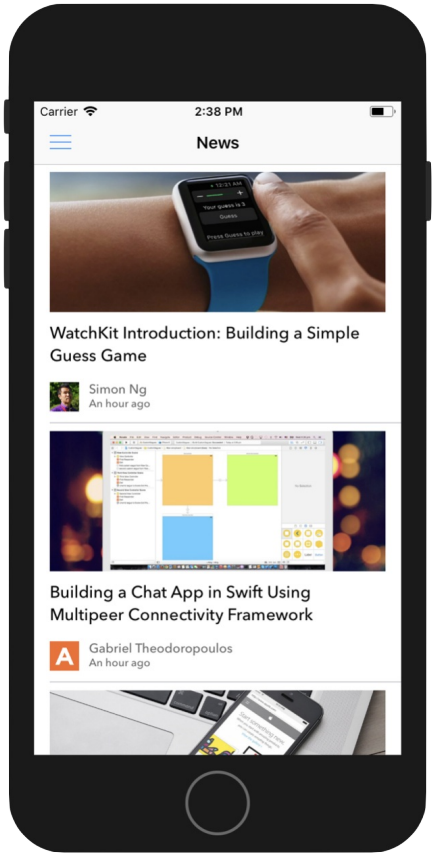


Figure 22.7. Running the demo app will show the News view

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 = #selector(SWRevealViewController.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 need to specify the method name using the `#selector` syntax to construct a selector.

Lastly, we add a gesture recognizer. Not only can you 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 gestures. Try to swipe right in the content area and see what you get.

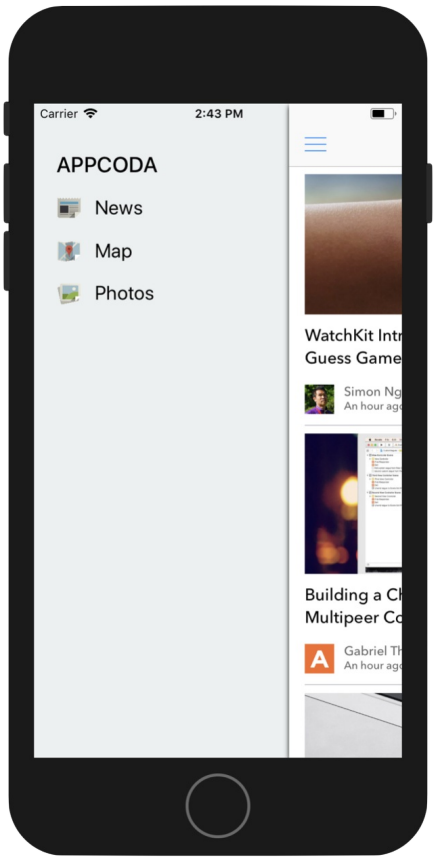


Figure 22.8. Tapping the menu button now shows the sidebar menu

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.

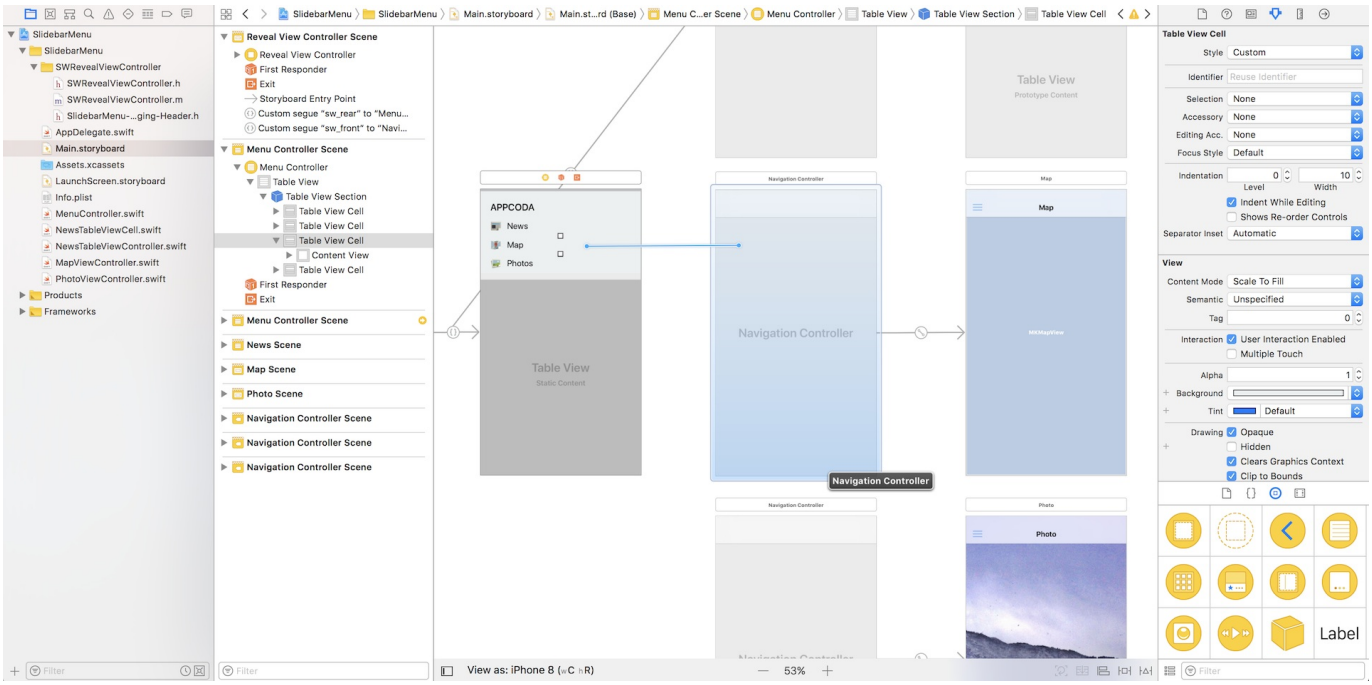


Figure 22.9. Connecting the map item with the navigation controller associated with the map view

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 = #selector(SWRevealViewController.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.

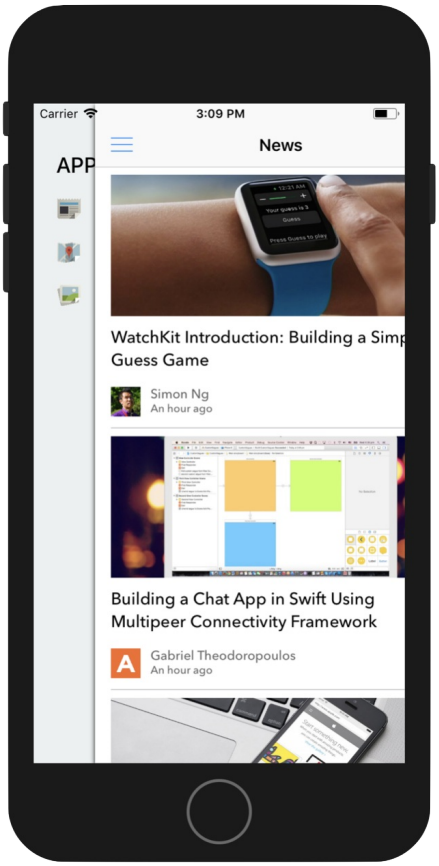


Figure 22.10. Resizing the sidebar menu

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 done. 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 slid from right.

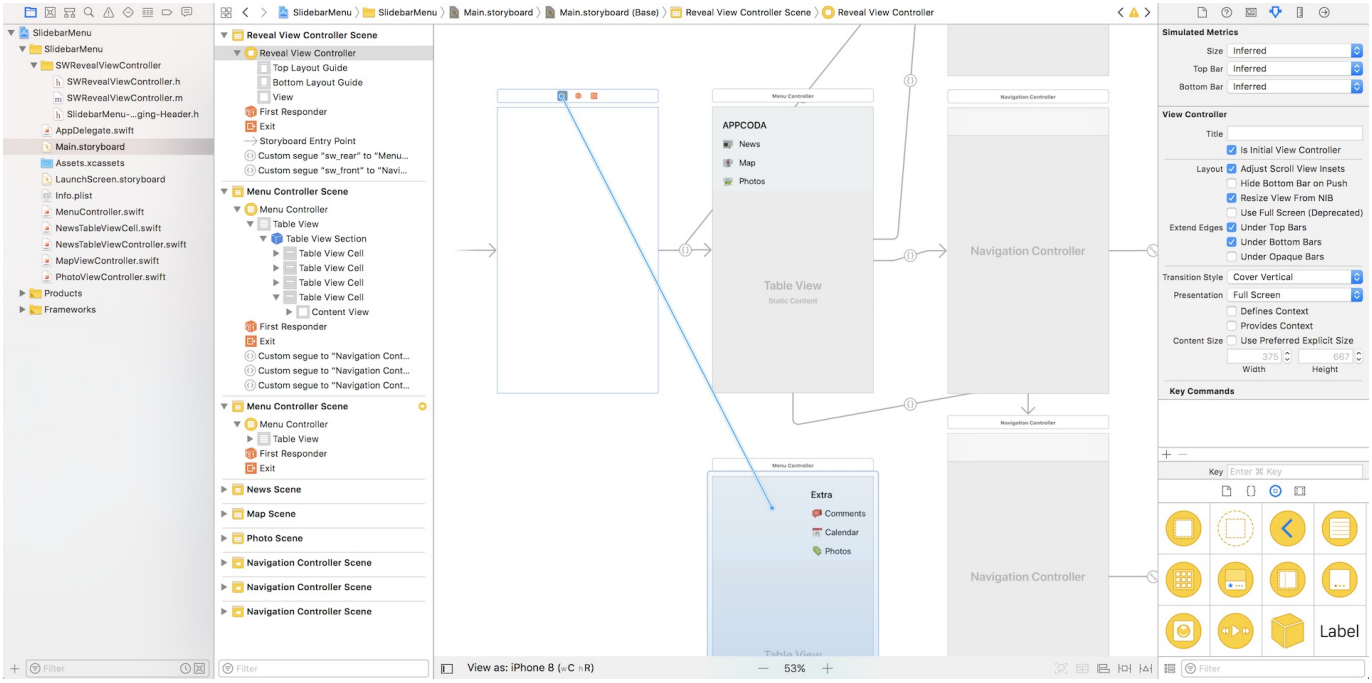


Figure 22.11. Associate the Reveal View Controller with the extra menu controller

Now drag a bar button item to the News view controller, and place it on the right side of the navigation bar. In the Identity inspector, set the *System Item* option to `Organize`.

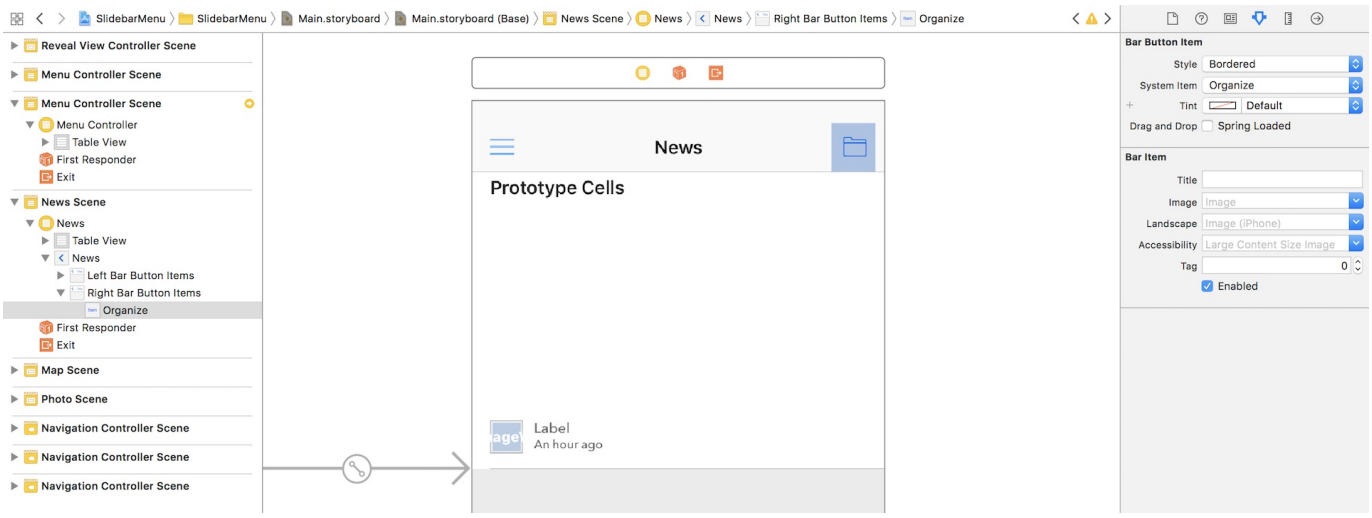


Figure 22.12. Adding an organize button to the navigation bar

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 = #selector(SWRevealViewController.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 to the outlet variable. That's it! Run the project again. Now the app has a right sidebar.

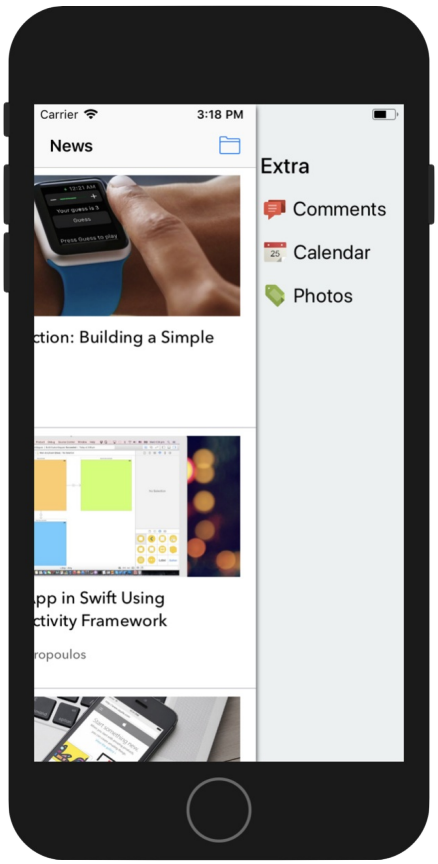


Figure 22.13. Tapping the Organize button shows the right sidebar

Refactoring the Code with Swift Extensions

So far the code works great. But if you look closely at the code, you will find that the code snippet below is duplicated in multiple view controllers:

```
if revealViewController() != nil {
    menuButton.target = revealViewController()
    menuButton.action = #selector(SWRevealViewController.revealToggle(_:))

    revealViewController().rightViewRevealWidth = 150
    extraButton.target = revealViewController()
    extraButton.action = #selector(SWRevealViewController.rightRevealToggle(_:))

view.addGestureRecognizer(self.revealViewController().panGestureRecognizer())
}
```

If you're asked to add a new menu item and a new set of content view controllers, you will probably copy & paste the above code snippet to the new view controller.

The code will work as usual, but this is not a very good programming practice. Programming is not static. You always need to make changes due to the change of requirements or feature enhancement.

Consider the above code snippet, what if you need to change the value of the `rightViewRevealWidth` property from `150` to `100`? You will have to modify the code in all the view controller classes that make use of the code snippet and update the value one by one. To be more exact, you will need to update the code in `NewsTableViewController`, `MapViewController` and `PhotoViewController`.

For ease of code management, we use to group reusable code in a common function. There are multiple ways to implement that. For example, you can create a super view controller with a common method of the above code snippet. Every view controller then extends from this super view controller. When you need to create a sidebar menu, you call the common method.

This is a viable solution, but there is a simpler way to do that. Swift provides a feature called *extensions* that allows developers to add functionalities to an existing class or structure. You declare an extension using the `extension` keyword. For example, to extend the functionality of `UIButton`, you write the code like this:

```
extension UIButton {
  // new functions go here
}
```

In the demo project, all the view controller classes extend from `UIViewController`. So we can extend its functionality to offer the sidebar menu.

> **Note:** Please note that UITableViewController also extends from UIViewController.

Let's first create a Swift file in the project. Right click `SlidebarMenu` and choose `New File...`. Select Swift File and name it `SidebarMenu.swift`. After Xcode creates the file, update the content to the following:

```
import UIKit

extension UIViewController {

    func addSideBarMenu(leftMenuButton: UIBarButtonItem?, rightMenuButton: UIBarBu
ttonItem? = nil) {
        if revealViewController() != nil {

            if let menuButton = leftMenuButton {
                menuButton.target = revealViewController()
                menuButton.action = #selector(SWRevealViewController.revealToggle(_
:))
            }

            if let extraButton = rightMenuButton {
                revealViewController().rightViewRevealWidth = 150
                extraButton.target = revealViewController()
                extraButton.action = #selector(SWRevealViewController.rightRevealT
oggle(_:))
            }

            view.addGestureRecognizer(self.revealViewController().panGestureRecogn
izer())
        }
    }
}
```

Here we extend the functionality of `UIViewController` by adding a new method called `addSideBarMenu`. The method accepts two parameters: `leftMenuButton` and `rightMenuButton`. The body of the method is very similar to the code snippet we used before, so I will not go into that again. We just group the reusuable code in this common method, and make it available for any classes that extends from `UIViewController`.

With this common method, we can now modify the `viewDidLoad` method of `NewsTableViewController`, `MapViewController`, and `PhotoViewController`:

NewsTableViewController:

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

    tableView.estimatedRowHeight = 242.0
    tableView.rowHeight = UITableViewAutomaticDimension

    addSideBarMenu(leftMenuButton: menuButton, rightMenuButton: extraButton)
}
```

MapViewController:

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

    addSideBarMenu(leftMenuButton: menuButton)
}
```

PhotoViewController:

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

    addSideBarMenu(leftMenuButton: menuButton)
}
```

As all these controllers extend from `UIViewController`, they automatically enjoy the functionality we added to the `UIViewController` class. Instead of scattering the same code snippet in multiple classes, we centralize it in a single file.

The code now looks much cleaner and is easier to maintain. In case we need to make any changes to the sidebar menu, we only need to modify the method `addSideBarMenu`.

It is not easy to get the code right the first time. Remember to refactor your code to make it better and better.

For reference, you can download the final project from http://www.appcoda.com/resources/swift5/SidebarMenu.zip.

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 http://www.appcoda.com/resources/swift5/NavTransitionStarter.zip. 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.

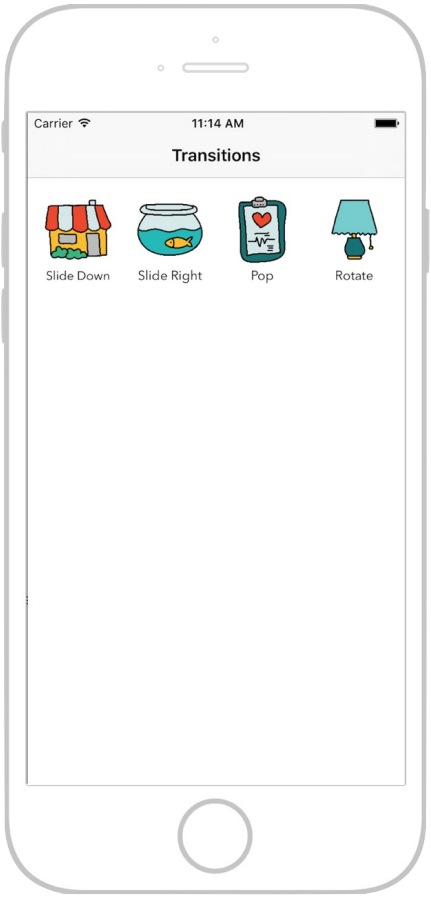


Figure 23.1. The demo app

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.

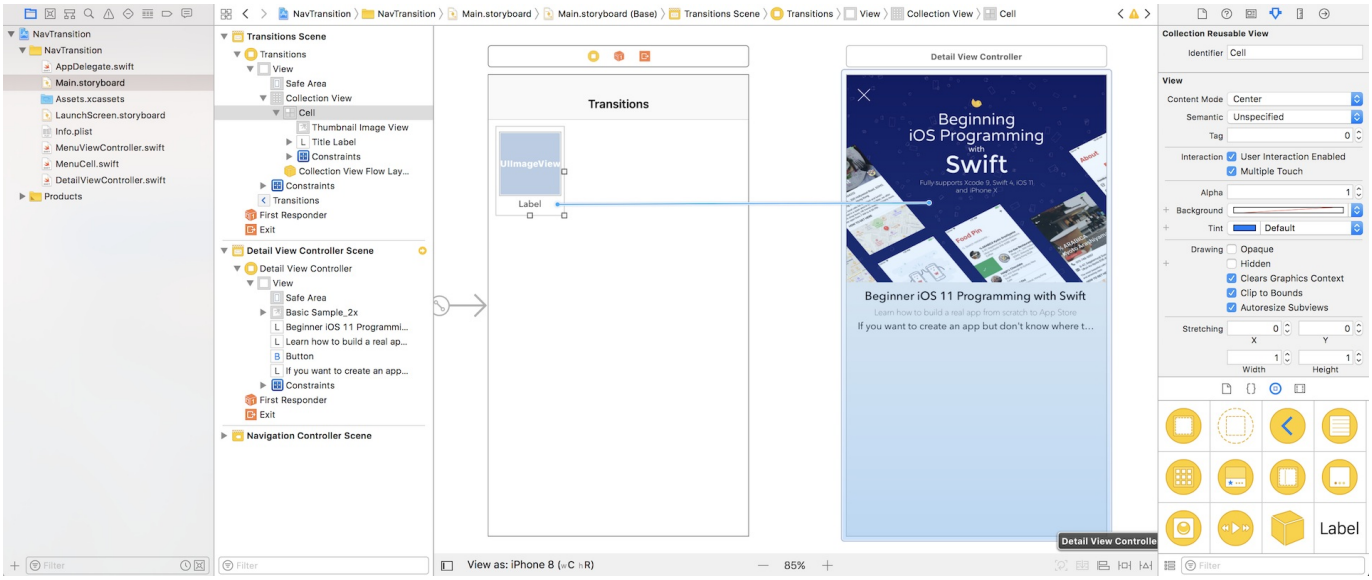


Figure 23.2. Connecting the collection view cell with the detail view controller

If you run the demo app again, tapping any of the items 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 Swift file. Name the class `SlideDownTransitionAnimator`.

As mentioned earlier, the animator object should adopt both the `UIViewControllerAnimatedTransitioning` and the `UIViewControllerTransitioningDelegate` protocols. So update the file content like this:

```
import UIKit

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 animationController(forPresented presented: UIViewController, presenting: UIV
iewController, source: UIViewController) -> UIViewControllerAnimatedTransitioning?
 {

    return self
}

func animationController(forDismissed dismissed: UIViewController) -> UIViewContro
llerAnimatedTransitioning? {

    return self
}
```

The `animationController(forPresented:presenting:source:)` method returns the animator object to use when presenting the view controller. Here we return `self`, as the `SlideDownTransitionAnimator` object is the animator.

The `animationController(forDismissed:)` 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(using:)`

- `animateTransition(using:)`

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(using:)` 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.

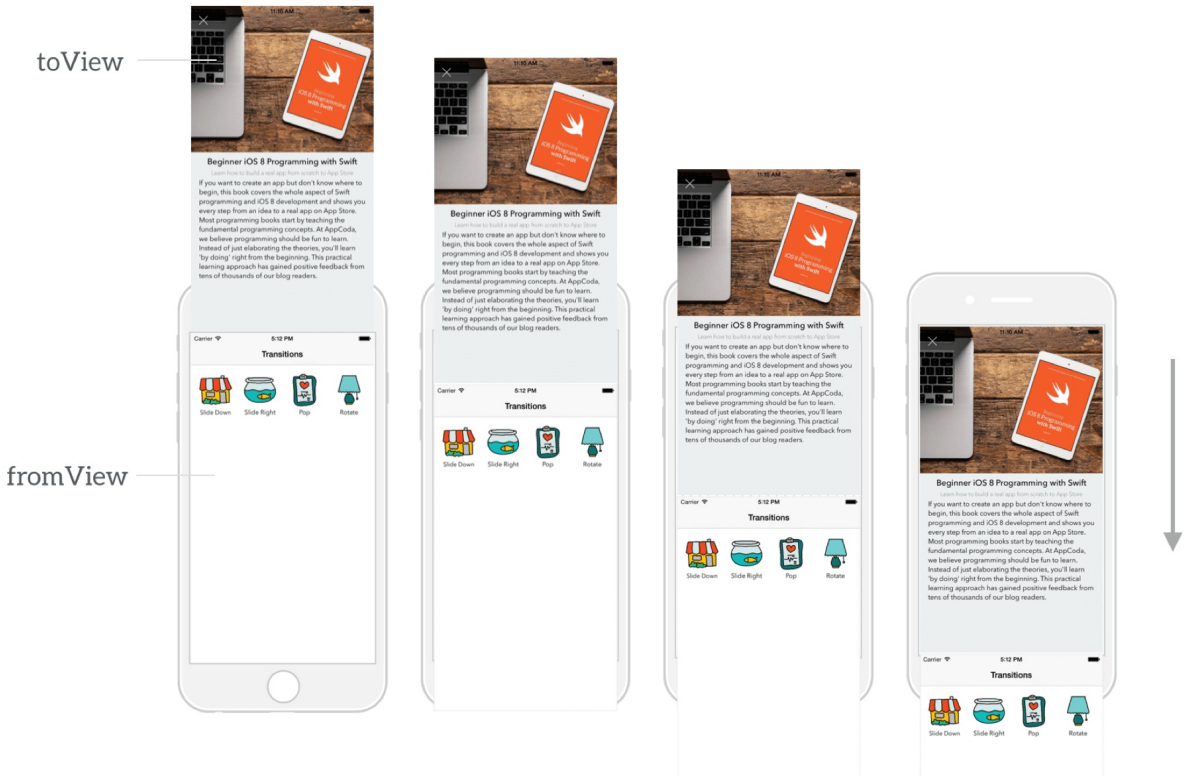


Figure 23.3. How the slidedown animation works

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(using transitionContext: UIViewControllerContextTransitioning?) -> TimeInterval {

    return duration
}

func animateTransition(using transitionContext: UIViewControllerContextTransitioning) {

    // Get reference to our fromView, toView and the container view
    guard let fromView = transitionContext.view(forKey: UITransitionContextViewKey.from) else {
        return
    }

    guard let toView = transitionContext.view(forKey: UITransitionContextViewKey.to) else {
        return
    }

    // Set up the transform we'll use in the animation
    let container = transitionContext.containerView

    let offScreenUp = CGAffineTransform(translationX: 0, y: -container.frame.height)
    let offScreenDown = CGAffineTransform(translationX: 0, y: 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.animate(withDuration: duration, delay: 0.0, usingSpringWithDamping: 0.8
, initialSpringVelocity: 0.8, options: [], animations: {

        fromView.transform = offScreenDown
        fromView.alpha = 0.5
        toView.transform = CGAffineTransform.identity
        toView.alpha = 1.0

    }, 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(using:)` 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 `animate` method of `UIView` to perform a spring animation:

```
    // Perform the animation
    UIView.animate(withDuration: duration, delay: 0.0, usingSpringWithDamping: 0.8
, initialSpringVelocity: 0.8, options: [], animations: {

        fromView.transform = offScreenDown
        fromView.alpha = 0.5
        toView.transform = CGAffineTransform.identity
        toView.alpha = 1.0

    }, completion: { finished in

        transitionContext.completeTransition(true)

    })
```

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 & alpha value, 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 `MenuViewController.swift` file, declare the following variable to hold the animator object:

```
let slideDownTransition = SlideDownTransitionAnimator()
```

Next, implement the `prepare(for:sender:)` method:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let toViewController = segue.destination
    let sourceViewController = segue.source as! MenuViewController

    toViewController.modalPresentationStyle = .fullScreen
    sourceViewController.modalPresentationStyle = .fullScreen

    if let selectedIndexPaths = sourceViewController.collectionView.indexPathsForS
electedItems {
        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.

In iOS 13, Apple introduced a new default card presentation (https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/modality/) when presenting a modal view. To make the transition work, we have to set the `modalPresentationStyle` property back to `.fullScreen`.

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.

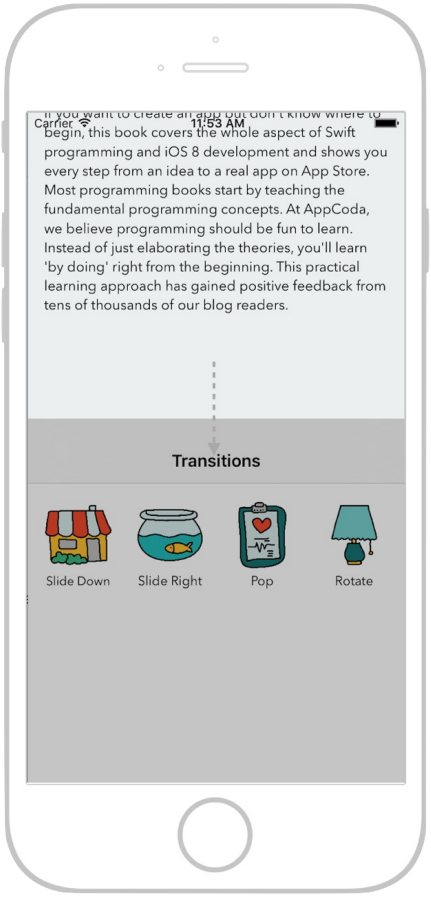


Figure 23.4. Tapping the slide down icon will transit to the detail view with a slide-down animation

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 `animationController(forDismissed:)` and `animationController(forPresented:presenting:source:)` methods like this:

```
func animationController(forPresented presented: UIViewController, presenting: UIV
iewController, source: UIViewController) -> UIViewControllerAnimatedTransitioning?
 {

    isPresenting = true
    return self
}

func animationController(forDismissed dismissed: UIViewController) -> UIViewContro
llerAnimatedTransitioning? {

    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(using transitionContext: UIViewControllerContextTransitioni
ng) {

    // Get reference to our fromView, toView and the container view
    guard let fromView = transitionContext.view(forKey: UITransitionContextViewKey
.from) else {
        return
    }

    guard let toView = transitionContext.view(forKey: UITransitionContextViewKey.t
o) else {
        return
    }
```

```
    // Set up the transform we'll use in the animation
    let container = transitionContext.containerView

    let offScreenUp = CGAffineTransform(translationX: 0, y: -container.frame.heigh
t)
    let offScreenDown = CGAffineTransform(translationX: 0, y: container.frame.heig
ht)

    // 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.animate(withDuration: duration, delay: 0.0, usingSpringWithDamping: 0.8
, initialSpringVelocity: 0.8, options: [], animations: {

        if self.isPresenting {
            fromView.transform = offScreenDown
            fromView.alpha = 0.5
            toView.transform = CGAffineTransform.identity
        } else {
            fromView.transform = offScreenUp
            fromView.alpha = 1.0
            toView.transform = CGAffineTransform.identity
            toView.alpha = 1.0
        }

    }, completion: { finished in

        transitionContext.completeTransition(true)

    })
}
```

For the reverse transition, `fromView` and `toView` are reversed. In other words, the detail view is now `fromView`, while the main view becomes `toView`.

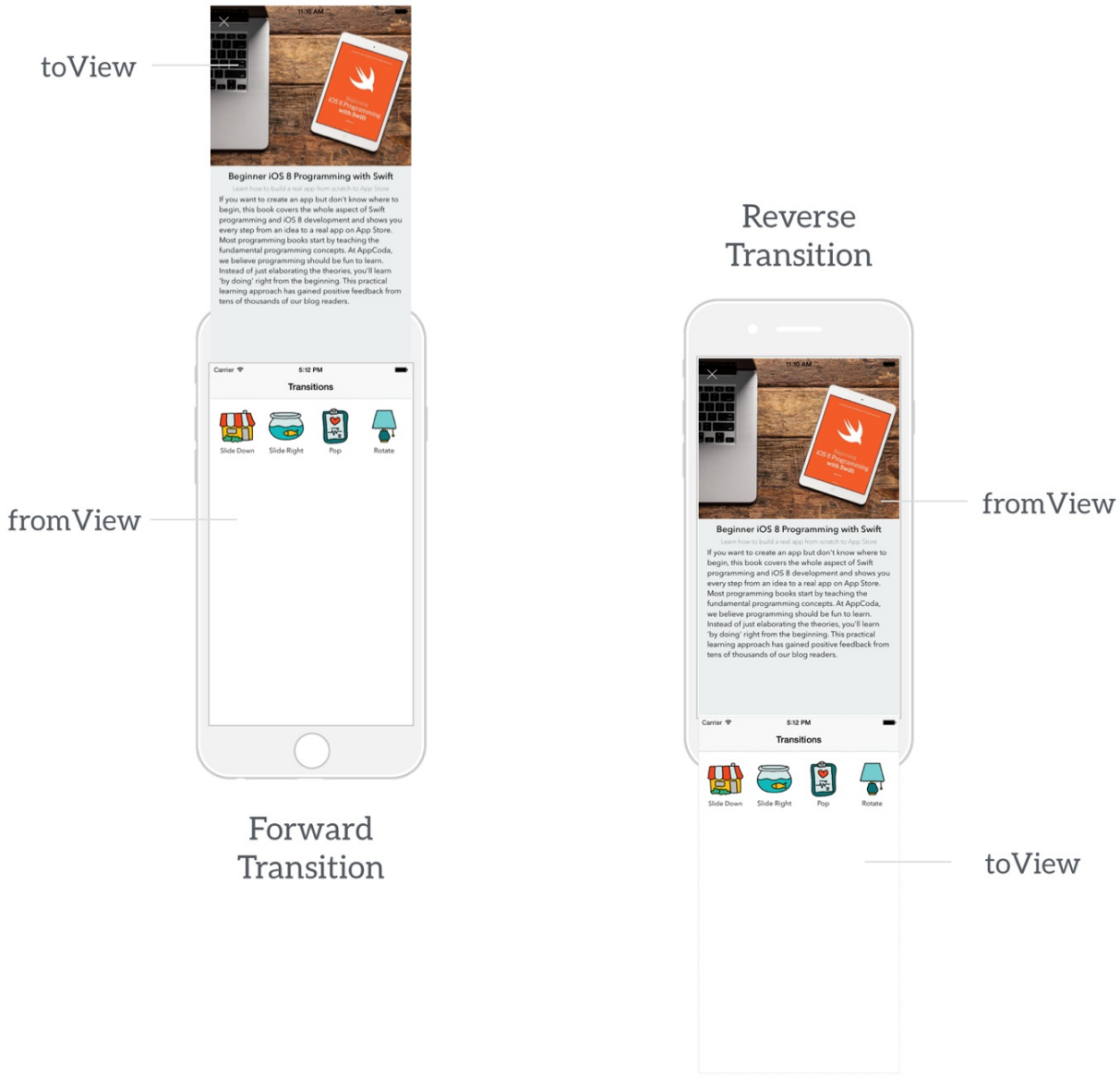
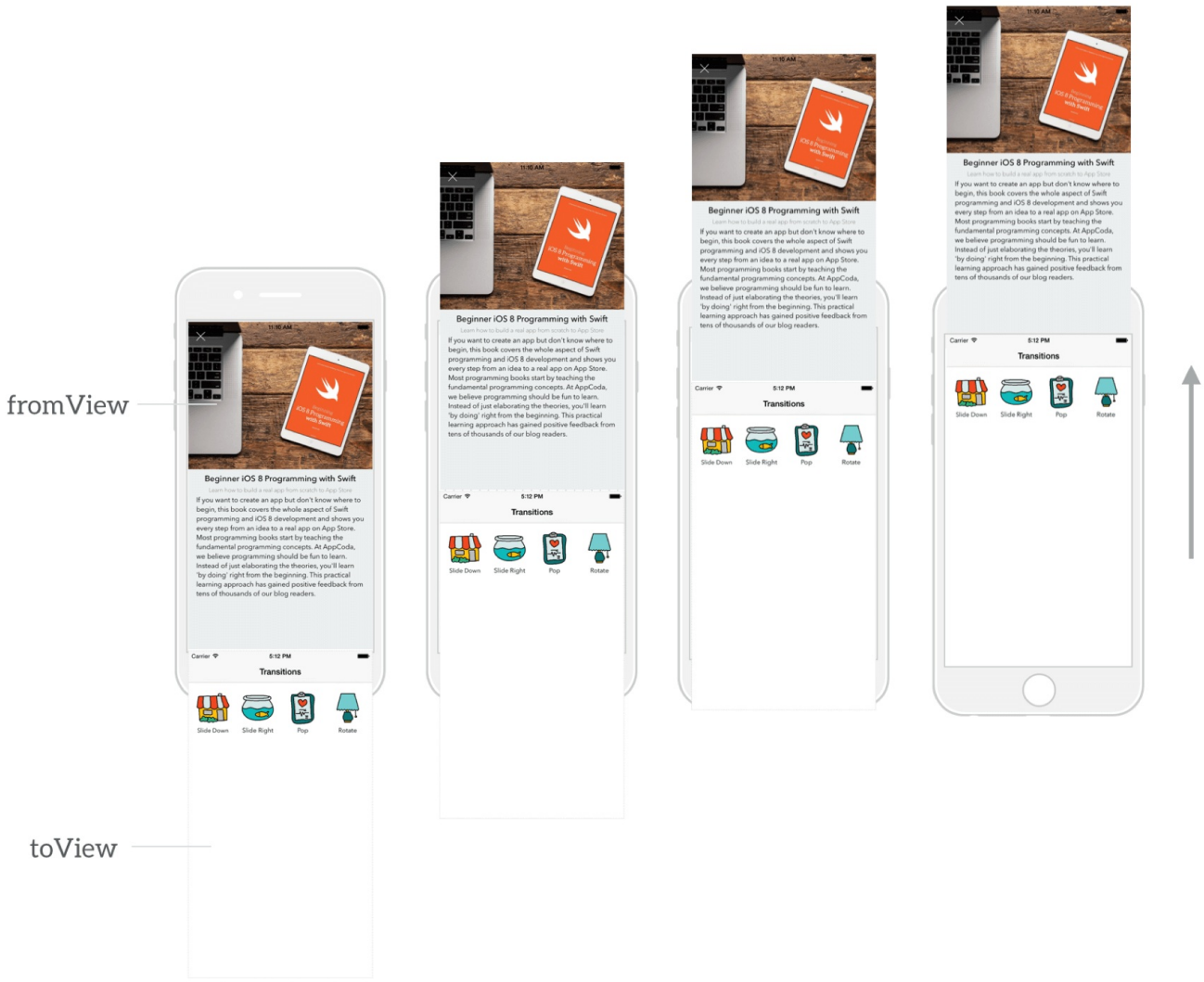


Figure 23.5. Forward and reverse transitions

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.



Figuer 23.6. Reverse transition now works as expected

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:

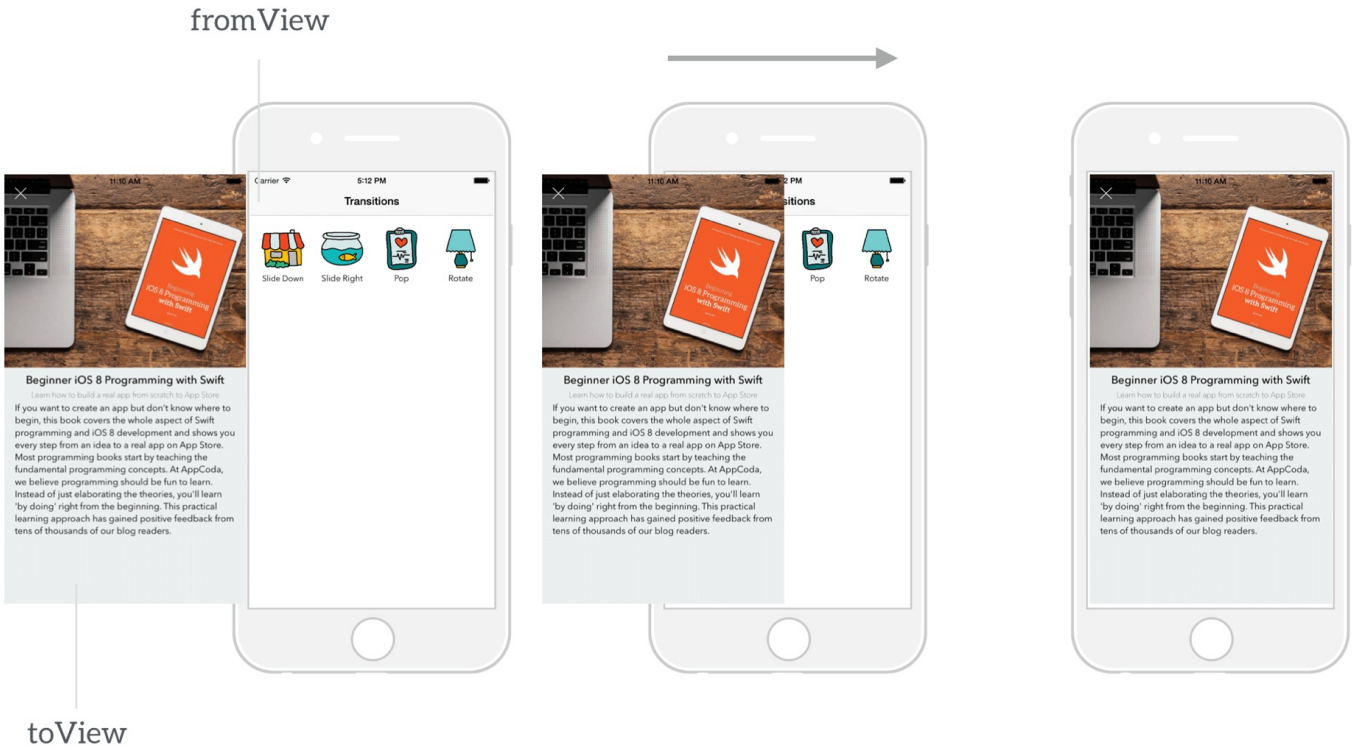


Figure 23.7. How the slide right transition works

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 Swift file named `SlideRightTransitionAnimator` with the following content:

```
import UIKit

class SlideRightTransitionAnimator: NSObject, UIViewControllerAnimatedTransitioning
, UIViewControllerTransitioningDelegate {

    let duration = 0.5
```

```
    var isPresenting = false

    func transitionDuration(using transitionContext: UIViewControllerContextTransitioning?) -> TimeInterval {
        return duration
    }

    func animateTransition(using transitionContext: UIViewControllerContextTransitioning) {

        // Get reference to our fromView, toView and the container view
        guard let fromView = transitionContext.view(forKey: UITransitionContextViewKey.from) else {
            return
        }

        guard let toView = transitionContext.view(forKey: UITransitionContextViewKey.to) else {
            return
        }

        // Set up the transform we'll use in the animation
        let container = transitionContext.containerView

        let offScreenLeft = CGAffineTransform(translationX: -container.frame.width, y: 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.animate(withDuration: duration, delay: 0.0, usingSpringWithDamping:
0.8, initialSpringVelocity: 0.8, options: [], animations: {

            if self.isPresenting {
                toView.transform = CGAffineTransform.identity
            } else {
                fromView.transform = offScreenLeft
            }

        }, completion: { finished in

            transitionContext.completeTransition(true)

        })
    }

    func animationController(forPresented presented: UIViewController, presenting:
 UIViewController, source: UIViewController) -> UIViewControllerAnimatedTransition
ing? {

        isPresenting = true
        return self
    }

    func animationController(forDismissed dismissed: UIViewController) -> UIViewCo
ntrollerAnimatedTransitioning? {

        isPresenting = false
        return self
    }

}
```

The code is very similar to the one we developed previously.

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 `CGAffineTransform.identity` 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 `MenuViewController.swift` file and declare the `slideRightTransition` variable:

```
let slideRightTransition = SlideRightTransitionAnimator()
```

Change the `prepare(for:sender:)` 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.

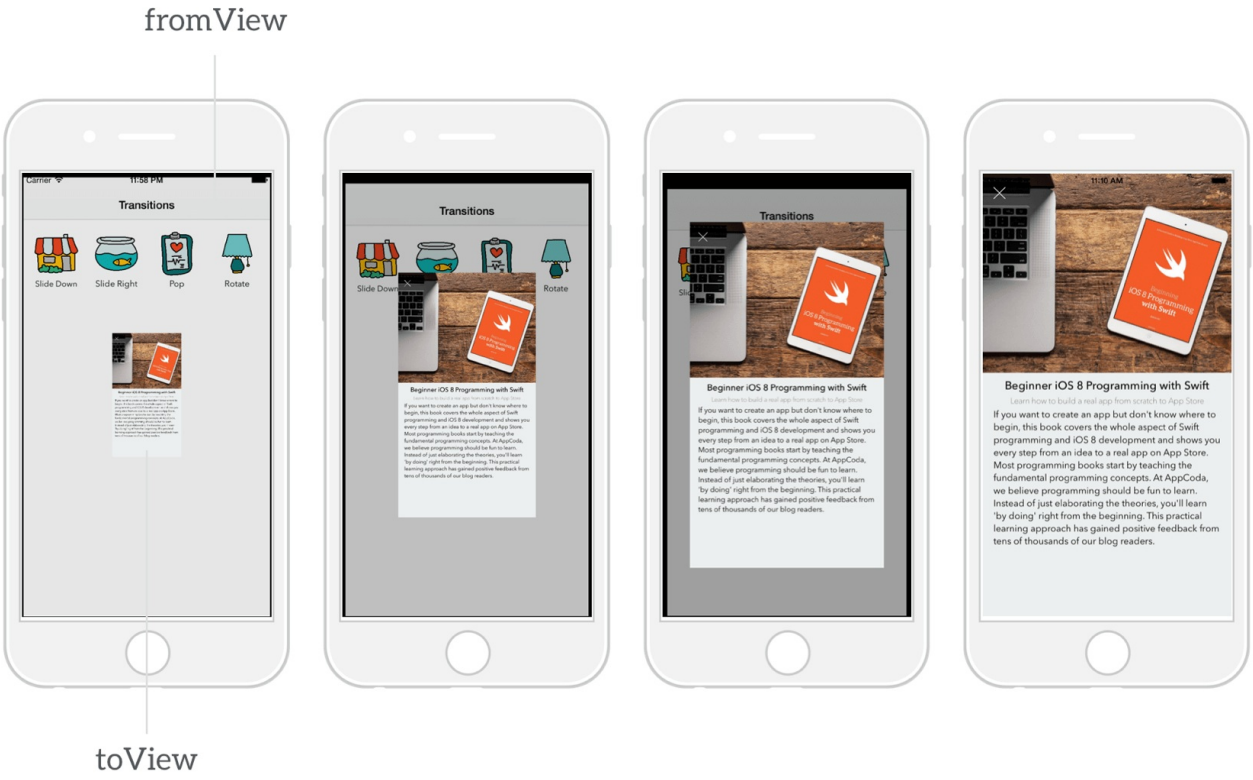


Figure 23.8. How the pop transition works

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 Swift file and name it `PopTransitionAnimator`. Make sure you import the `UIKit` framework and implement the class like this:

```
import UIKit

class PopTransitionAnimator: NSObject, UIViewControllerAnimatedTransitioning, UIVi
ewControllerTransitioningDelegate {

    let duration = 0.5
    var isPresenting = false

    func transitionDuration(using transitionContext: UIViewControllerContextTransi
tioning?) -> TimeInterval {
```

```
        return duration
    }

    func animateTransition(using transitionContext: UIViewControllerContextTransit
ioning) {

        // Get reference to our fromView, toView and the container view
        guard let fromView = transitionContext.view(forKey: UITransitionContextVie
wKey.from) else {
            return
        }

        guard let toView = transitionContext.view(forKey: UITransitionContextViewK
ey.to) else {
            return
        }

        // Set up the transform we'll use in the animation
        let container = transitionContext.containerView

        let minimize = CGAffineTransform(scaleX: 0, y: 0)
        let offScreenDown = CGAffineTransform(translationX: 0, y: container.frame.
height)
        let shiftDown = CGAffineTransform(translationX: 0, y: 15)
        let scaleDown = shiftDown.scaledBy(x: 0.95, y: 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.animate(withDuration: duration, delay: 0.0, usingSpringWithDamping:
0.8, initialSpringVelocity: 0.8, options: [], animations: {
```

```
            if self.isPresenting {
                fromView.transform = scaleDown
                fromView.alpha = 0.5
                toView.transform = CGAffineTransform.identity
            } else {
                fromView.transform = offScreenDown
                toView.alpha = 1.0
                toView.transform = CGAffineTransform.identity
            }

        }, completion: { finished in

            transitionContext.completeTransition(true)

        })
    }

    func animationController(forPresented presented: UIViewController, presenting:
 UIViewController, source: UIViewController) -> UIViewControllerAnimatedTransition
ing? {

        isPresenting = true
        return self
    }

    func animationController(forDismissed dismissed: UIViewController) -> UIViewCo
ntrollerAnimatedTransitioning? {

        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 the following CGAffineTransform to minimize the detail view:

```
CGAffineTransform(scaleX: 0, y: 0)
```

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 `MenuViewController.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.

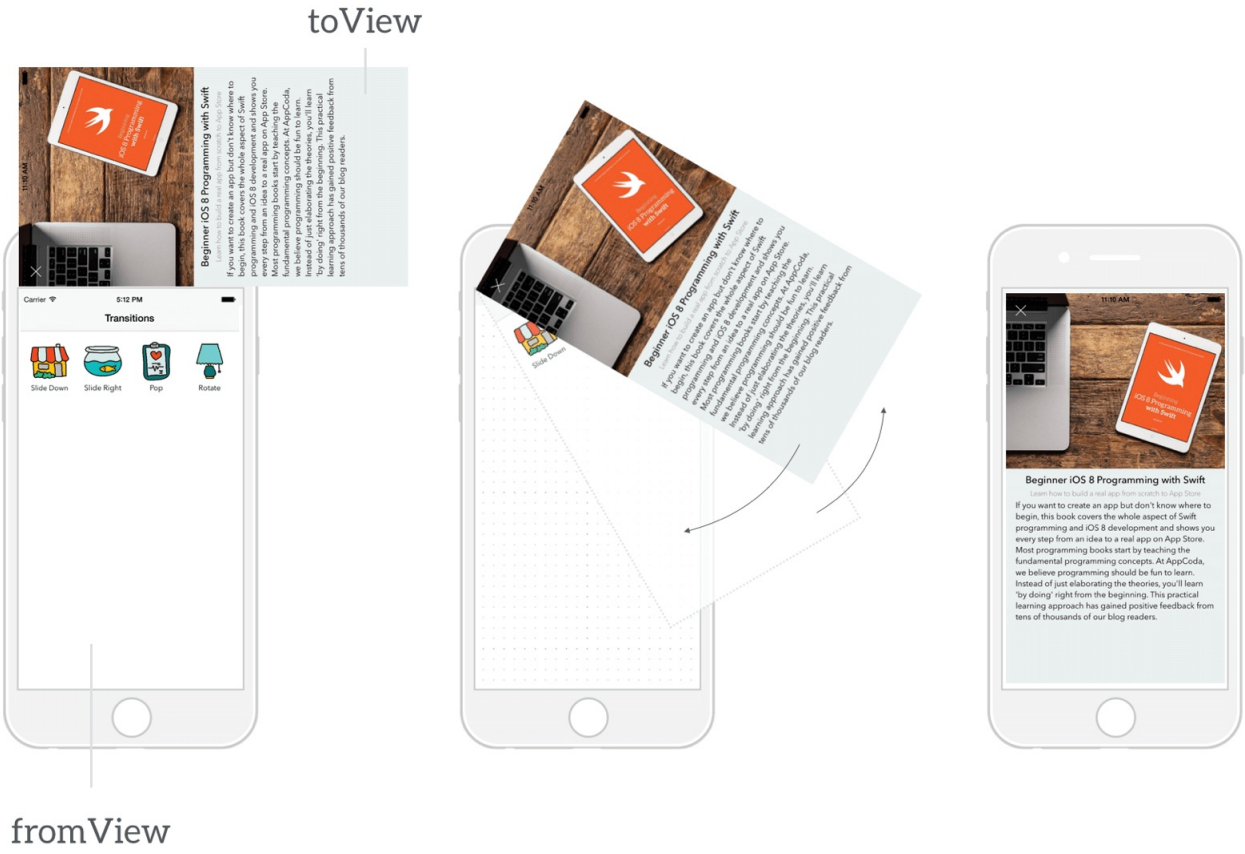


Figure 23.9. How the rotation transition works

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 Swift file called `RotateTransitionAnimator`. Implement the class like this:

```
import UIKit

class RotateTransitionAnimator: NSObject, UIViewControllerAnimatedTransitioning, U
IViewControllerTransitioningDelegate {

    let duration = 0.5
    var isPresenting = false

    func transitionDuration(using transitionContext: UIViewControllerContextTransi
```

```
tioning?) -> TimeInterval {
        return duration
    }

    func animateTransition(using transitionContext: UIViewControllerContextTransit
ioning) {

        // Get reference to our fromView, toView and the container view
        guard let fromView = transitionContext.view(forKey: UITransitionContextVie
wKey.from) else {
            return
        }

        guard let toView = transitionContext.view(forKey: UITransitionContextViewK
ey.to) else {
            return
        }

        // Set up the transform we'll use in the animation
        let container = transitionContext.containerView

        // Set up the transform for rotation
        // The angle is in radian. To convert from degree to radian, use this form
ula
        // radian = angle * pi / 180
        let rotateOut = CGAffineTransform(rotationAngle: -90 * CGFloat.pi / 180)

        // Change the anchor point and position
        toView.layer.anchorPoint = CGPoint(x:0, y:0)
        fromView.layer.anchorPoint = CGPoint(x:0, y:0)
        toView.layer.position = CGPoint(x:0, y:0)
        fromView.layer.position = CGPoint(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.animate(withDuration: duration, delay: 0.0, usingSpringWithDamping:
```

```
0.8, initialSpringVelocity: 0.8, options: [], animations: {

            if self.isPresenting {
                fromView.transform = rotateOut
                fromView.alpha = 0
                toView.transform = CGAffineTransform.identity
                toView.alpha = 1.0
            } else {
                fromView.alpha = 0
                fromView.transform = rotateOut
                toView.alpha = 1.0
                toView.transform = CGAffineTransform.identity
            }

        }, completion: { finished in

            transitionContext.completeTransition(true)

        })
    }

    func animationController(forPresented presented: UIViewController, presenting:
 UIViewController, source: UIViewController) -> UIViewControllerAnimatedTransition
ing? {

        isPresenting = true
        return self
    }

    func animationController(forDismissed dismissed: UIViewController) -> UIViewCo
ntrollerAnimatedTransitioning? {

        isPresenting = false
        return self
    }

}
```

Let's discuss the first code snippet. To build the animation, the first thing that comes to your mind is to create a rotation transform. 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 = CGAffineTransform(rotationAngle: -90 * CGFloat.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.

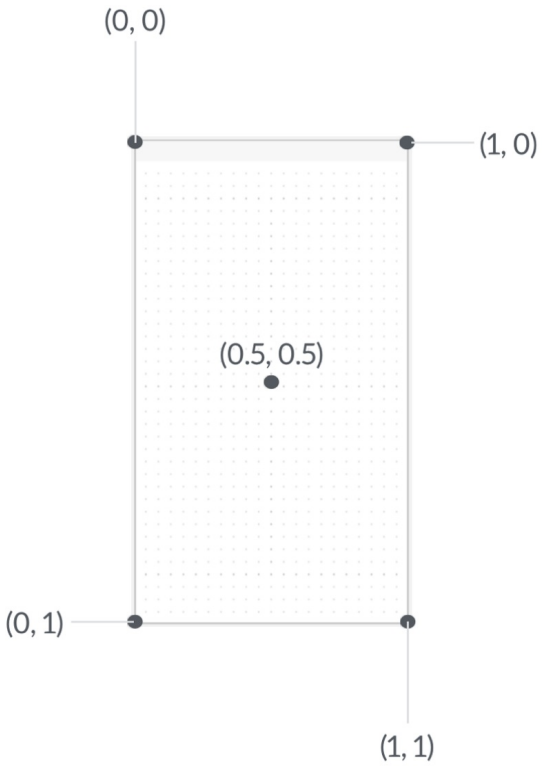


Figure 23.10. The unit coordinate space

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 = CGPoint(x:0, y:0)
fromView.layer.anchorPoint = CGPoint(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:

Figure 23.11. Understanding anchor point

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 `MenuViewController.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 http://www.appcoda.com/resources/swift5/NavTransition.zip.

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 http://www.appcoda.com/resources/swift5/SlideDownMenuStarter.zip. 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.

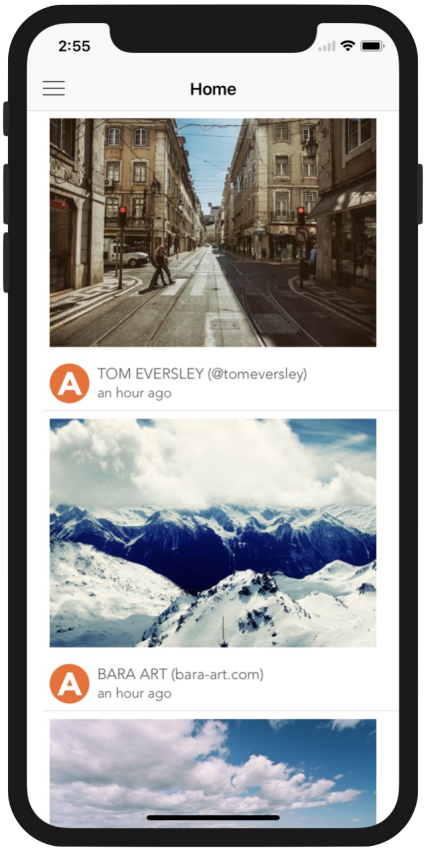


Figure 24.1. Running the starter project will give you this app

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.

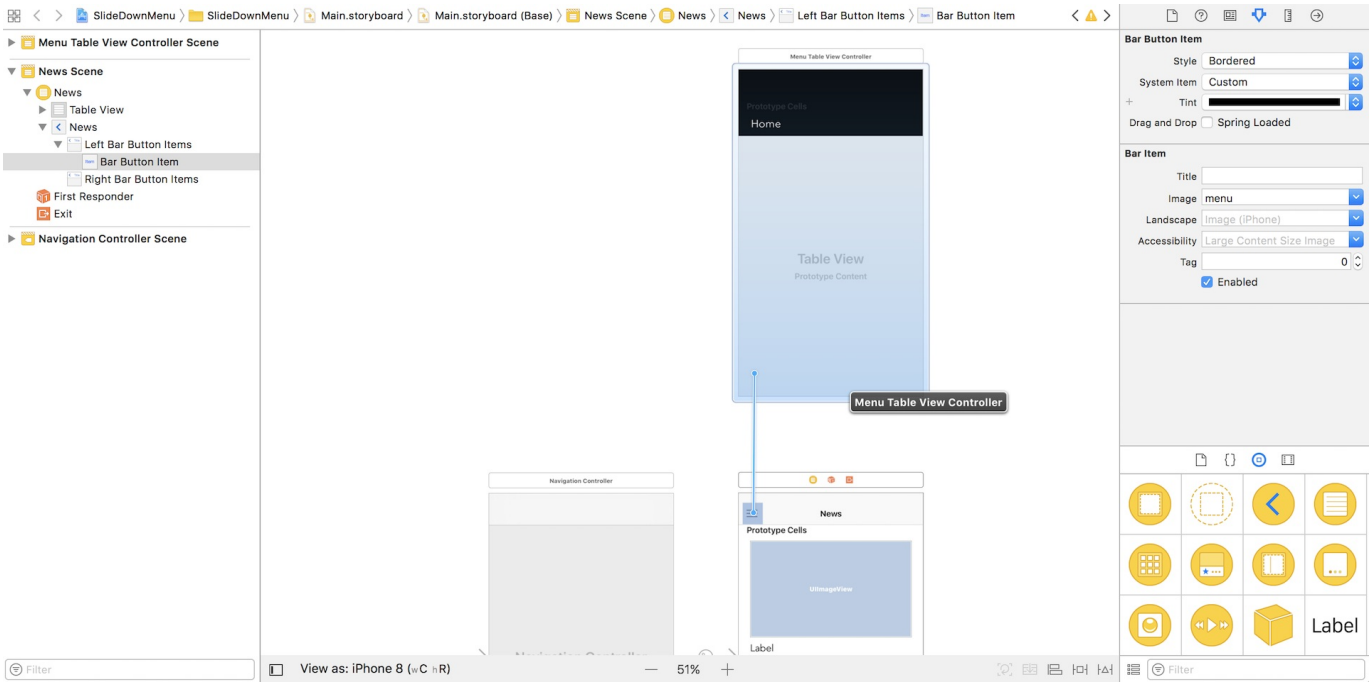


Figure 24.2. Connecting the menu button with the menu view controller

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 `NewsTableViewController.swift` file and insert an unwind action method:

```
@IBAction func unwindToHome(segue: UIStoryboardSegue) {
    let sourceController = segue.source as! MenuTableViewController
    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 `unwindToHomeWithSegue:` option under selection segue.

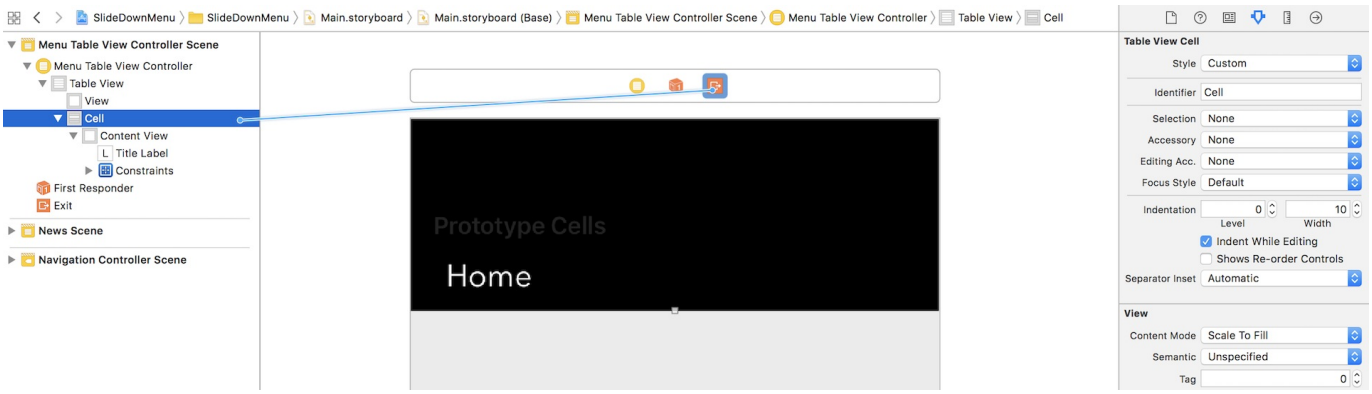


Figure 24.3. Creating the unwind segue for the prototype cell

Now test the app again. 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 prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let menuTableViewController = segue.source as! MenuTableViewController
    if let selectedIndexPath = menuTableViewController.tableView.indexPathForSelec
tedRow {
        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 *Tech* in the menu, the app then changes the title to *Tech*. However, if you tap the menu button again, the menu controller still highlights *Home* in white, instead of *Tech*.

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 prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let menuTableViewController = segue.destination as! MenuTableViewController
    menuTableViewController.currentItem = self.title!
}
```

When the menu button is tapped, the `prepare(for:sender:)` method will be called before trasitioning 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.

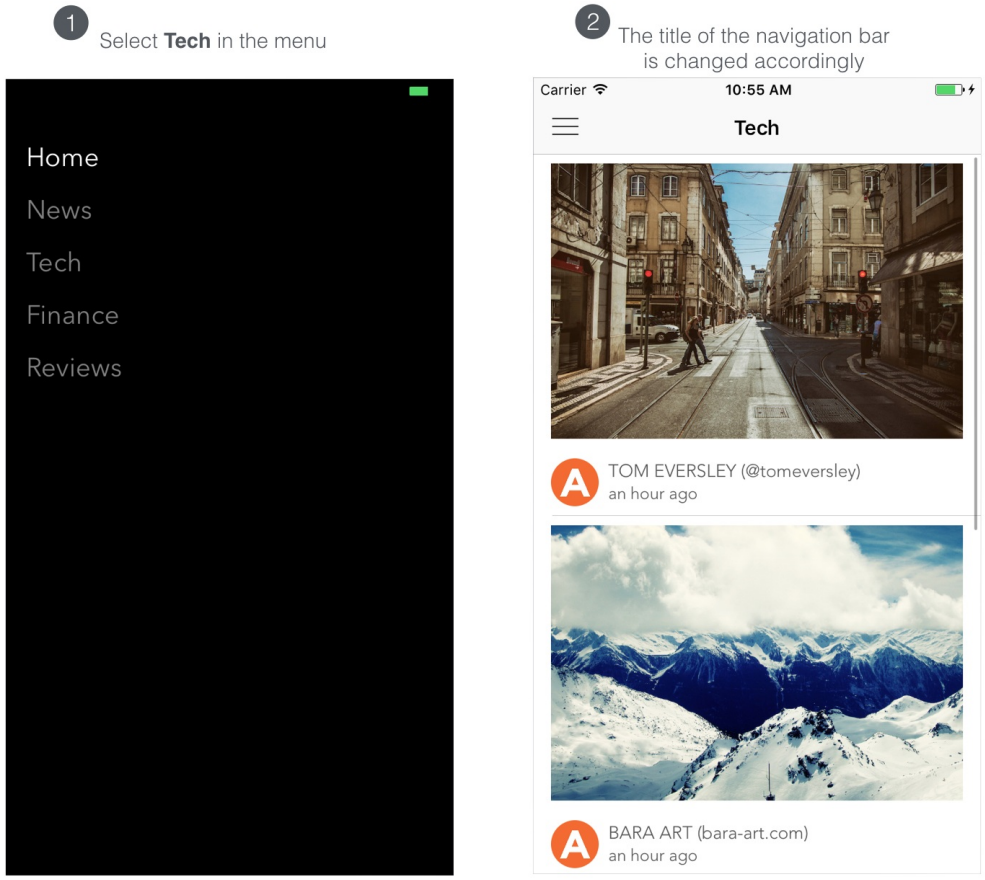


Figure 24.4. The title of the navigation bar is now changed correctly

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.

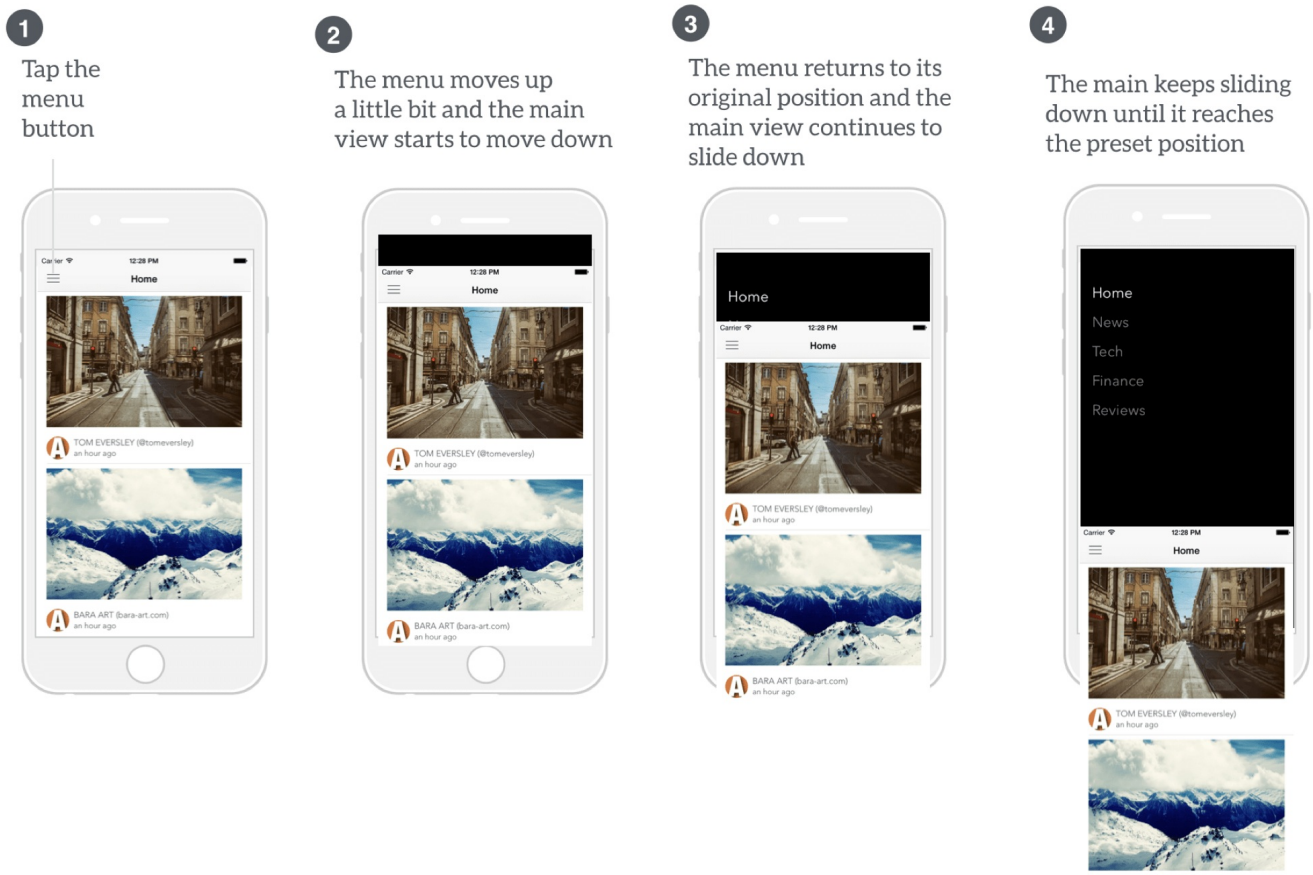


Figure 24.5. The slide down animation

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 Swift file. Name the class `MenuTransitionManager` and update the class like this:

```
import UIKit

class MenuTransitionManager: NSObject, UIViewControllerAnimatedTransitioning, UIVi
ewControllerTransitioningDelegate {

    let duration = 0.5
    var isPresenting = false

    var snapshot:UIView?
```

```
    func transitionDuration(using transitionContext: UIViewControllerContextTransi
tioning?) -> TimeInterval {
        return duration
    }

    func animateTransition(using transitionContext: UIViewControllerContextTransit
ioning) {

        // Get reference to our fromView, toView and the container view
        guard let fromView = transitionContext.view(forKey: UITransitionContextVie
wKey.from) else {
            return
        }

        guard let toView = transitionContext.view(forKey: UITransitionContextViewK
ey.to) else {
            return
        }

        // Set up the transform we'll use in the animation
        let container = transitionContext.containerView
        let moveDown = CGAffineTransform(translationX: 0, y: container.frame.heigh
t - 150)
        let moveUp = CGAffineTransform(translationX: 0, y: -50)

        // Add both views to the container view
        if isPresenting {
            toView.transform = moveUp
            snapshot = fromView.snapshotView(afterScreenUpdates: true)
            container.addSubview(toView)
            container.addSubview(snapshot!)
        }

        // Perform the animation
        UIView.animate(withDuration: duration, delay: 0.0, usingSpringWithDamping:
0.8, initialSpringVelocity: 0.8, options: [], animations: {

            if self.isPresenting {
                self.snapshot?.transform = moveDown
                toView.transform = CGAffineTransform.identity
            } else {
```

```
                self.snapshot?.transform = CGAffineTransform.identity
                fromView.transform = moveUp
            }

        }, completion: { finished in

            transitionContext.completeTransition(true)

            if !self.isPresenting {
                self.snapshot?.removeFromSuperview()
            }
        })
    }

    func animationController(forPresented presented: UIViewController, presenting:
 UIViewController, source: UIViewController) -> UIViewControllerAnimatedTransition
ing? {

        isPresenting = true
        return self
    }

    func animationController(forDismissed dismissed: UIViewController) -> UIViewCo
ntrollerAnimatedTransitioning? {

        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.snapshotView(afterScreenUpdates: true)
```

By calling the `snapshotView(afterScreenUpdates:)` 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 = CGAffineTransform.identity
```

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 `prepare(for:sender:)` method, add a couple of lines of code to hook up the animation and change the modal presentation style:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let menuTableViewController = segue.destination as! MenuTableViewController
    menuTableViewController.currentItem = self.title!
    menuTableViewController.transitioningDelegate = menuTransitionManager
    menuTableViewController.modalPresentationStyle = .fullScreen
}
```

On iOS 13, Apple has changed to present a modal view in a card-like format. In order to make the transition work properly, we have to set the `modalPresentationStyle` property to `.fullScreen`.

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, ac
tion: #selector(delegate.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() {
    dismiss(animated: true, completion: nil)
}
```

Here, we simply dismiss the view controller by calling the `dismiss(animated:completion:)` method.

Lastly, insert a line of code in the `prepare(for:sender:)` 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.

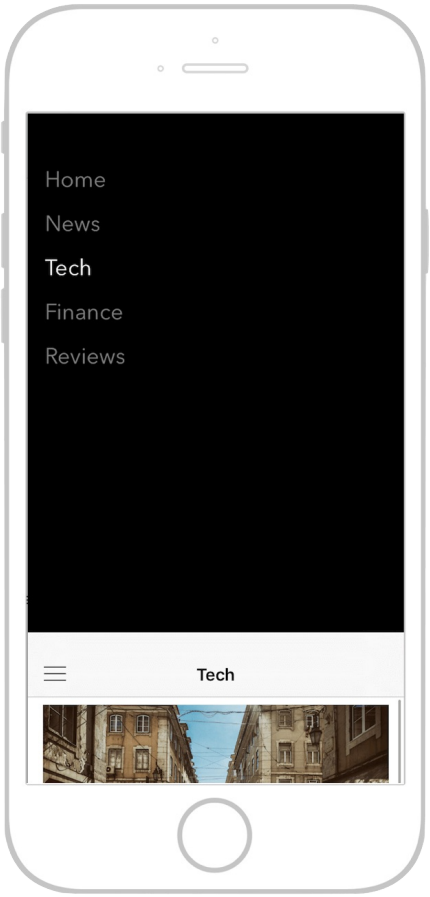


Figure 24.6. Tapping the snapshot now dismisses the menu

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 http://www.appcoda.com/resources/swift5/SlideDownMenu.zip.

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 was seriously one of the most exciting features for the SDK at the time. Prior to iOS 8, if you wanted to display dynamic content in a table view with variable heights, you would need to calculate the row height manually.

In iOS 11, Apple's engineers take this feature even further. The self-sizing feature is enabled automatically. In other words, header views, footer views and cells use self-sizing by default for displaying dynamic content.

While this feature is now enabled without the need of configurations in iOS 13, I want you to understand what happens under the hood.

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 `UITableView.automaticDimension`

If we express the last point in code, it looks like this:

```
tableView.estimatedRowHeight = 95.0
tableView.rowHeight = UITableView.automaticDimension
```

This is what iOS 13 (actually since iOS 11) has done for you.

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.

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

http://www.appcoda.com/resources/swift5/SelfSizingCellStarter.zip and run it, you will have an app like the one shown below.

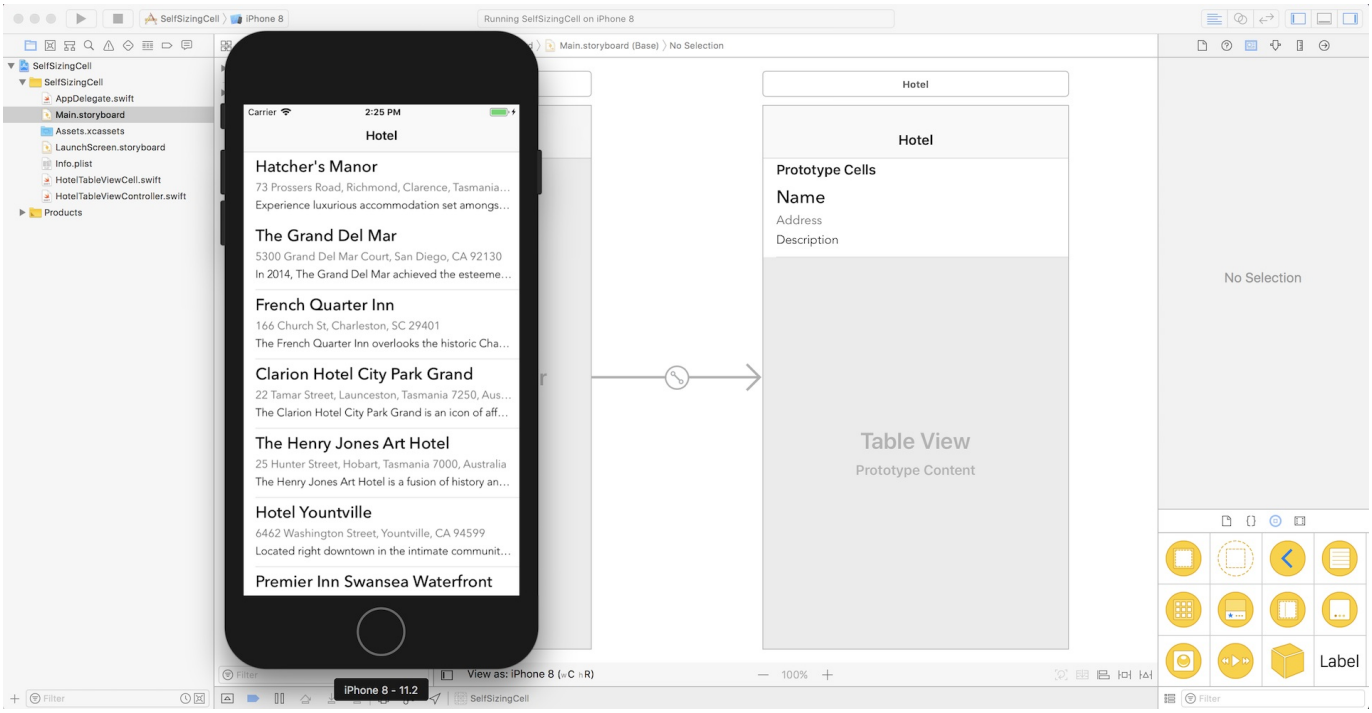


Figure 25.1. Sample result of the starter project

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.

In iOS 13, all you need to do is define appropriate layout constraints 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.

First, press and hold command key to select address, description and name labels. Then click the `Embed in Stack` button to embed them in a stack view.

Next, we are going to add spacing constraints for the stack view. Click the Pin/Add New Constraints button, and set the space value for each side (refer to the figure below). Click `Add 4 Constraints` to add the constraints.

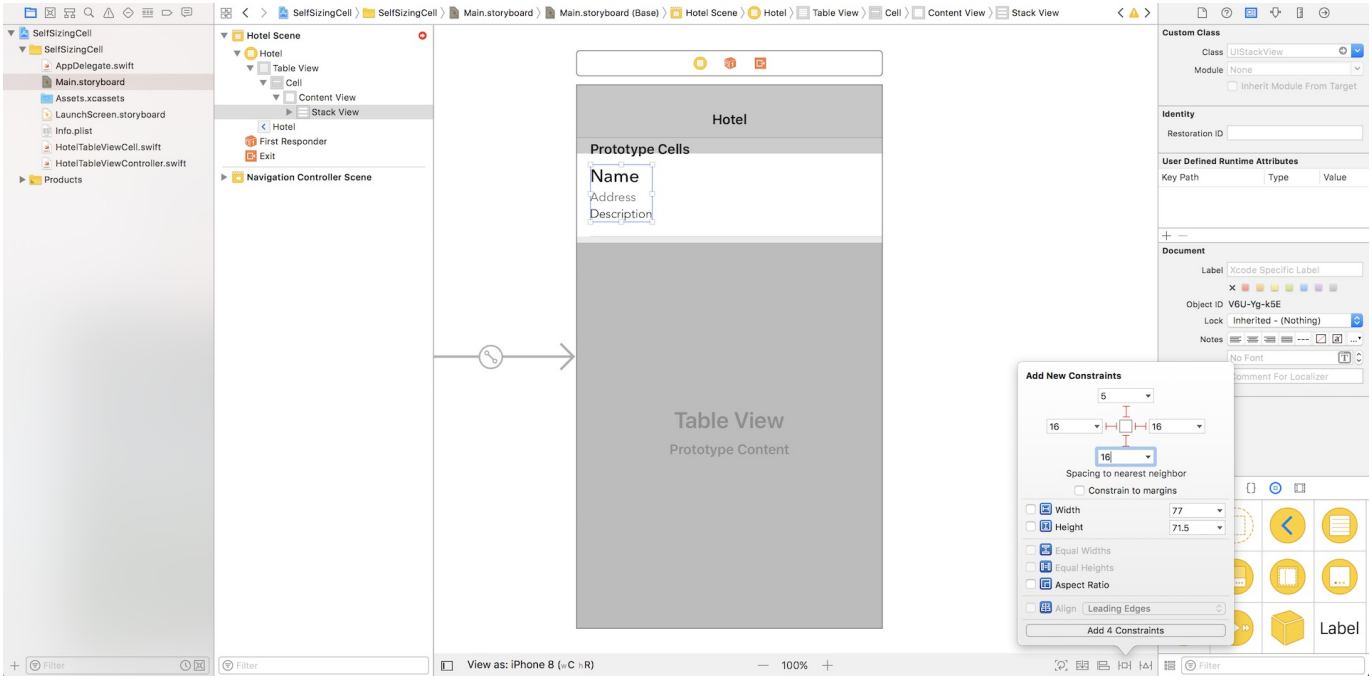


Figure 25.2. Adding spacing constraints for each side of the stack view

Interface Builder detects some ambiguities of the layout. In the Document Outline view, click the disclosure arrow 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).

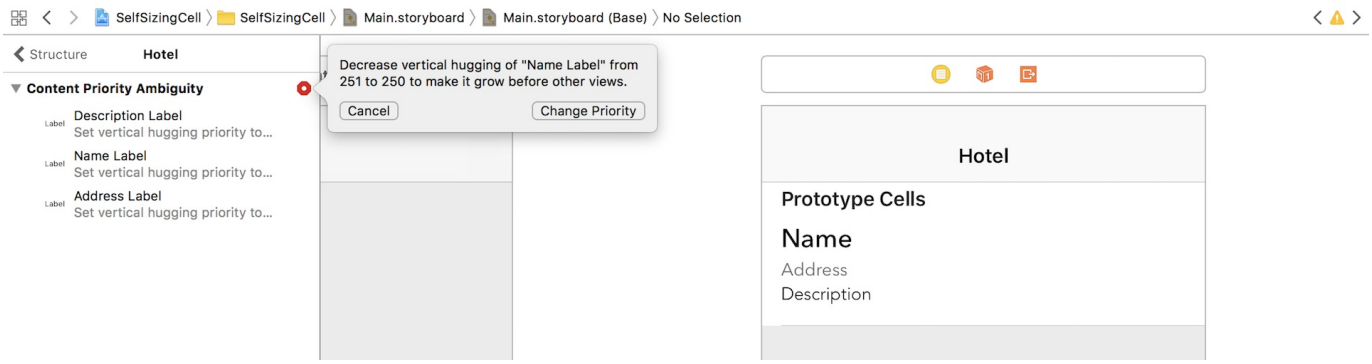


Figure 25.3. Resolving the layout issues

If you have configured the constraints correctly, your final layout should look similar to this:

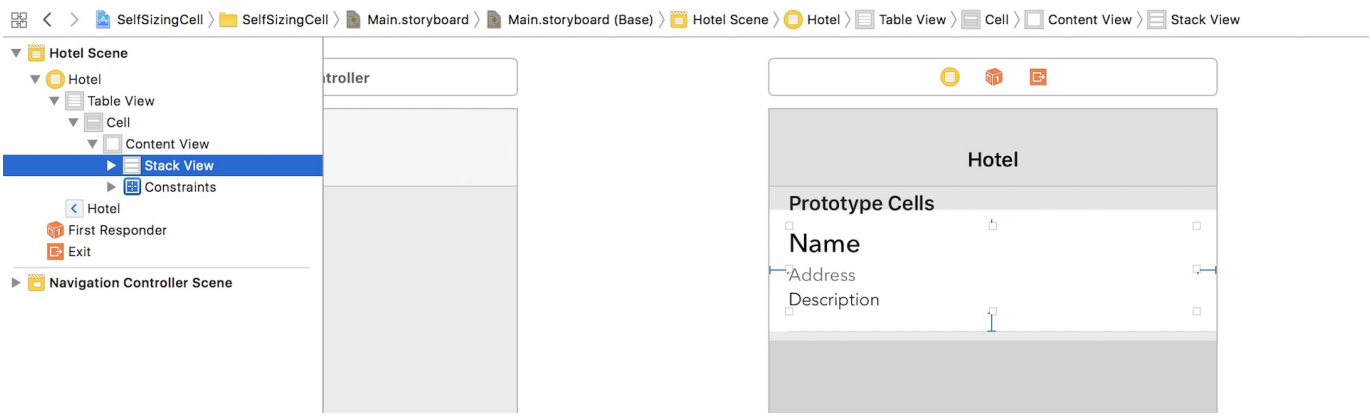


Figure 25.4 The final layout of the prototype cell

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 = UITableView.automaticDimension
```

The lines of code set the `estimatedRowHeight` property to `95` points, which is the current row height, and the `rowHeight` property to `UITableView.automaticDimension`, which is the default row height in iOS. 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.

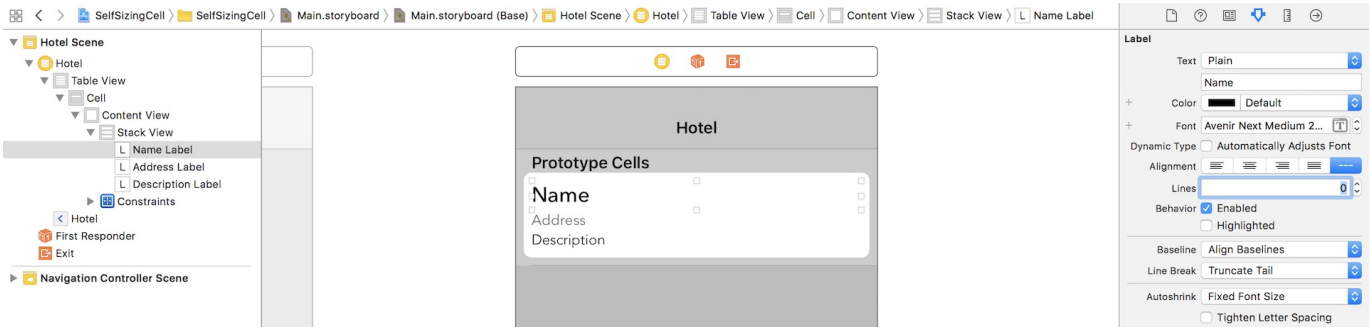


Figure 25.5. Setting the number of lines to 0

Once you made the changes, you can run the project again. This time the cells should be resized properly with respect to the content.

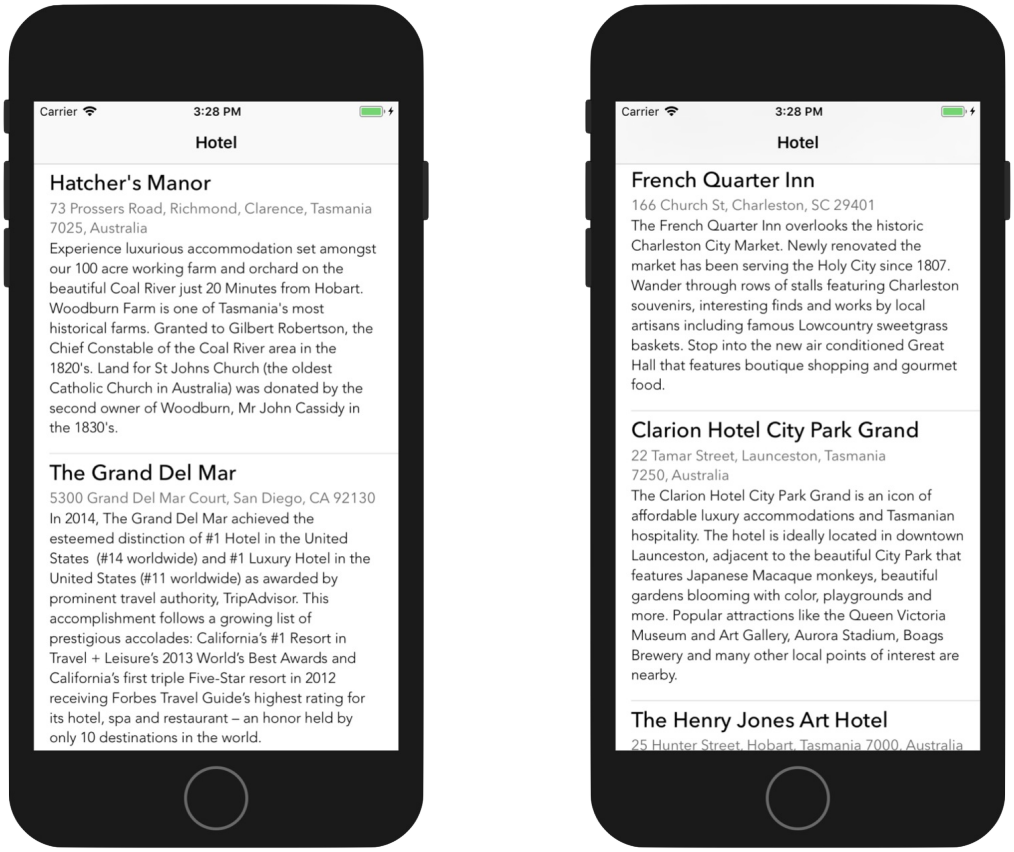


Figure 25.6. The cells now self resize

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.

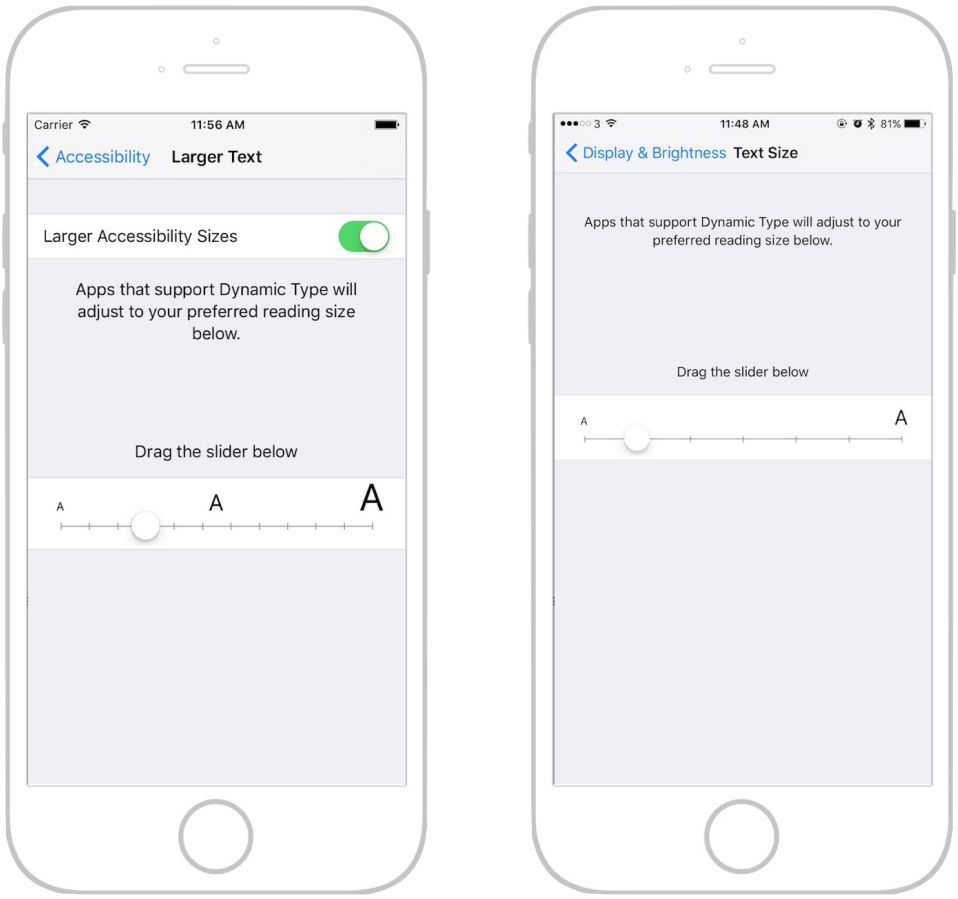


Figure 25.7. Text size configuration in Settings

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.

From iOS 8 and onwards, 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 to the Attributes inspector. Change the font option to `Headline`.

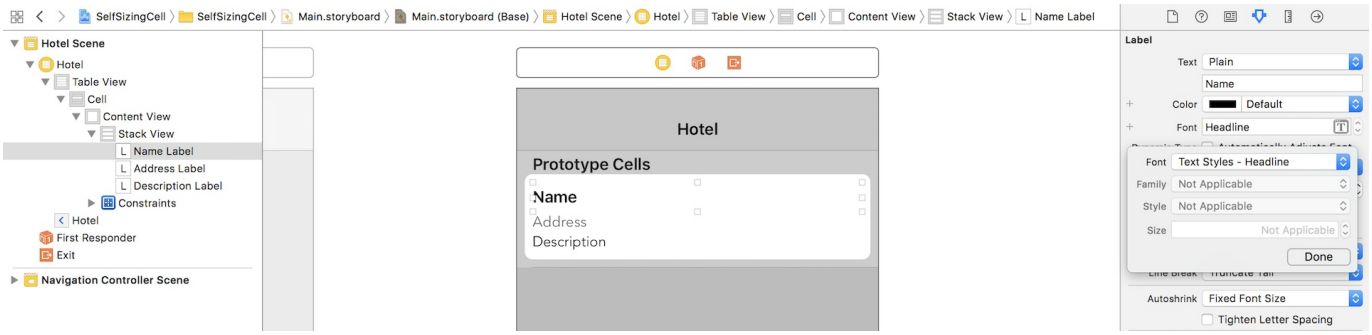


Figure 25.8. Changing the font from a custom font to a text style

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.

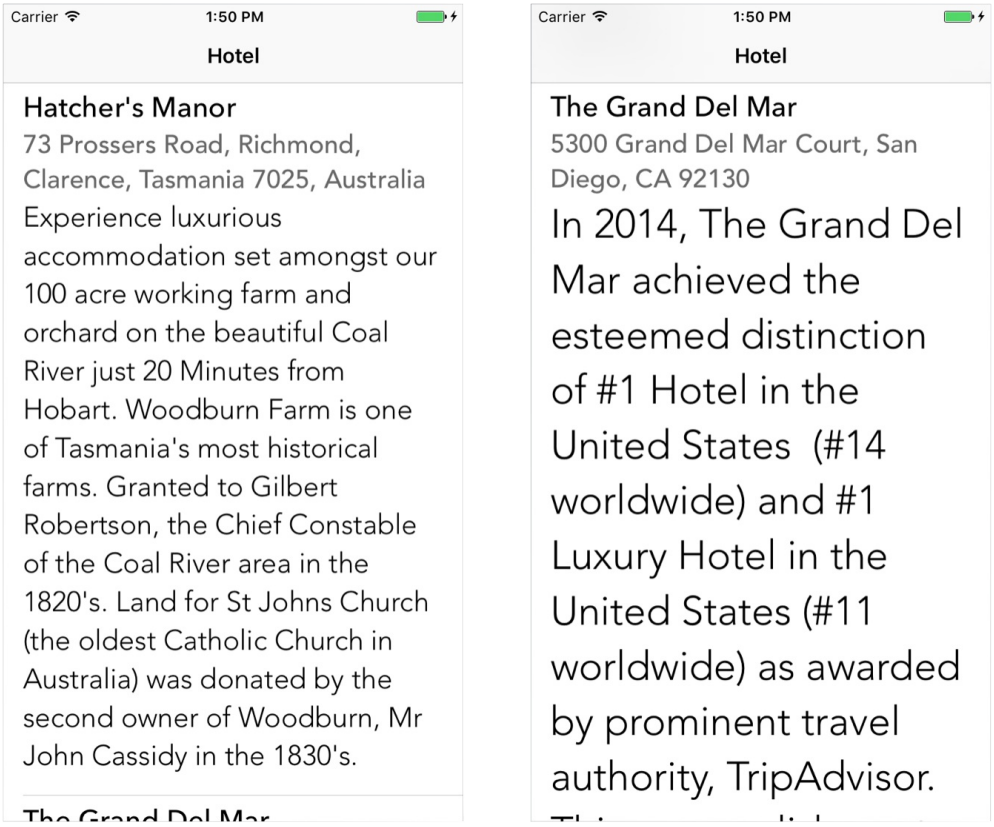


Figure 25.9. Increasing the font size in Settings

Now run the app and it should adapt to the text size change.

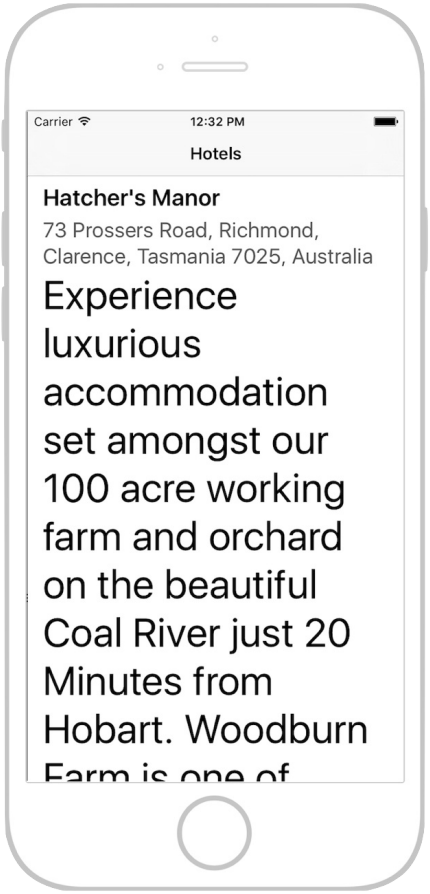


Figure 25.10. The text in the demo app scales automatically

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 re-launch it to effectuate the change.

Prior to iOS 11, your app has to listen to the `UIContentSizeCategoryDidChangeNotification` event and perform a table view reload in order to respond to the size change.

```
NotificationCenter.default.addObserver(self, selector: #selector(onTextSizeChange)
, name: .UIContentSizeCategoryDidChange, object: nil)
```

Now in iOS 13, you just need to enable an option in Interface Builder and iOS will handle the rest. Go to Interface Builder and select the name label. In the Attributes inspector, tick the *Automatically Adjusts Font* option.

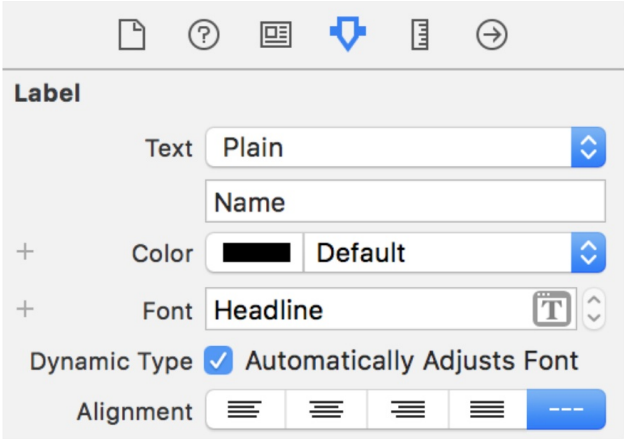


Figure 25.11. The Automatically Adjusts Font option

Repeat the same procedures for the other two labels. 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.

Alternatively, if you prefer to change the setting programmatically, all you need to do is set the label's `adjustsFontForContentSizeCategory` property to `true`. Here is an example:

```
@IBOutlet weak var nameLabel:UILabel! {
    didSet {
        nameLabel.adjustsFontForContentSizeCategory = true
    }
}
@IBOutlet weak var addressLabel:UILabel! {
    didSet {
        addressLabel.adjustsFontForContentSizeCategory = true
    }
}
@IBOutlet weak var descriptionLabel:UILabel! {
    didSet {
        descriptionLabel.adjustsFontForContentSizeCategory = true
    }
}
```

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 default font in iOS. Fortunately, you are allowed to configure your own font for Dynamic Type. You can refer to chapter 16 for the details of the implementation.

Summary

By now, you should understand how to implement self sizing cells. This feature is particularly useful when you need to display dynamic content with variable length. As you can see, the iOS API has has taken care of the heavy lifting. All you need to do is define the required auto layout constraints.

For reference, you can download the final project from http://www.appcoda.com/resources/swift5/SelfSizingCell.zip.

Chapter 26
XML Parsing, RSS and Expandable Table View Cells

One of the most important tasks that a developer has to deal with when creating applications is data handing 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 `JSONSerialization` 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 `XMLParser` 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 `XMLParser` is a very convenient class and makes the parsing of XML data a piece of cake.

Being more specific, let me introduce you the `XMLParserDelegate` 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. </title>) 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 http://www.appcoda.com/resources/swift5/SimpleRSSReaderStarter.zip.

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.

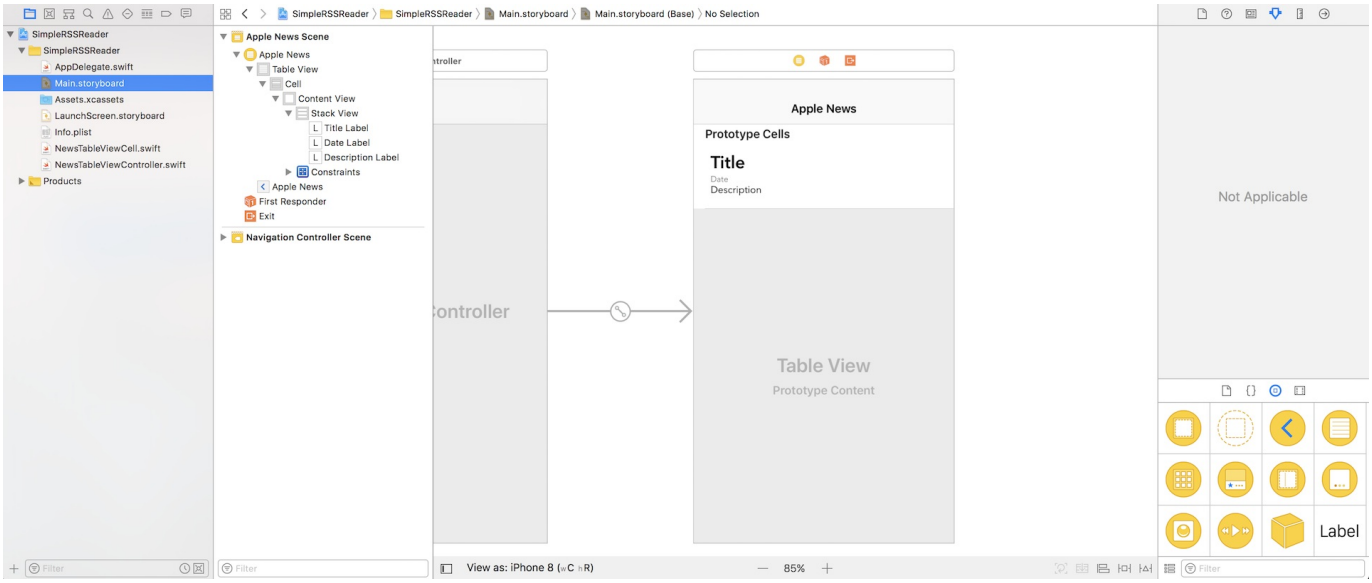


Figure 26.1. The starter project of the Simple RSS Reader app

A Sample RSS Feed

We will use a free RSS feed from Apple (https://developer.apple.com/news/rss/news.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" xmlns:content="http://purl.org/rss/1.0/modules/content/" xmlns:
atom="http://www.w3.org/2005/Atom">
<channel>
<atom:link href="https://developer.apple.com/news/rss/news.rss" rel="self" type="a
pplication/rss+xml" />
<title>News - Apple Developer</title>
<link>https://developer.apple.com/news/</link>
<description>Apple Developer News and Updates feed provided by Apple, Inc.</descri
ption>
<language>en-US</language>
<lastBuildDate>Thu, 16 Nov 2017 13:00:00 PST</lastBuildDate>
<generator>Custom</generator>
<copyright>Copyright 2017, Apple Inc.</copyright>

<item>
<title>Update Your watchOS Apps</title>
<link>https://developer.apple.com/news/?id=11162017a</link>
<guid>https://developer.apple.com/news/?id=11162017a</guid>
<description>Enable your watchOS apps to connect anywhere and anytime, even withou
t a phone nearby, by updating for watchOS 4 and Apple Watch Series 3. Take advanta
ge of increased performance, new background modes for navigation and audio recordi
ng, built-in altimeter capabilities, direct connections to accessories with Core B
luetooth, and more. In addition, the size limit of a watchOS app bundle has increa
sed from 50 MB to 75 MB.Please note that starting April 1, 2018, updates to watchO
S 1 apps will no longer be accepted. Updates must be native apps built with the wa
tchOS 2 SDK or later. New watchOS apps should be built with the watchOS 4 SDK or l
ater.Learn about developing for watchOS 4.</description>
<pubDate>Thu, 16 Nov 2017 13:00:00 PST</pubDate>
<content:encoded><![CDATA[<p>Enable your watchOS apps to connect anywhere and anyt
ime, even without a phone nearby, by updating for watchOS 4 and Apple Watch Series
 3. Take advantage of increased performance, new background modes for navigation a
nd audio recording, built-in altimeter capabilities, direct connections to accesso
ries with Core Bluetooth, and more. In addition, the size limit of a watchOS app b
undle has increased from 50 MB to 75&amp;nbsp;MB.</p><p>Please note that starting Apri
l 1, 2018, updates to watchOS 1 apps will no longer be accepted. Updates must be n
```

```
ative apps built with the watchOS 2 SDK or later. New watchOS apps should be built
 with the watchOS 4 SDK or later.</p><p><a href="https://developer.apple.com/
watchos/">Learn about developing for watchOS 4</a>.</p>]]></content:encoded>
</item>
<item>
<title>Websites on iPhone X</title>
<link>https://developer.apple.com/news/?id=11132017a</link>
<guid>https://developer.apple.com/news/?id=11132017a</guid>
<description>Your websites will automatically display properly on the Super Retina
 screen on iPhone X, as Safari automatically insets your content within the safe a
rea so it’s clear of the rounded corners and sensor housing. If your website is de
signed with full-width horizontal navigation, you can choose to take full advantag
e of the edge-to-edge display by using a new WebKit API introduced in iOS 11.2. St
art testing your website today with the iPhone X simulator, included with Xcode 9.
2 beta.Learn more about designing websites for iPhone X.</description>
<pubDate>Mon, 13 Nov 2017 15:50:00 PST</pubDate>
<content:encoded><![CDATA[<p>Your websites will automatically display properly on
the Super Retina screen on iPhone X, as Safari automatically insets your content w
ithin the safe area so it’s clear of the rounded corners and sensor housing. If yo
ur website is designed with full-width horizontal navigation, you can choose to ta
ke full advantage of the edge-to-edge display by using a new WebKit API introduced
 in iOS 11.2. Start testing your website today with the iPhone X simulator, includ
ed with <a href="https://developer.apple.com/download/">Xcode 9.2 beta</
a>.</p><p><a href="https://webkit.org/blog/7929/designing-websites-for-iphone-x/">
Learn more about designing websites for iPhone X</a>.</p>]]></content:encoded>
</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 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 `XMLParser` class adopts the event-driven approach. It generates a message for each type of parsing event to its delegate, that adopts the `XMLParserDelegate` protocol. To better elaborate the concept, let's consider the following simplified XML content:

```
<item>
<title>Websites on iPhone X</title>
<pubDate>Mon, 13 Nov 2017 15:50:00 PST</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	`parserDidStartDocument(_:)`
2	Found the start tag for element *item*	`parser(_:didStartElement:namespaceURI:qualifiedName:attribu`
3	Found the start tag for element *title*	`parser(_:didStartElement:namespaceURI:qualifiedName:attribu`
4	Found the characters *Websites on iPhone X*	`parser(_:foundCharacters:)`

5	Found the end tag for element *title*	`parser(_:didEndElement:namespaceURI:qualifiedName:)`
6	Found the start tag for element *pubDate*	`parser(_:didStartElement:namespaceURI:qualifiedName:attribu`
7	Found the characters *Mon, 13 Nov 2017 15:50:00 PST*	`parser(_:foundCharacters:)`
8	Found the end tag for element *pubDate*	`parser(_:didEndElement:namespaceURI:qualifiedName:)`
9	Found the end tag for element *item*	`parser(_:didEndElement:namespaceURI:qualifiedName:)`
10	Ended parsing the XML document	`parserDidEndDocument(_:)`

By implementing the methods of the `XMLParserDelegate` 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 Swift file. Name the file `FeedParser`.

Here are a few things we will implement in the class:

1. Download the content of RSS Feed asynchronously. The `XMLParser` 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 `URLSession` to download the content asynchronously.

2. Once the XML content is downloaded, we will initialize an `XMLParser` object to start the parsing.
3. Use the `XMLParser` 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 a `parserCompletionHandler` closure when the parsing completes.

Let's see everything step by step. Initially, open the `FeedParser.swift` file, and adopt the `XMLParserDelegate` protocol. It's necessary to do that in order to handle the data later.

```
typealias ArticleItem = (title: String, description: String, pubDate: String)

class FeedParser: NSObject, XMLParserDelegate {

}
```

Here we also use an alias to represent the tuple, which has the essential fields of an article.

> After a type alias is declared, the aliased *name* can be used instead of the *existing type* everywhere in your program.

Now declare an array of tuples to store the items:

```
private var rssItems: [ArticleItem] = []
```

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://docs.swift.org/swift-book/LanguageGuide/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID329.

Let's also declare an enumeration for the XML tag that we are interested:

```
enum RssTag: String {
    case item = "item"
    case title = "title"
    case description = "description"
    case pubDate = "pubDate"
}
```

Next, declare the following variables in the `FeedParser` class:

```
private var currentElement = ""
private var currentTitle:String = "" {
    didSet {
        currentTitle = currentTitle.trimmingCharacters(in: CharacterSet.whitespace
sAndNewlines)
    }
}
private var currentDescription:String = "" {
    didSet {
        currentDescription = currentDescription.trimmingCharacters(in: CharacterSet
.whitespacesAndNewlines)
    }
}
private var currentPubDate:String = "" {
    didSet {
        currentPubDate = currentPubDate.trimmingCharacters(in: CharacterSet.whites
pacesAndNewlines)
    }
}

private var parserCompletionHandler:(([(title: String, description: String, pubDat
e: 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.

Note: Property observer is a handy feature in Swift. You can specify a `didSet` (or `willSet`) observer for a property. The `willSet` observer is called before the value is store, while the `didSet` observer is called right after the new value is stored. Take the `currentTitle` property as an example. When a new title is set, the code block of `didSet` will be executed and perform change on the new value.

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 = URLRequest(url: URL(string: feedUrl)!)
    let urlSession = URLSession.shared
    let task = urlSession.dataTask(with: request, completionHandler: { (data, resp
onse, error) -> Void in

        guard let data = data else {
            if let error = error {
                print(error)
            }
            return
        }

        // Parse XML data
        let parser = XMLParser(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 `URLSession` 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: XMLParser) {
    rssItems = []
}
```

To begin, here we just initialize an empty `rssItems` array. When a new element (e.g. `<item>`) is found, the `parser(_:didStartElement:namespaceURI:qualifiedName:attributes:)` method is called. Insert this method in the class:

```
func parser(_ parser: XMLParser, didStartElement elementName: String, namespaceURI
: String?, qualifiedName qName: String?, attributes attributeDict: [String : Strin
g] = [:]) {

    currentElement = elementName

    if currentElement == RssTag.item.rawValue {
        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: XMLParser, foundCharacters string: String) {

    switch currentElement {
    case RssTag.title.rawValue: currentTitle += string
    case RssTag.description.rawValue: currentDescription += string
    case RssTag.pubDate.rawValue: 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: XMLParser, didEndElement elementName: String, namespaceURI:
String?, qualifiedName qName: String?) {

    if elementName == RssTag.item.rawValue {
        let rssItem = (title: currentTitle, description: currentDescription, pubDa
te: 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: XMLParser) {
    parserCompletionHandler?(rssItems)
}
```

Optionally, we also implement the following in the `FeedParser` class:

```
func parser(_ parser: XMLParser, parseErrorOccurred parseError: Error) {
    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: [ArticleItem]?
```

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

```
let feedParser = FeedParser()
feedParser.parseFeed(feedUrl: "https://developer.apple.com/news/rss/news.rss", com
pletionHandler: {
    (rssItems: [ArticleItem]) -> Void in

    self.rssItems = rssItems
    OperationQueue.main.addOperation({ () -> Void in
        self.tableView.reloadSections(IndexSet(integer: 0), with: .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: I
nt) -> 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: IndexPat
h) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPat
h) 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.

Figure 26.2. The Simple RSS Reader app now reads and parses the Apple's news feed

That's it. For reference, you can download the final Xcode project from http://www.appcoda.com/resources/swift5/SimpleRSSReader.zip.

Expanding and Collapsing Table View Cells

Currently, the app displays the full content of each news article. It may be a bit lengthy to some users. Wouldn't it be great if we just show the first few lines of the news content? And, when the user taps the cell, it will expand to show the full content. Conversely, if the user taps the cell with full content, the cell collapses to display the excerpt of the article.

With self sizing cells that we have already implemented, it is not very difficult to add this feature.

First, let's limit the description to display the first four lines of the content. There are multiple ways to do that. You can go to the storyboard, and set the *lines* option of the description label to `4`. This time, I want to show you how to do it in code.

Open `NewsTableViewCell.swift` and update the code in the `didSet` observer of the `descriptionLabel` like this:

```
@IBOutlet weak var descriptionLabel:UILabel! {
    didSet {
        descriptionLabel.numberOfLines = 4
    }
}
```

That's it. If you run the app now, it will display an excerpt of the news articles.

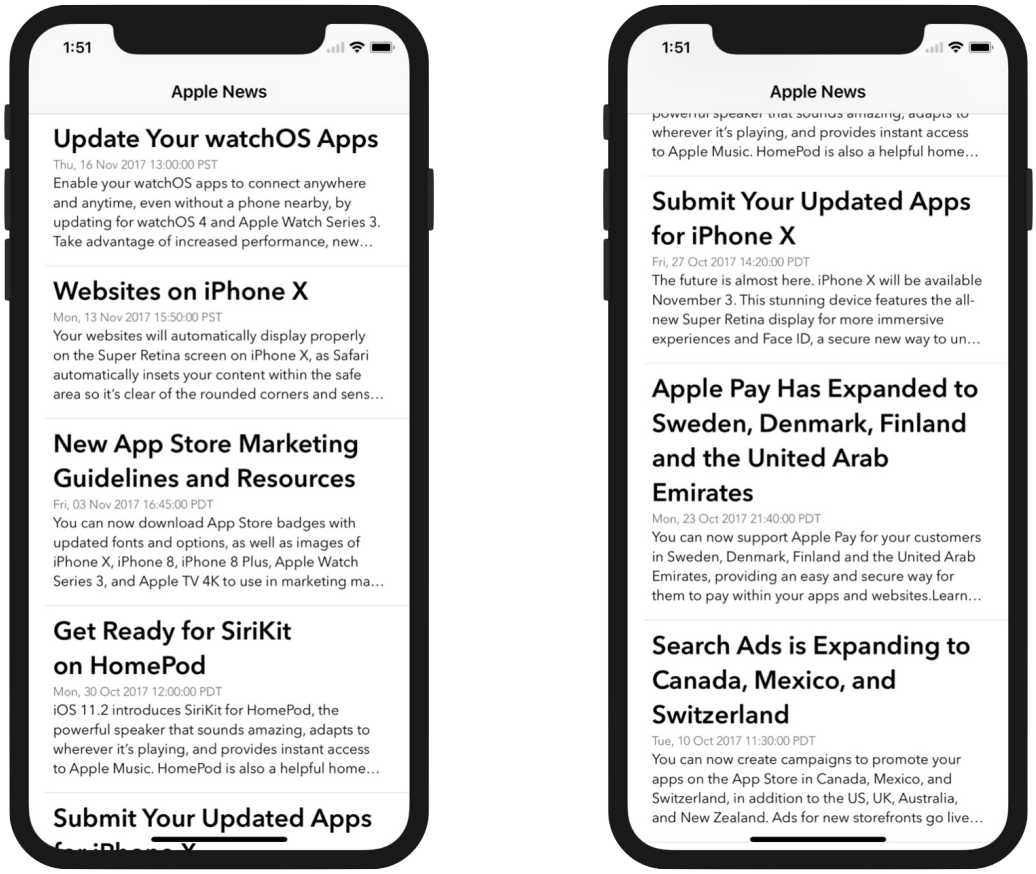


Figure 26.3. The table cells now display the first few lines of the article

Probably you already know how to expand (and collapse) the cell. Here are a couple of things we have to do:

- Implement the `tableView(_:didSelectRowAt:)` method - when a cell is tapped (or selected), this method will be called.
- To expand the cell, we will implement the method to change the `numberOfLines` property of the description label from `4` to `0`. Conversely, when the user taps the cell again, we will change the line number to `4` again.

When you translate the above into code, it will be like this:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexP
ath) {
    tableView.deselectRow(at: indexPath, animated: true)
    let cell = tableView.cellForRow(at: indexPath) as! NewsTableViewCell

    tableView.beginUpdates()
    cell.descriptionLabel.numberOfLines = (cell.descriptionLabel.numberOfLines == 0
) ? 4 : 0
    tableView.endUpdates()
}
```

At the beginning, we deselect the cell and retrieve the selected cell. Then we set the `numberOfLines` property of the description label to either `4` (collapse) or `0` (expand).

You probably notice that we call `beginUpdates()` and `endUpdates()` of `tableView`. This is to notify the table view that we got some changes to the cell. When `endUpdates()` is invoked, the table view animates the cell changes. If you forget to call these two methods, the cell will not update itself even if the value of `numberOfLines` is changed.

Now run the app to have a quick test. It works! Tapping the cell will expand the cell content. If you tap the same cell again, it collapses.

But there is a bug in the existing app.

Say, if you expand a cell, you will find another expanded cell as you scroll through the table. The problem is due to cell reuse as we have explained in the beginner book. To avoid the issue, we will have to keep track of the state (expanded/collapsed) for each cell.

First, declare a new enum called `CellState` in `NewsTableViewController` to indicate the two possible cell states:

```
enum CellState {
    case expanded
    case collapsed
}
```

Next, declare an array variable to store the state for each cell:

```
private var cellStates: [CellState]?
```

The `cellStates` is not initialized by default because we have no idea about the total number of RSS feed items. Instead, we will initialize the array after we retrieve the RSS items in the `viewDidLoad` method. Insert the following line of code after `self.rssItems = rssItems`:

```
self.cellStates = [CellState](repeating: .collapsed, count: rssItems.count)
```

We initialize the array items with the default state `.collapsed`.

Next, modify the `tableView(_:didSelectRowAt:)` method to update the state of the selected cell:

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexP
ath) {
    tableView.deselectRow(at: indexPath, animated: true)
    let cell = tableView.cellForRow(at: indexPath) as! NewsTableViewCell

    tableView.beginUpdates()
    cell.descriptionLabel.numberOfLines = (cell.descriptionLabel.numberOfLines == 0
) ? 4 : 0
    cellStates?[indexPath.row] = (cell.descriptionLabel.numberOfLines == 0) ? .exp
anded : .collapsed
    tableView.endUpdates()
}
```

In the above code, we just update the cell state in reference to the number of lines set in the description label.

Lastly, update the `tableView(_:cellForRowAt:)` method such that we set the `numberOfLines` property of the description label according to the cell state:

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

    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPat
h) as! NewsTableViewCell

    // Configure the cell...
    if let item = rssItems?[indexPath.row] {
        cell.titleLabel.text = item.title
        cell.dateLabel.text = item.pubDate
        cell.descriptionLabel.text = item.description

        if let cellStates = cellStates {
            cell.descriptionLabel.numberOfLines = (cellStates[indexPath.row] == .e
xpanded) ? 0 : 4
        }
    }

    return cell
}
```

Great! The bug should now be fixed. Run the app again and play around with it. All the cells can expand / collapse properly.

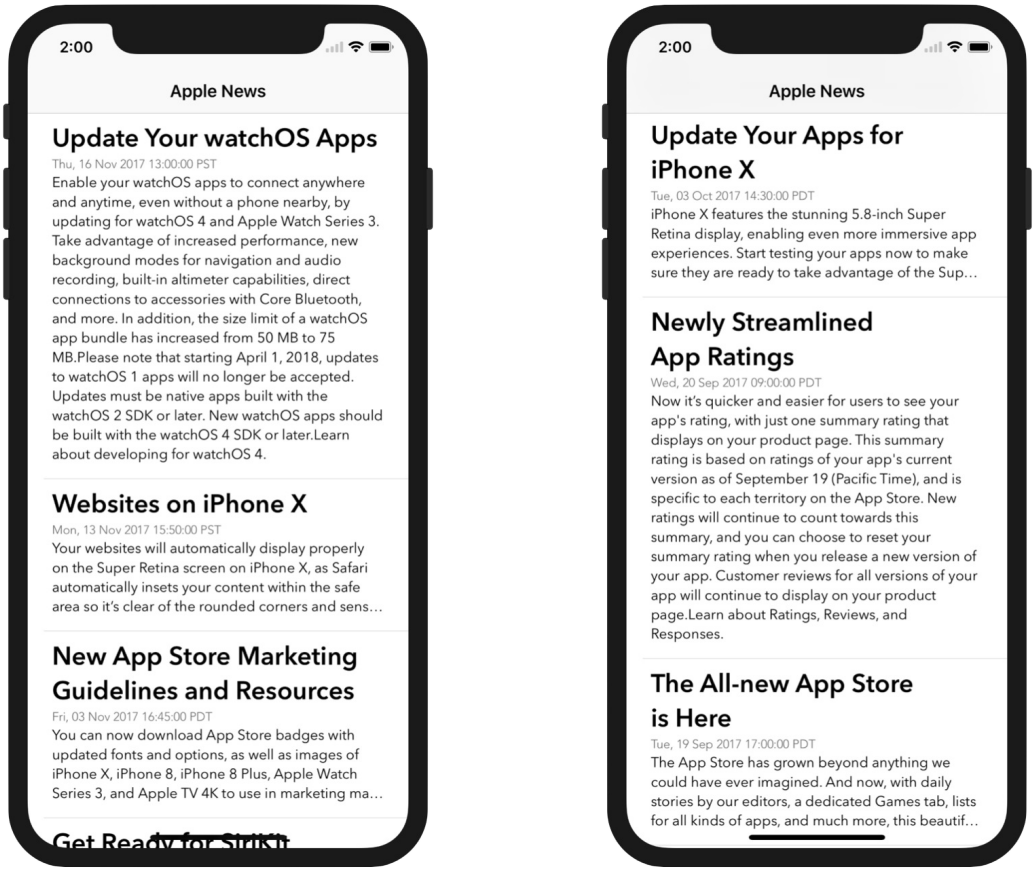
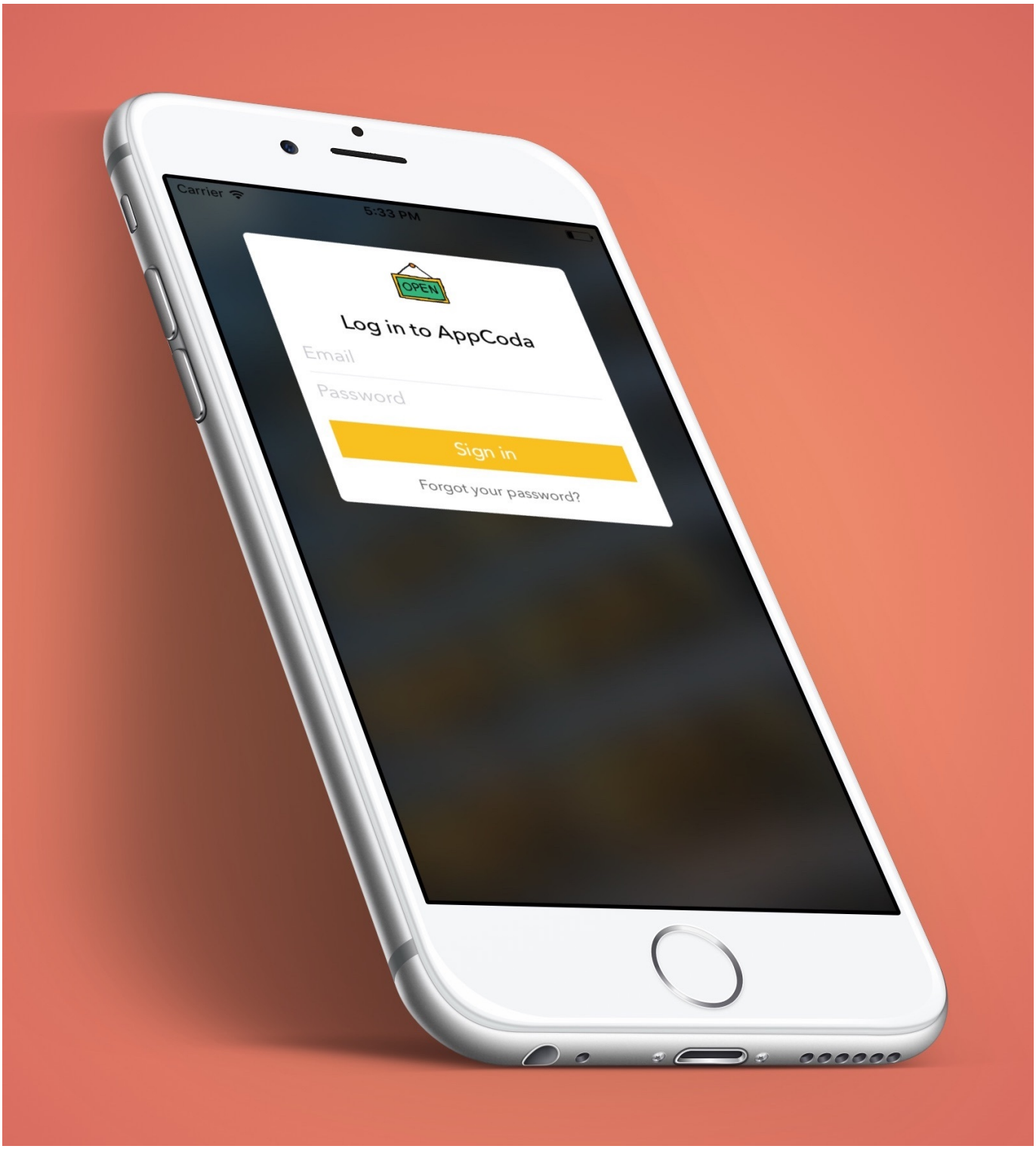


Figure 26.4. Expanding and collapsing table view cells

For reference, you can download the full Xcode project from http://www.appcoda.com/resources/swift5/SimpleRSSReaderCellExpand.zip

Chapter 27
Applying a Blurred Background Using UIVisualEffect



It's been seven 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.

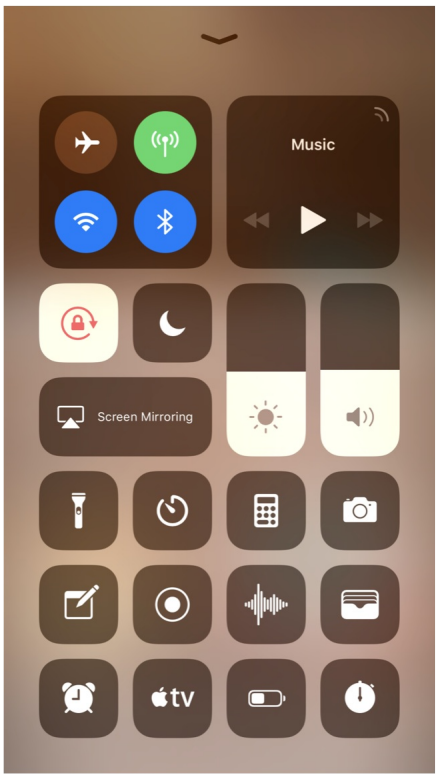


Figure 27.1. Sample translucent and blurring-style effects in iOS

From iOS 8 and onwards, Apple opened the APIs and 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 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 http://www.appcoda.com/resources/swift5/VisualEffectStarter.zip 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.

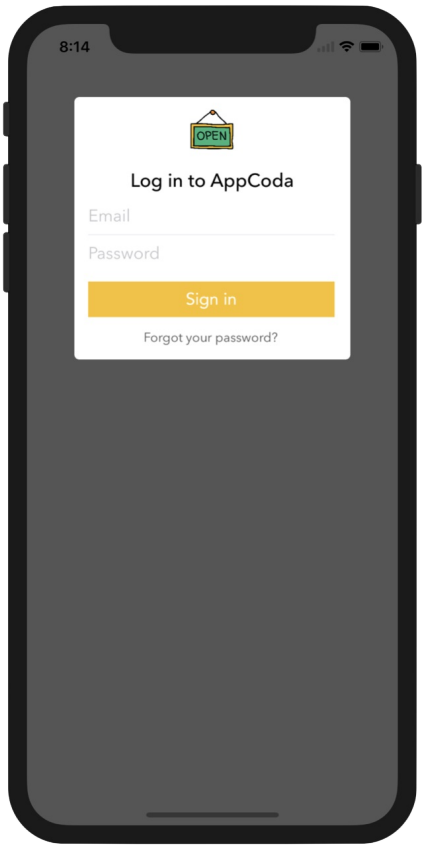


Figure 27.2. The demo app

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 `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.

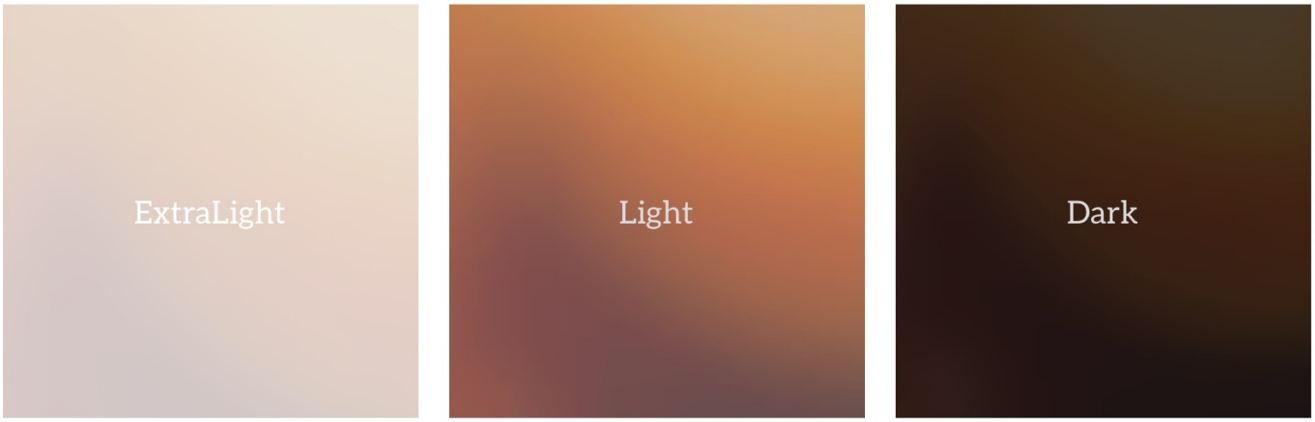


Figure 27.3. Blur effects with different styles

- `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: UIBlurEffect.Style.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 make sure you use iPhone 8 as the device. 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. You can refer to the document outline view and make sure the image view is placed above the login view. In the Size inspector, set the value of x and y to `0`, width to `375` and height to `667`.

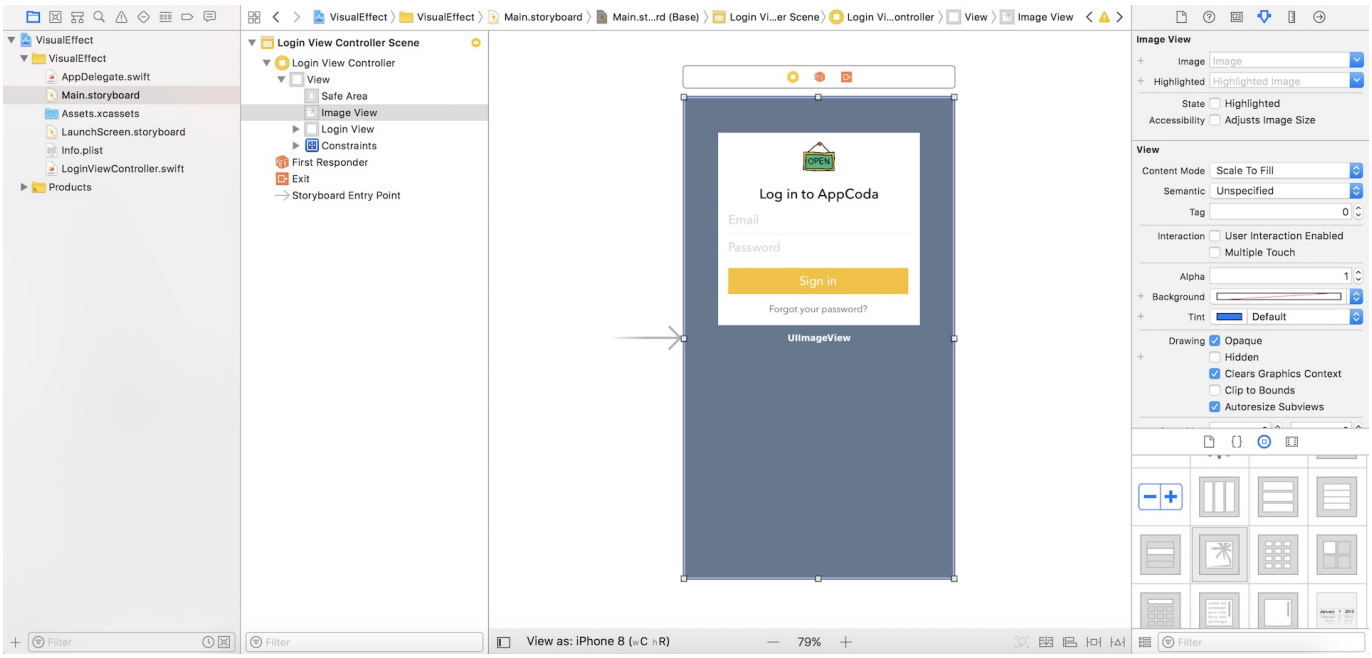


Figure 27.4. Adding an image view to the view controller

Once you have added the image view, select it and add the auto layout constraints. Click the pin button, and set the space value for each side to zero. Make sure you uncheck `Constrain to margins` and then click `Add 4 constraints`.

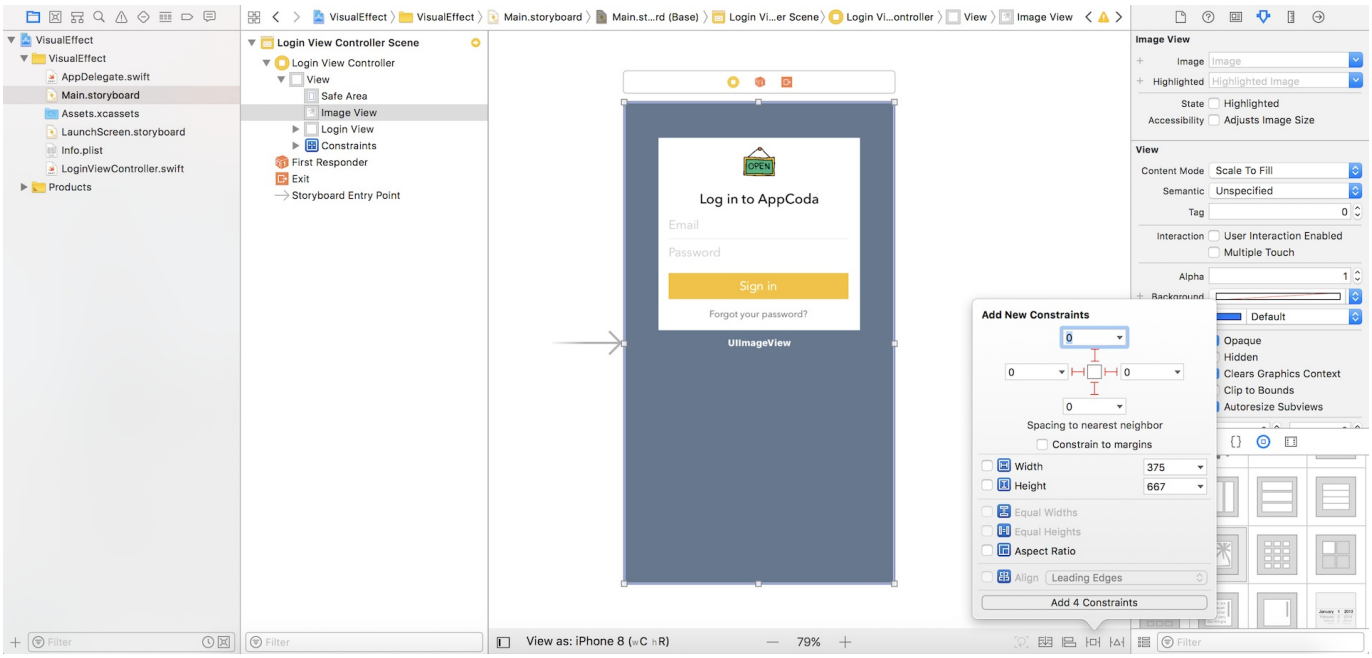


Figure 27.5. Adding an image view to the view controller

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.

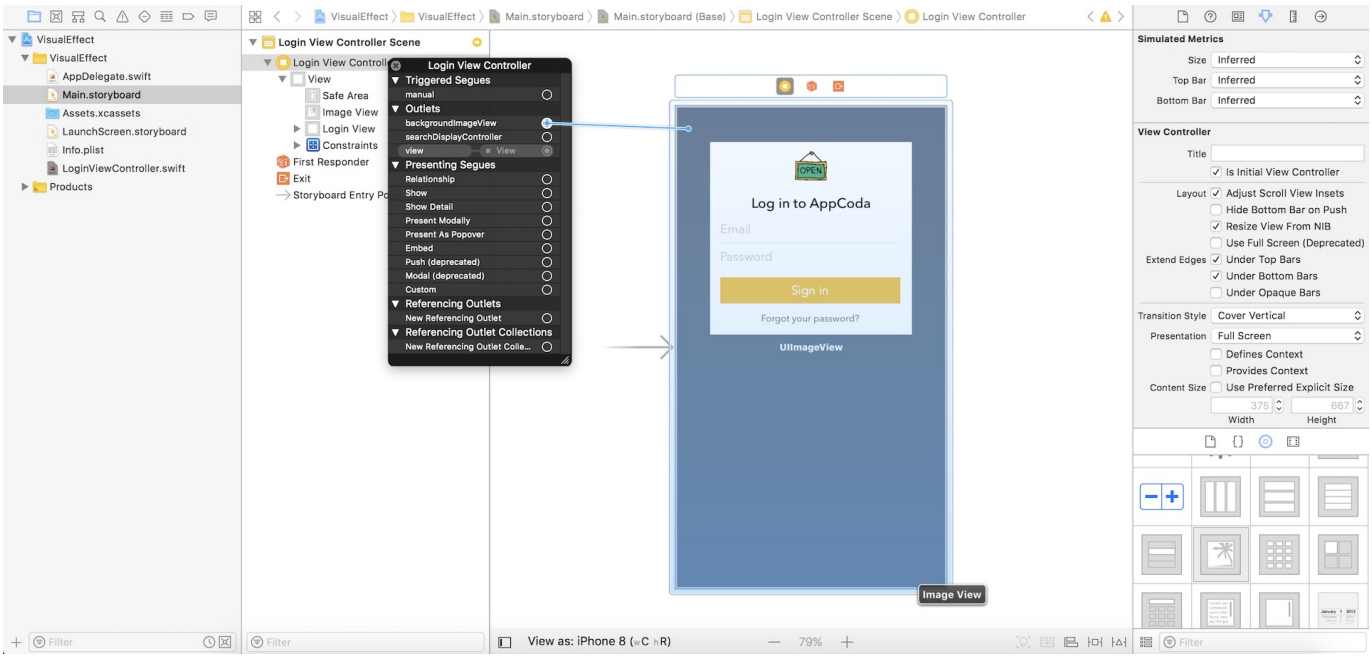


Figure 27.6. Connecting the background image view with the outlet variable

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`:

```
private let imageSet = ["cloud", "coffee", "food", "pmq", "temple"]
```

Also, declare a variable to hold the `UIVisualEffectView` object:

```
private 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: UIBlurEffect.Style.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: UITraitCollectio
n?) {
    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.

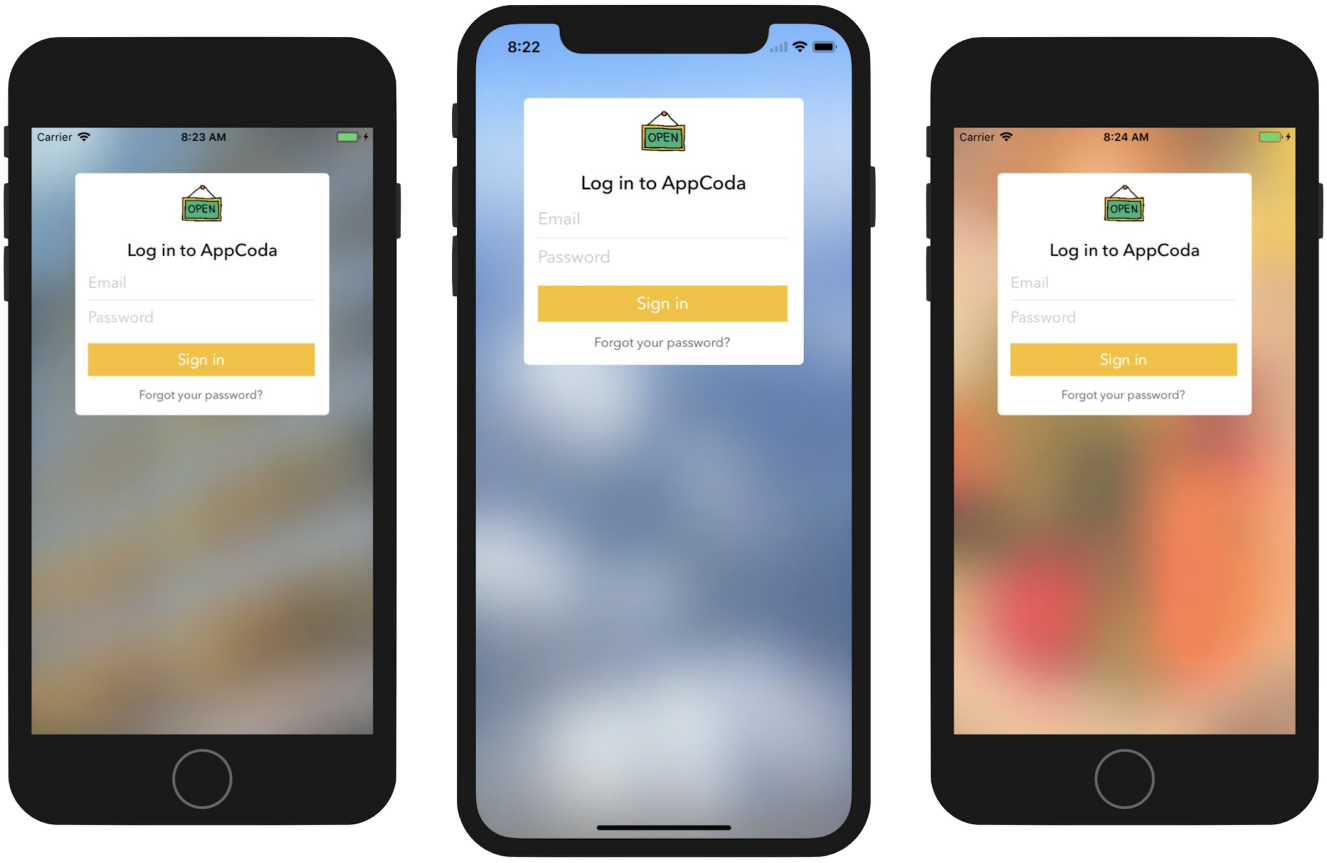


Figure 27.7. Sample blurred backgrounds

Have you tried to turn the simulator sideway? The blurred background works very well on all iPhone devices, except iPhone 11/11 Pro. The look is not bad but you should notice a gray bar on the side.

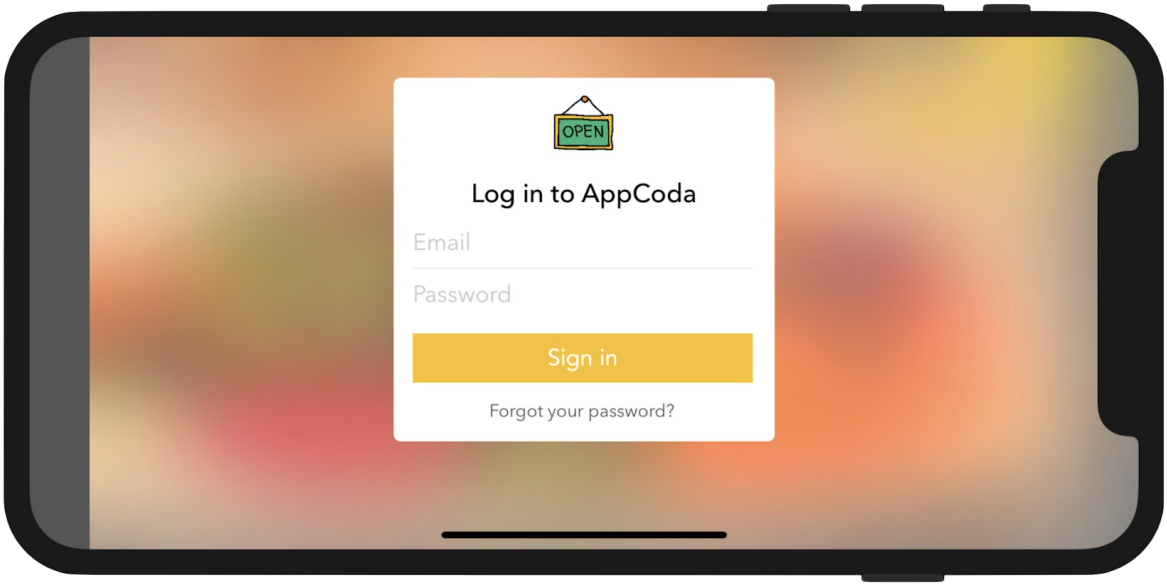


Figure 27.8. The blurred background on iPhone X (landscape)

As you know, since Xcode 9, Apple introduced a new layout concept known as Safe Area. When we add the spacing constraints for the background image view (see figure 27.5), each side of the view is pinned to the layout guide of the safe area.

To fix the layout issue, select the bottom constraint of the background image view. In the Attributes inspector, change the *Second Item* option from `Safe Area.Bottom` to `Superview.Bottom`.

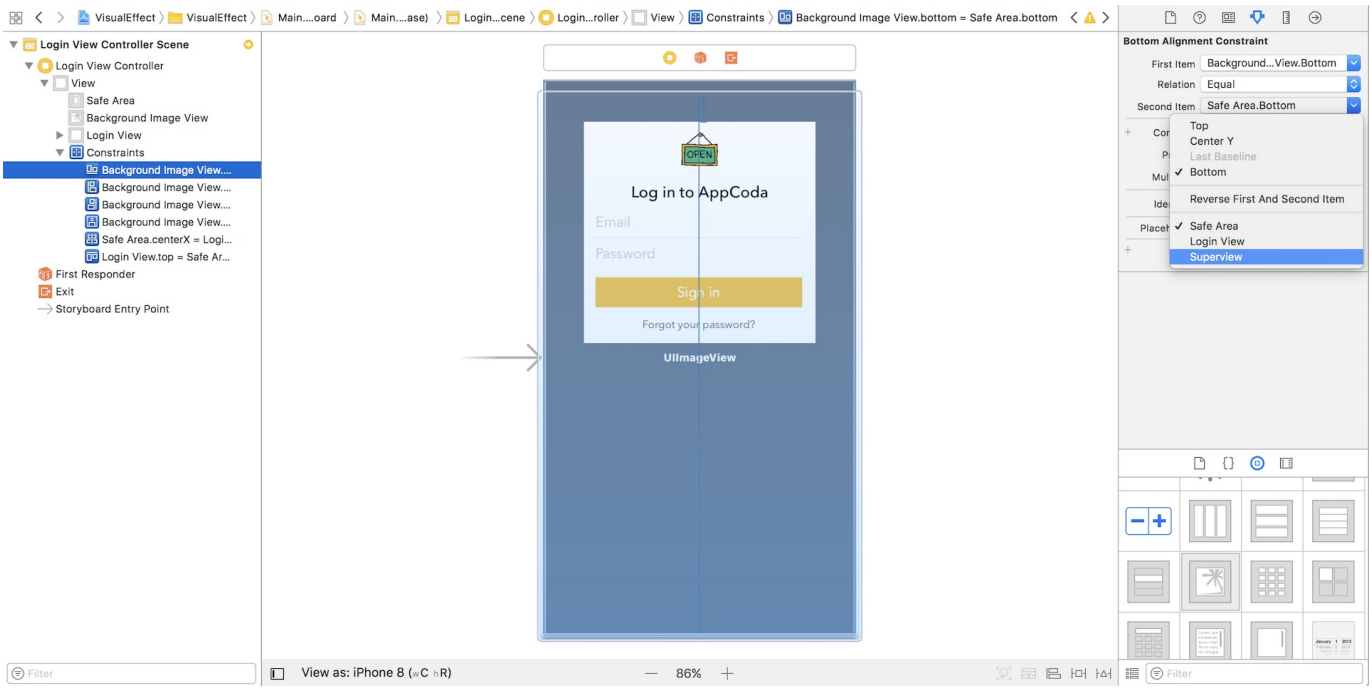


Figure 27.9. The blurred background on iPhone X (landscape)

Repeat the same procedures for the leading, trailing and top constraints such that the *Second Item* option is changed to `Superview`. In other words, we want the background image view extends beyond the safe area.

Now run the app again on iPhone 11/11 Pro. The blurred background should look good even the device is in landscape mode.

For reference, you can download the final project from http://www.appcoda.com/resources/swift5/VisualEffect.zip.

Chapter 28
Using Touch ID and Face ID For Authentication

With the debut of iPhone X in late 2017, iOS now supports two types of authentication mechanism: Touch ID and Face ID.

Let's first talk about the Touch ID.

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.

Along with the release of the new iPhone X in November 2017, Apple is now moving away from Touch ID to Face ID, which uses your face for authentication. By simply glancing at your iPhone X, Face ID securely unlocks the device.

> **Quick note:**To learn more about Face ID, you can refer to this link.

Similar to Touch ID, you can also use this new authentication mechanism to authorize purchases from the App Store and payments with Apple Pay.

Security and privacy are the two biggest concerns for the fingerprint sensor and the Face ID data. 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, A8X, A9 and A10 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.

To safeguard the Face ID data, which is a mathematical representation of your face, Apple applied the same way they did with Touch ID. The Face ID data is encrypted and protected by the Security Enclave.

> **Quick note:**To learn more about Secure Enclave, you can refer to the Apple's Security White Paper for iOS.

The Local Authentication Framework

Back 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. Starting from iOS 10, Apple released the public APIs for Touch ID authentication. Starting from iOS 11, the APIs

support both Touch ID and Face ID, depending on the authentication mechanism the device is using. Therefore, you can now integrate your apps with fingerprint or Face authentication, potentially replacing passwords or PINs.

The usage of the Touch ID or Face ID is based on the 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 or Face 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 & Face ID) and `deviceOwnerAuthentication` are the two available policies. The former policy indicates the device owner must use Touch ID / Face ID for authentication. On the other hand, the latter policy indicates the device owner can authenticate using biometry or the device password. In other words, the device owner is asked to authenticate through Touch ID (or Face ID) first if it is available and enabled. Otherwise, the user is asked to enter the device passcode.
- `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 for 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 http://www.appcoda.com/resources/swift5/TouchIDStarter.zip. 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 or Face ID. If the authentication is successful, the app will show you the Home screen. However, the fingerprint or face authentication is not always successful. The user may use a device without Touch ID or Face 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. For iPhone X with Face ID enabled, the user just needs to look at the iPhone and then the app will automatically perform the authentication.
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.

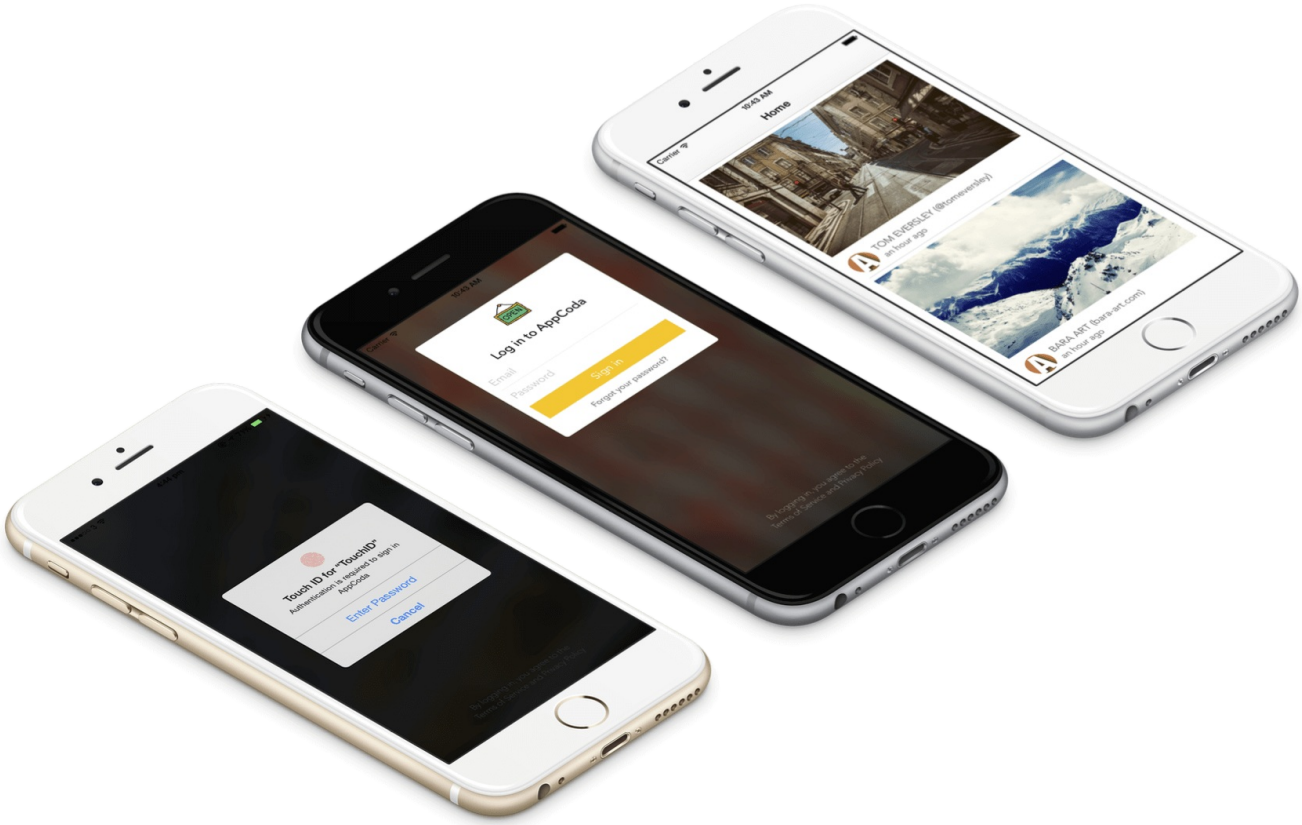


Figure 28.1. Authenticating using Touch ID

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.

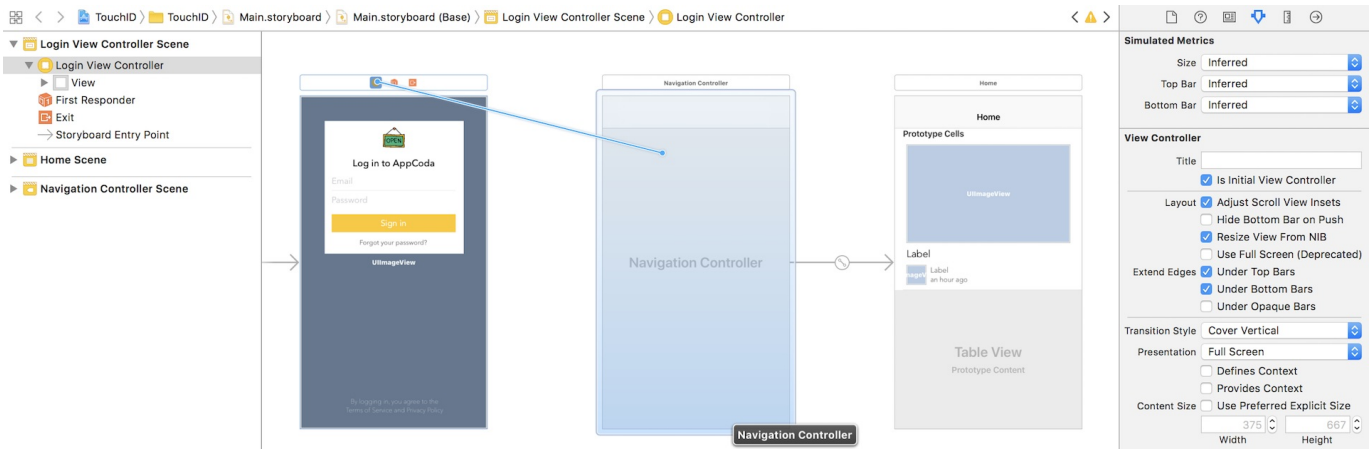


Figure 28.2. Control-drag from the Login View Controller to the navigation controller

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/Face ID is based on the Local Authentication framework. 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.

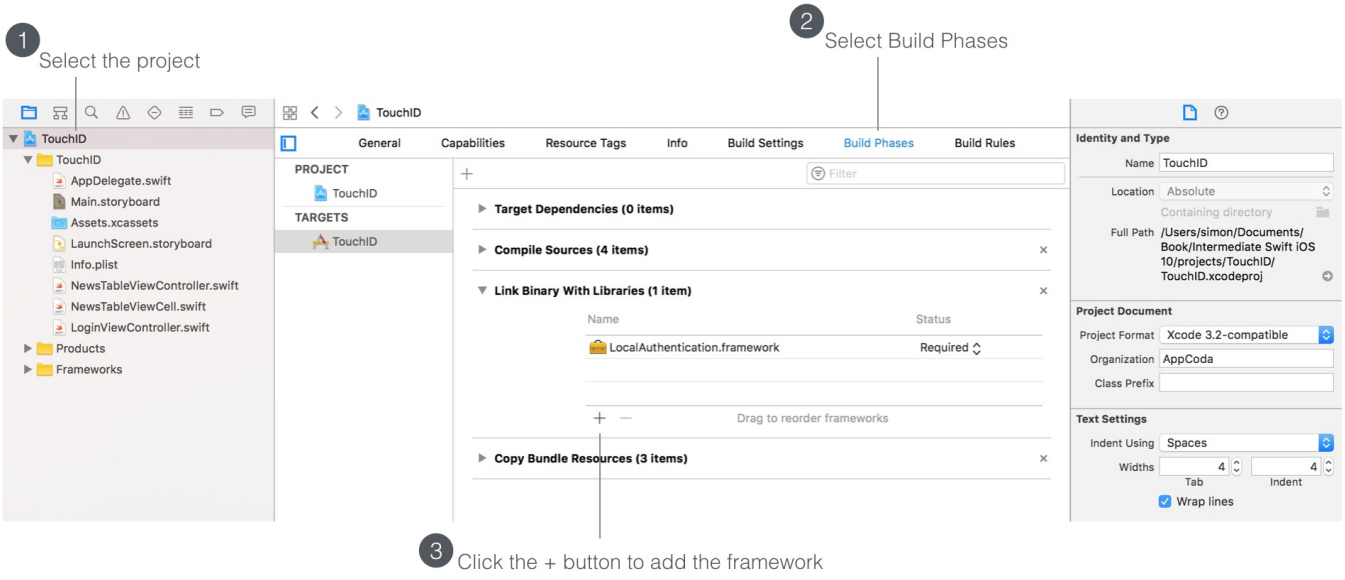


Figure 28.3. Adding the LocalAuthentication framework to the project

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.isHidden = 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 and Face ID, the login view is hidden by default.

To support Touch ID and Face ID, we will create a new method called `authenticateWithBiometric` in the class. Let's start with the code snippet and insert it in the `LoginViewController` class:

```
func authenticateWithBiometric() {
    // Get the local authentication context.
    let localAuthContext = LAContext()
    let reasonText = "Authentication is required to sign in AppCoda"
    var authError: NSError?

    if !localAuthContext.canEvaluatePolicy(LAPolicy.deviceOwnerAuthenticationWithB
iometrics, error: &authError) {

        if let error = authError {
            print(error.localizedDescription)
        }

        // Display the login dialog when Touch ID is not available (e.g. in simula
tor)
        showLoginDialog()

        return
    }
}
```

The core of the Local Authentication framework is the `LAContext` class. To use Touch ID or Face ID, the very first thing is to instantiate a `LAContext` object.

The next step is to ask the framework if the Touch ID or Face ID authentication can be performed on the device by calling the `canEvaluatePolicy` method. As mentioned earlier, the framework supports the `deviceOwnerAuthenticationWithBiometrics` policy, that indicates the device owner authenticates using Touch ID.

We pass this policy to the method to check if the device supports Touch ID or Face ID authentication. If the method returns a `true` value, this indicates the device is capable to use one of these biometric authentication and the user has enabled either Touch ID or Face ID as the authentication mechanism. If a `false` value is returned, that means you

cannot use any of them to authenticate the user. In this case, you should provide an alternative authentication method. Here we just call up the `showLoginDialog` method to fall back to password authentication.

Once we've confirmed that the Touch ID / Face ID is supported, we can proceed to perform the corresponding authentication. Continue to insert the following lines of code in the `authenticateWithBiometric` method:

```
// Perform the Biometric authentication
localAuthContext.evaluatePolicy(LAPolicy.deviceOwnerAuthenticationWithBiometrics,
localizedReason: reasonText, reply: { (success: Bool, error: Error?) -> Void in

    // Failure workflow
    if !success {
        if let error = error {
            switch error {
            case LAError.authenticationFailed:
                print("Authentication failed")
            case LAError.passcodeNotSet:
                print("Passcode not set")
            case LAError.systemCancel:
                print("Authentication was canceled by system")
            case LAError.userCancel:
                print("Authentication was canceled by the user")
            case LAError.biometryNotEnrolled:
                print("Authentication could not start because you haven't enrolled
 either Touch ID or Face ID on your device.")
            case LAError.biometryNotAvailable:
                print("Authentication could not start because Touch ID / Face ID i
s not available.")
            case LAError.userFallback:
                print("User tapped the fallback button (Enter Password).")
            default:
                print(error.localizedDescription)
            }
        }

        // Fallback to password authentication
        OperationQueue.main.addOperation({
            self.showLoginDialog()
```

```
            })

        } else {

            // Success workflow

            print("Successfully authenticated")
            OperationQueue.main.addOperation({
                self.performSegue(withIdentifier: "showHomeScreen", sender: nil)
            })
        }

    })
```

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 if the device supports Touch ID. 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.

For devices that support Face ID, no dialog will be presented. The user just needs to look at the iPhone to perform the Face ID authentication.

In the closure, we first check if the authentication is successful. If it is true, we simply call up the `performSegue(withIdentifier:)` 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 / Face 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.
- `.biometryNotAvailable` - Touch ID / Face ID is not available on the device.
- `.biometryNotEnrolled` - the user has not configured either Touch ID or Face ID yet.

In the implementation, we simply log the error to the console. When an error occurs, the app will fall back to password authentication by calling `showLoginDialog()`:

```
// Fallback to password authentication
OperationQueue.main.addOperation({
    self.showLoginDialog()
})
```

Because the reply block is run in the 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:

```
authenticateWithBiometric()
```

Before you run the project to test the app, you will have to edit the `Info.plist` file and insert an entry with the key `Privacy - Face ID Usage Description` for Face authentication. In its value field, specify a reason why your app needs biometric authentication.

Now you're ready to test the app. Make sure you run the app on a real device with Touch ID or Face ID support (e.g. iPhone 8 or iPhone X). Once launched, the app should ask for Touch ID authentication if your device supports Touch ID. On iPhone X, you just need to look at your device and Face ID authentication happens instantly.

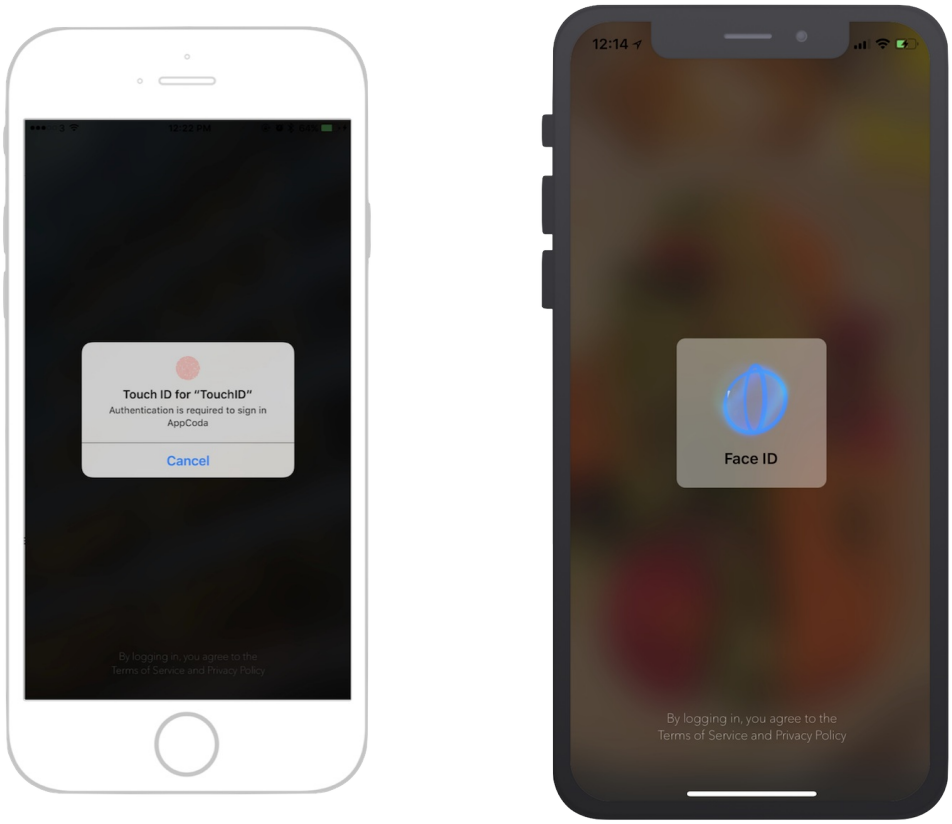


Figure 28.4. Authenticating with Touch ID or Face ID

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:

```
No identities are enrolled.
```

Password Authentication

Now you have implemented the Touch ID / Face 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"
 {
        performSegue(withIdentifier: "showHomeScreen", sender: nil)
    } else {
        // Shake to indicate wrong login ID/password
        loginView.transform = CGAffineTransform(translationX: 25, y: 0)
        UIView.animate(withDuration: 0.2, delay: 0.0, usingSpringWithDamping: 0.15
, initialSpringVelocity: 0.3, options: .curveEaseInOut, animations: {

            self.loginView.transform = CGAffineTransform.identity

        }, 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.

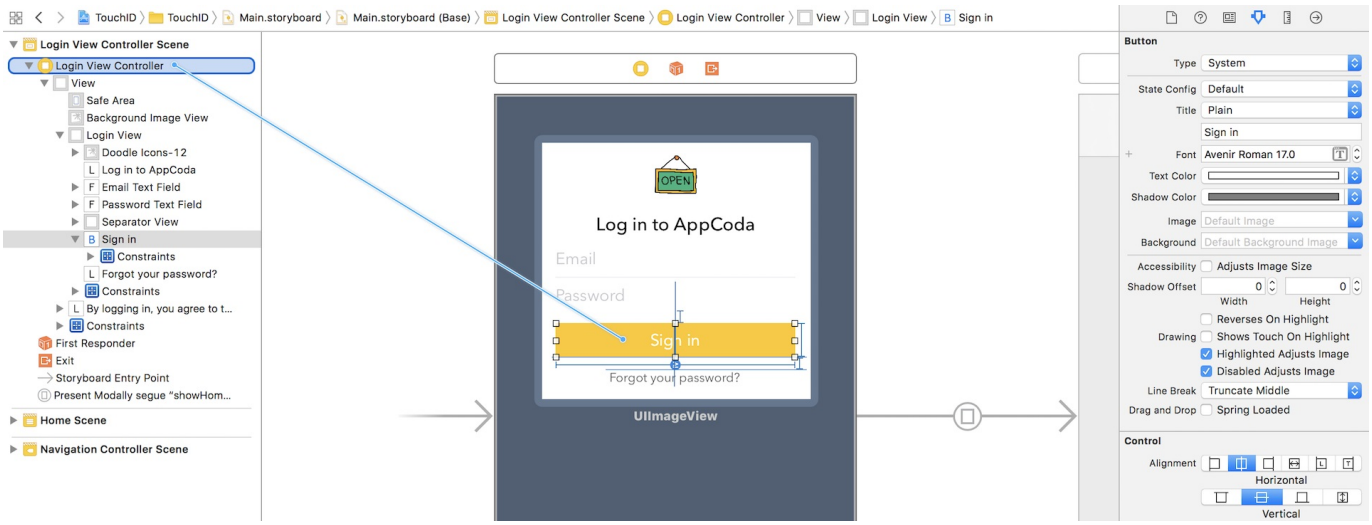
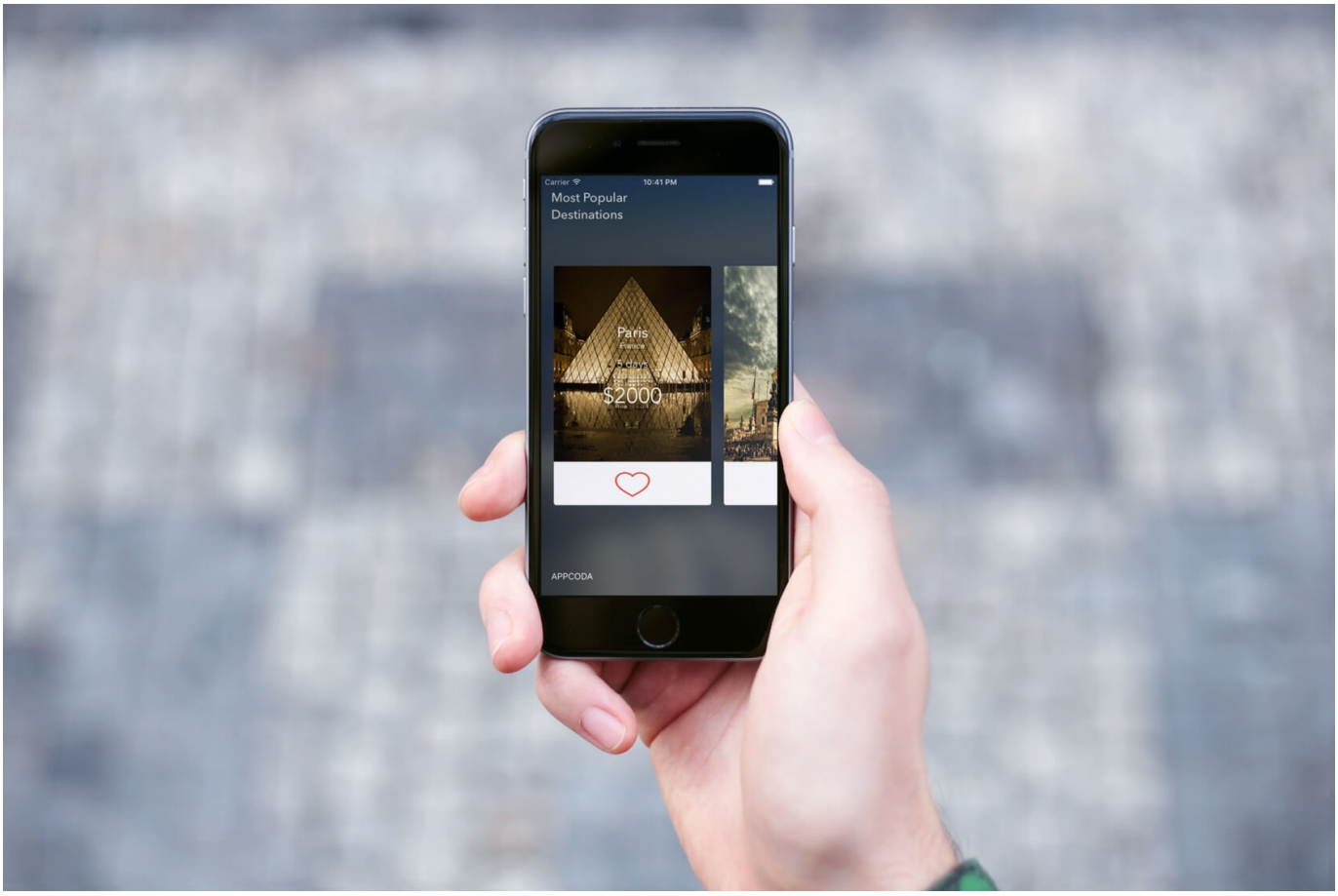


Figure 28.5. Connecting the Sign In button with the action method

Build and run the project again. You should now be able to log in the app even if you choose to fall back to the password authentication. Tapping the Sign In button without entering the password will "shake" the login dialog.

For reference, you can download the final project from http://www.appcoda.com/resources/swift5/TouchID.zip.

Chapter 29
Building a Carousel-Like User Interface



Kickstarter is one of my favorite crowdfunding services. The current version of the app uses a table view to list all the crowdfunding projects. Before the revamp of its user interface, it displayed all featured projects in a carousel, with which you could 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.

Figure 29.1. A carousel UI design (left: An older version of the Kickstarter app, right: our demo app)

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 http://www.appcoda.com/resources/swift5/TripCardStarter.zip. 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 `375` 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.

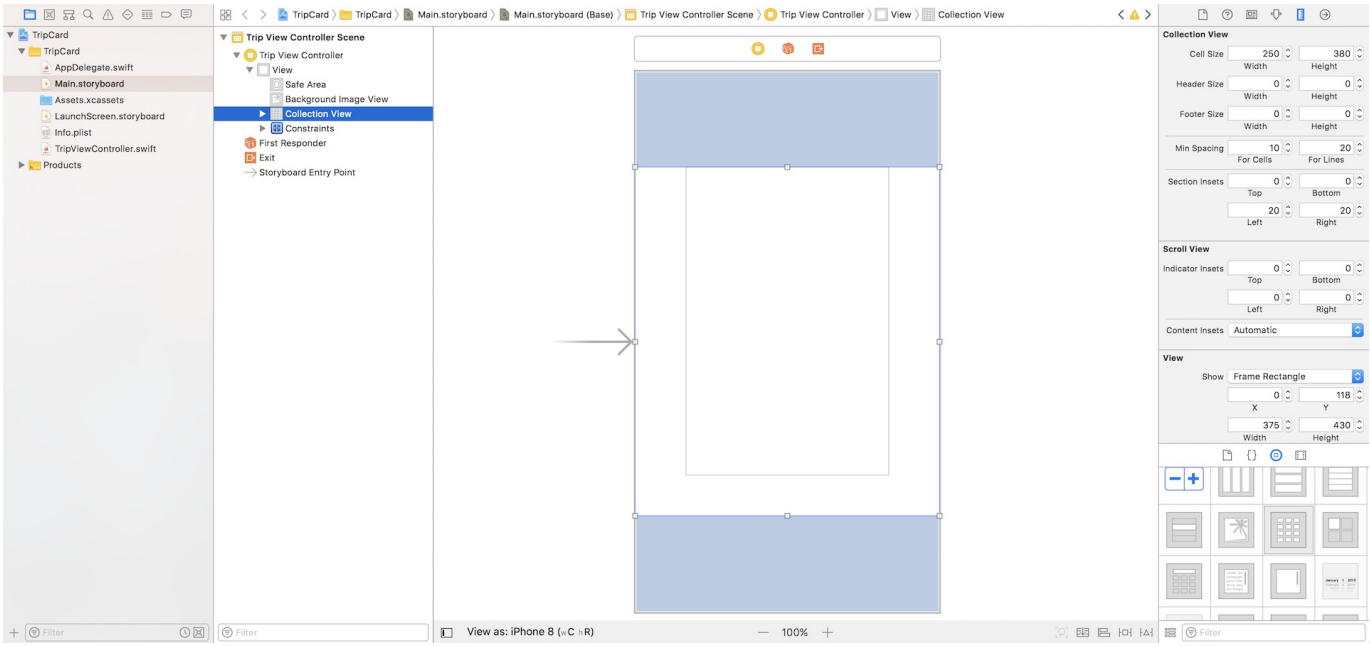


Figure 29.2. Adding a collection view to the view controller

Your storyboard should 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:

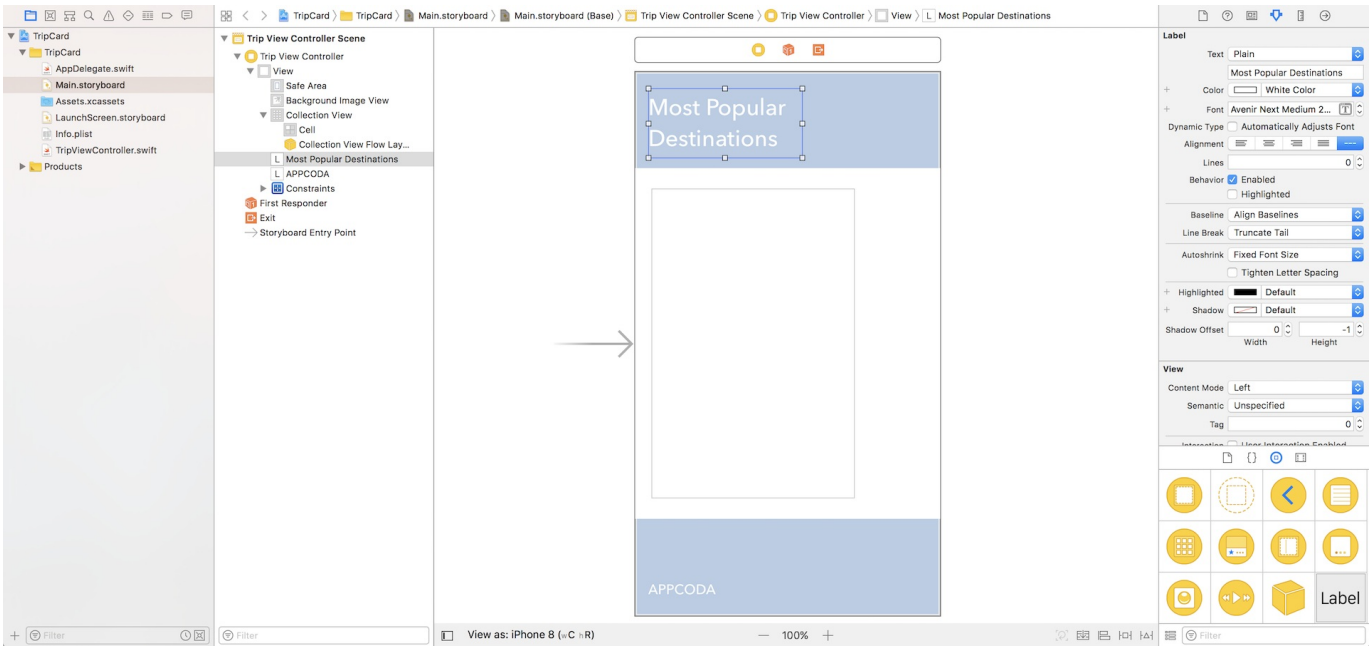


Figure 29.3. Adding two labels to the view controller

So far we haven't configured any auto layout constraint. First, select the *Most Popular Destinations* label. Click the *Add New Constraint* (or Pin) button to add a couple of spacing and size constraints. Select the left and top bar, and check both width and height checkboxes. Click `Add 4 Constraints` to add the constraints.

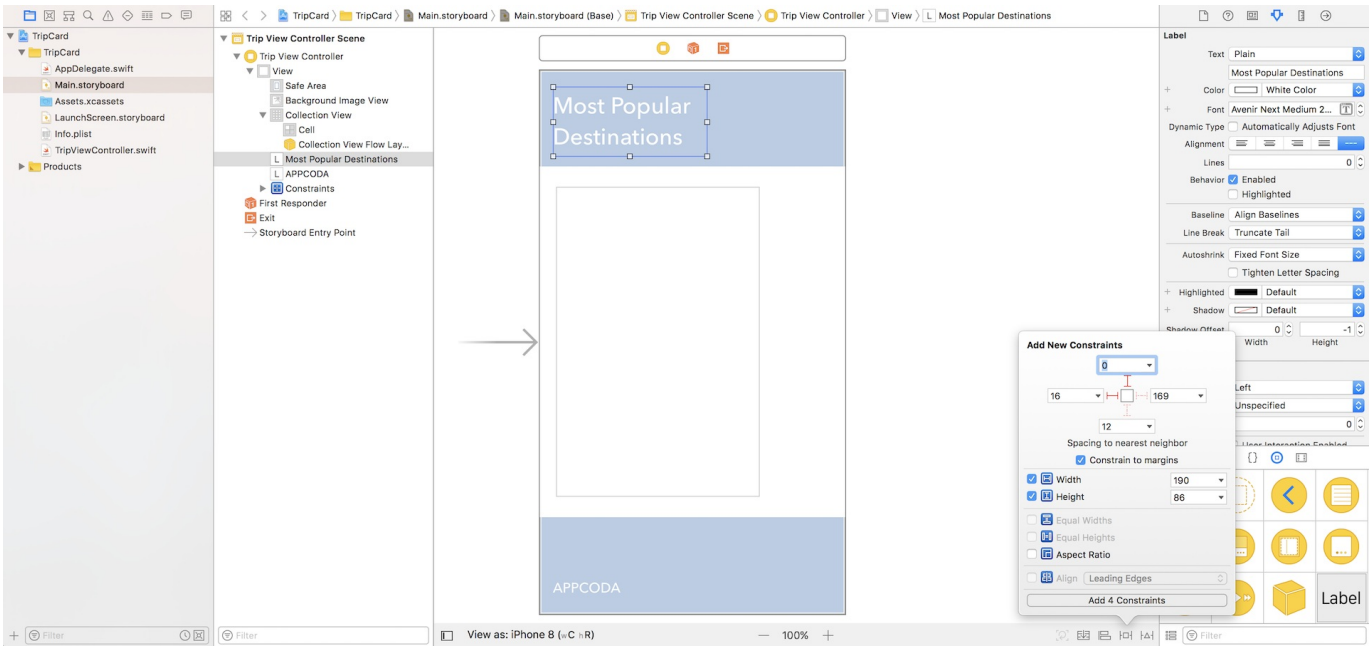


Figure 29.4. Adding constraints for the title label

For the bottom label, click `Add New Constraints` button to add two spacing constraints. Click the bar of both left and bottom sides, and then click `Add 2 Constraints`.

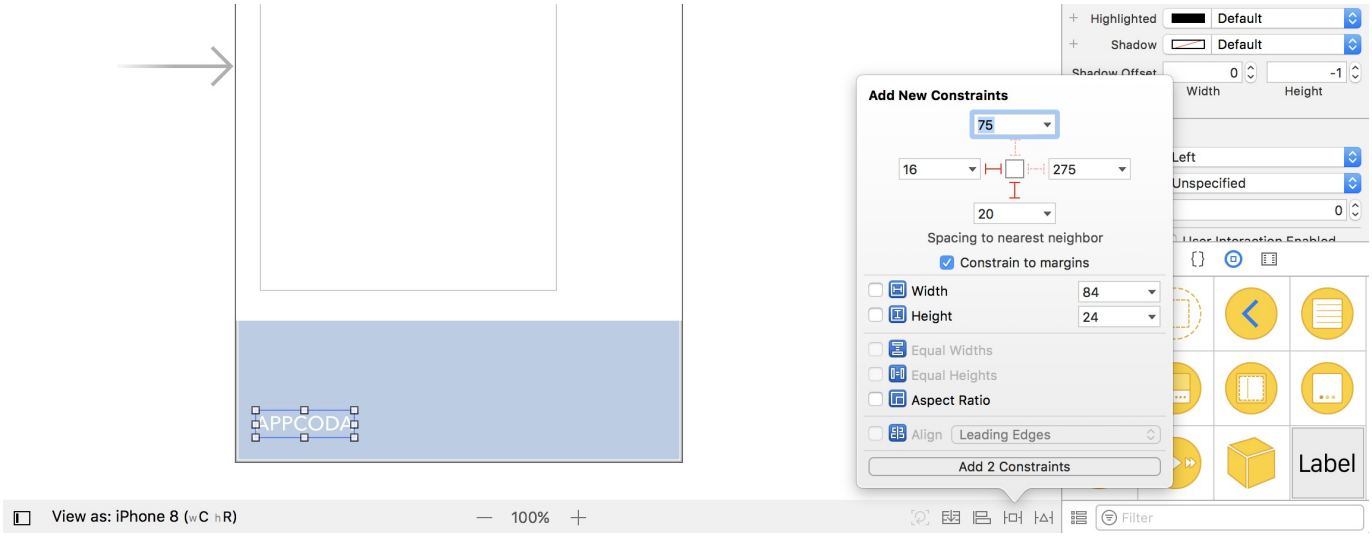


Figure 29.5. Adding constraints for the bottom label

Now let's add a few layout constraints to the collection view. Select the collection view and click the *Align* button of the auto layout bar. 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.

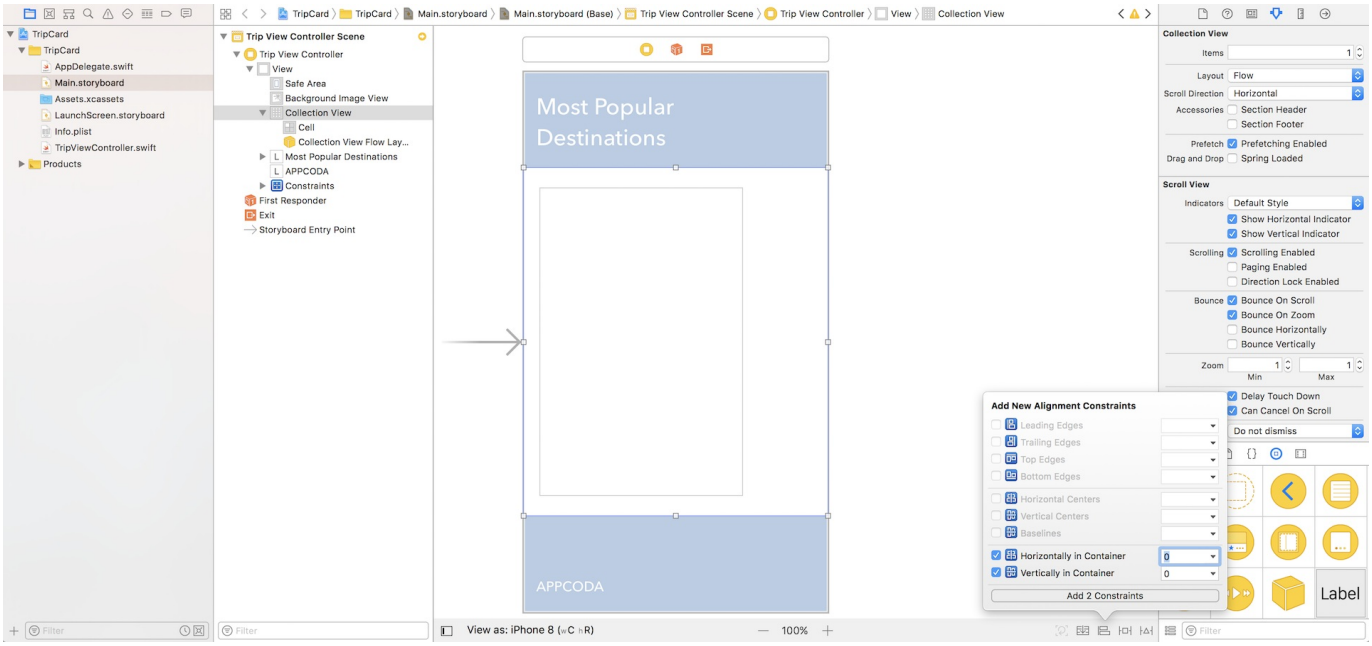


Figure 29.6. Adding alignment constraints to the collection view

Xcode should indicate some missing constraints. Click the *Add New Constraints* button and select the dashed red line corresponding to the top, left and right sides. Uncheck the *Constrain to margins* option and click *Add 3 Constraints*. This ensures that the left and right sides of the collection view align perfectly with the background image view. Also, the collection view is several points away from the title label.

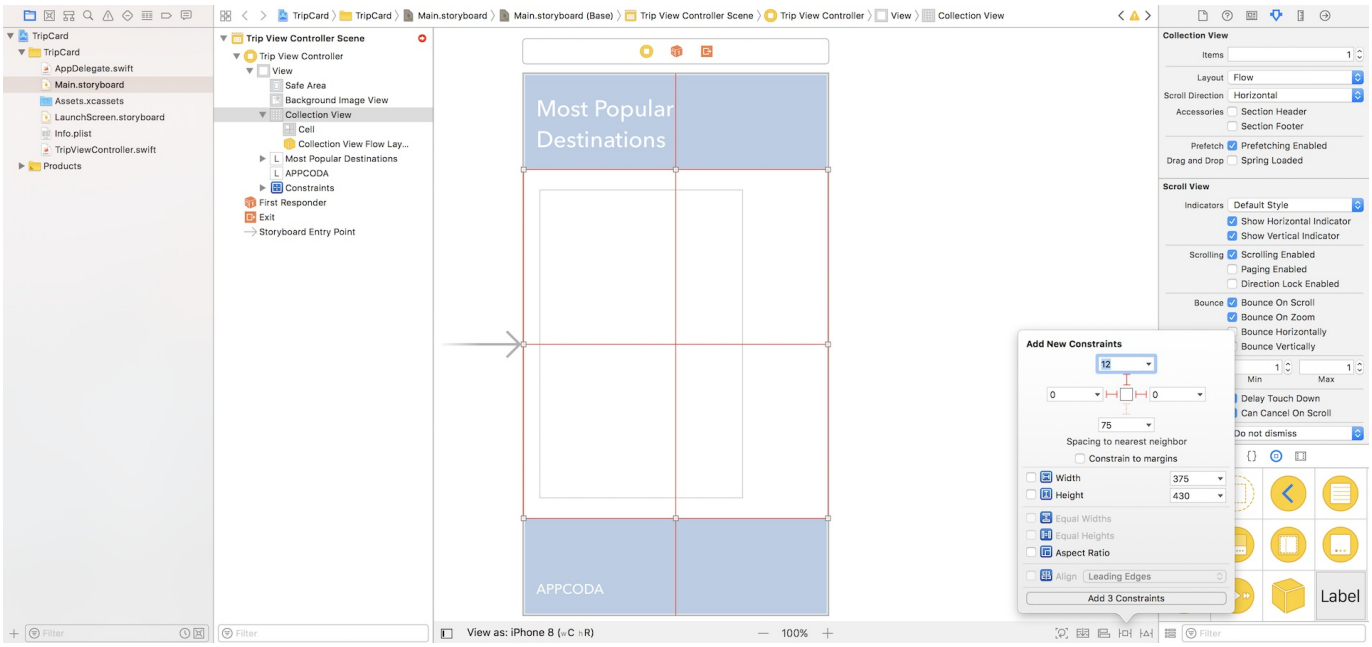


Figure 29.7. Adding spacing constraints to the collection view

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 to Bounds` 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:

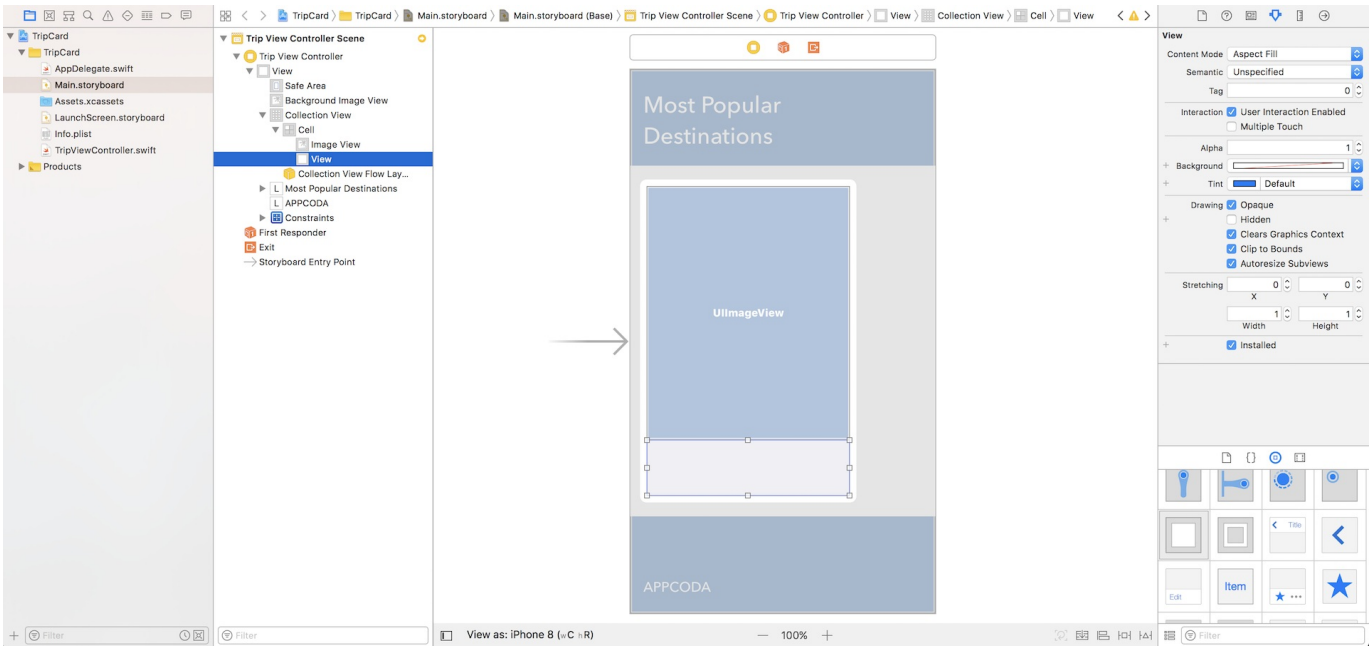


Figure 29.8. The design of the collection view cell

Later, we will change the size of the collection view with reference 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.

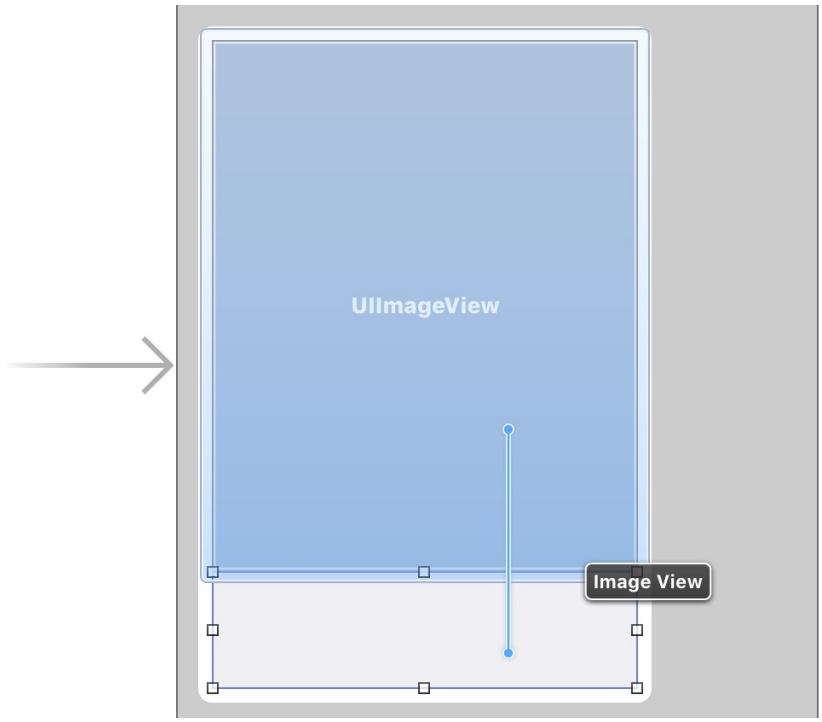


Figure 29.9. Control drag from the image view to the view

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.

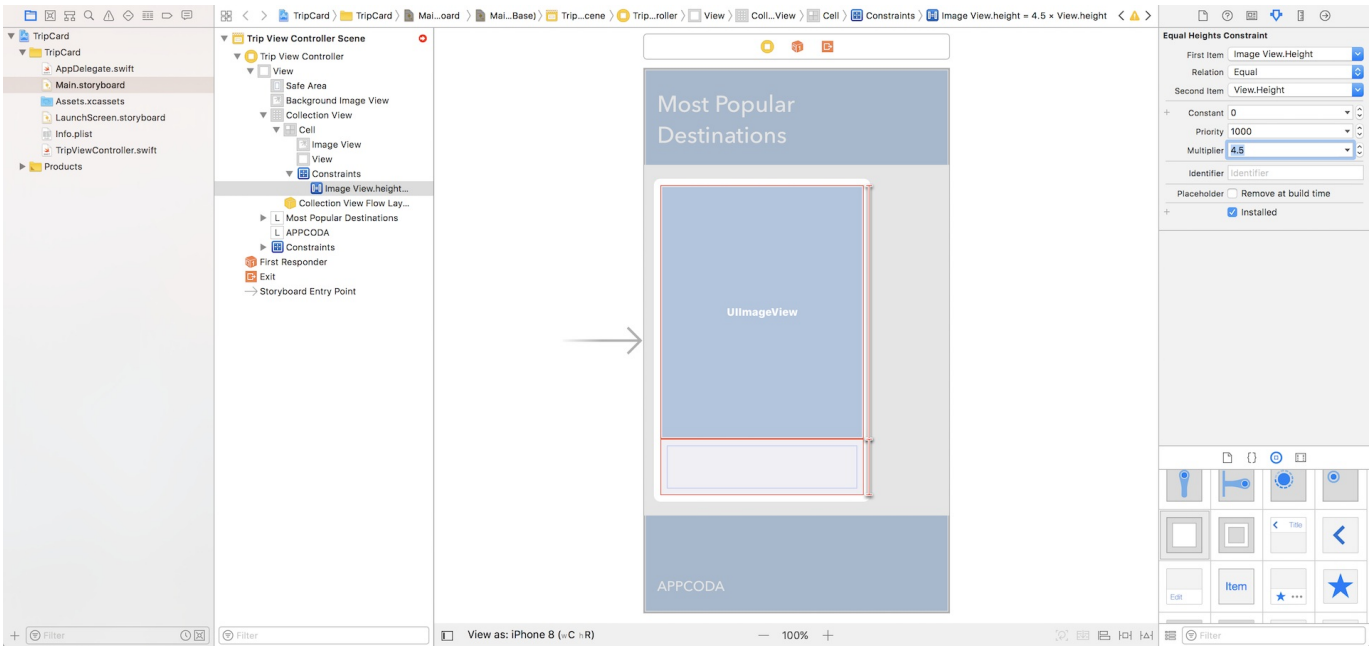


Figure 29.10. Editing the height constraint

Now select the image view and define the spacing constraints. Click the *Add New Constraints* 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 *Add New Constraints* button. Click the dashed red lines that correspond to the left, right and bottom sides.

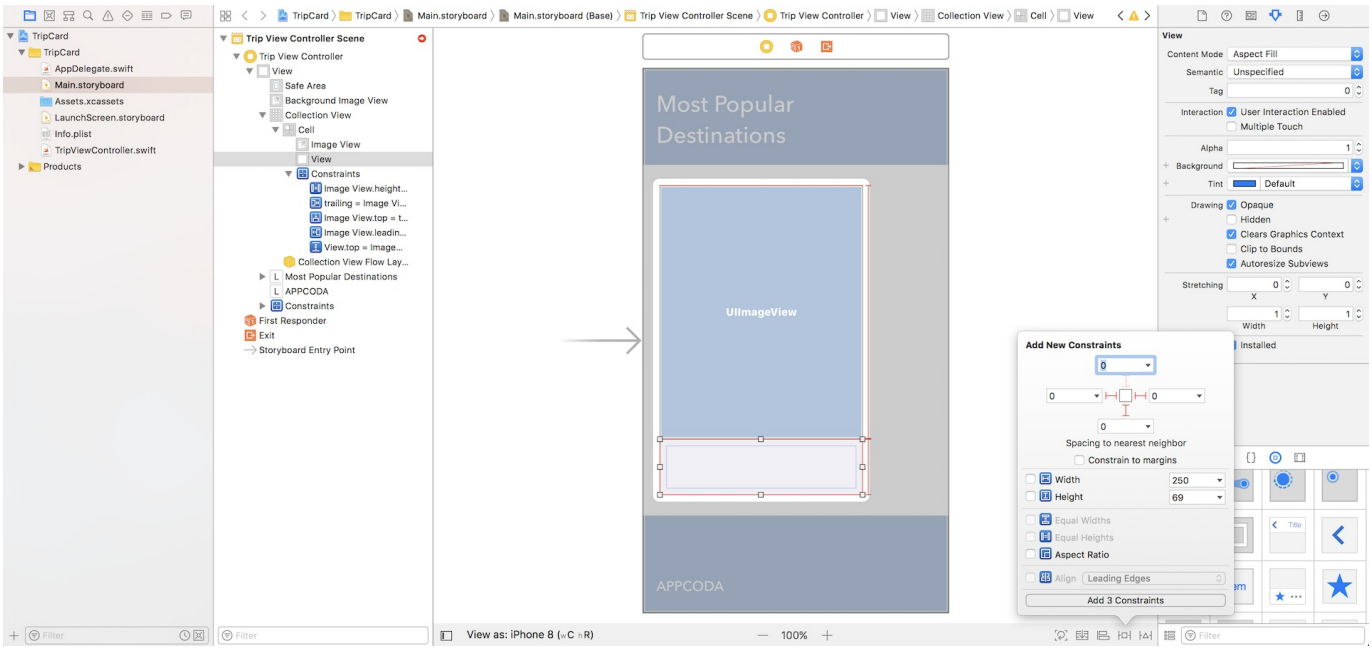


Figure 29.11. Adding spacing constraints for the view in the collection cell

If you follow every step correctly, you've defined all the required constraints for the image view and the internal view. 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.

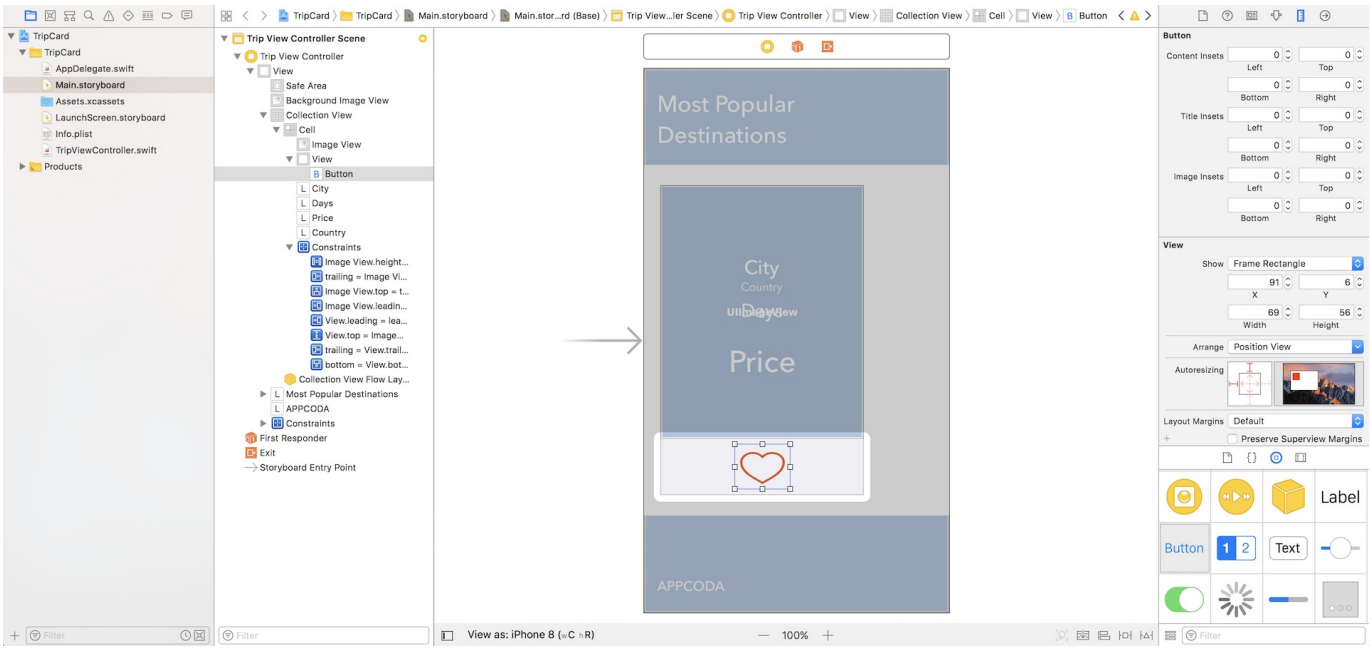


Figure 29.12. Cell design after adding the labels and buttons

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.

Figure 29.13. Control-drag from the City label to the image view to add a couple of layout constraints

In the popover menu, select both *Vertical Spacing* and *Center Horizontally* (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.

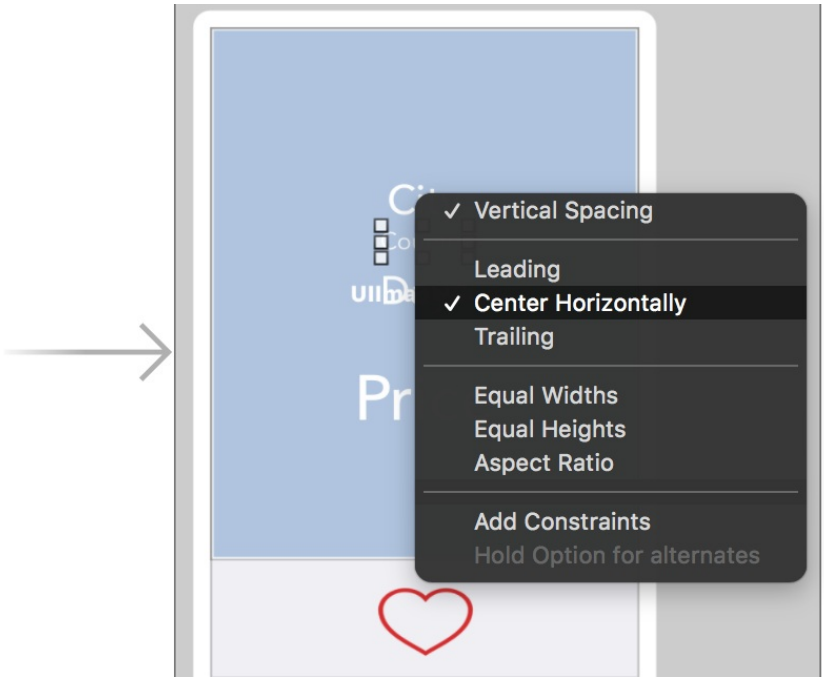


Figure 29.14. Hold the shift key to select multiple layout constraints

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.

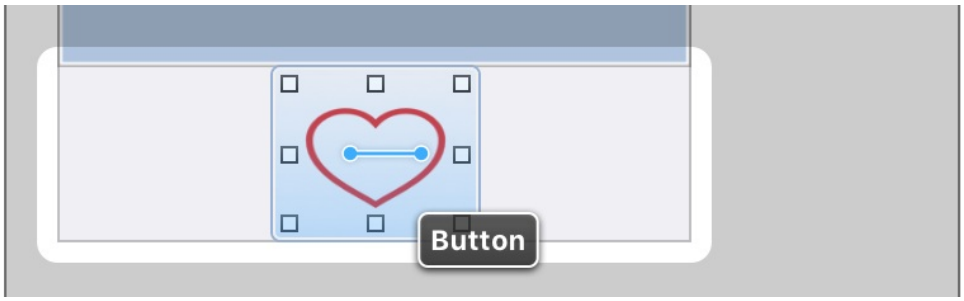


Figure 29.15. Adding size constraints for the heart button

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*.

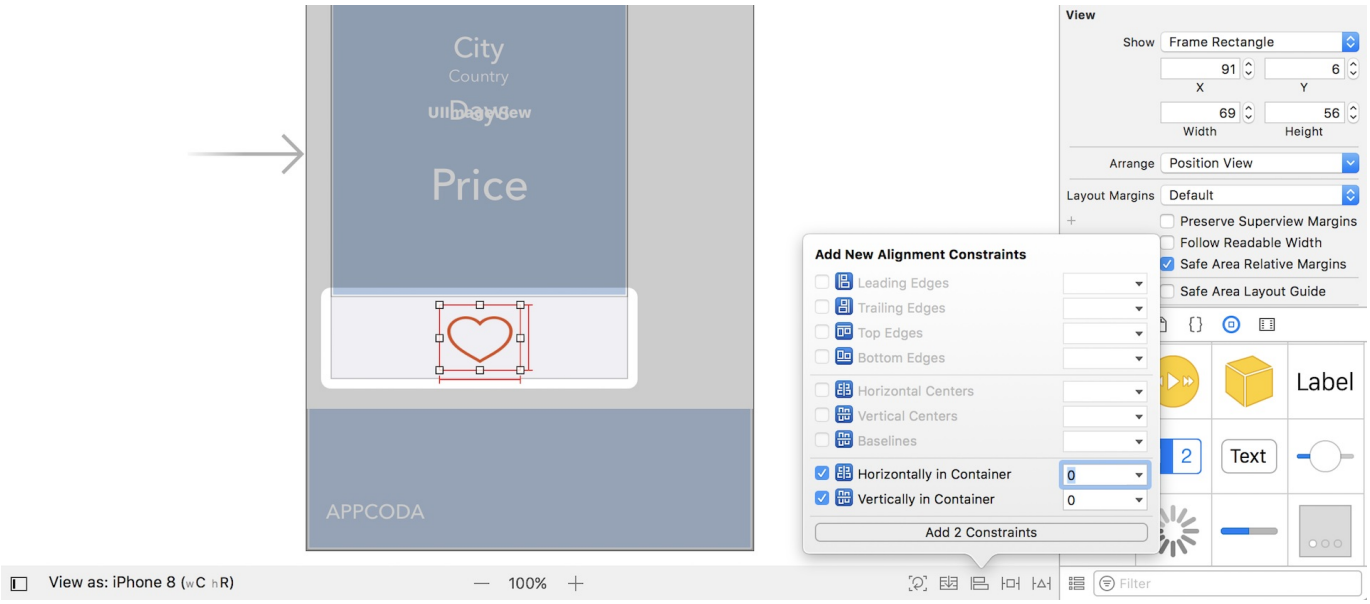


Figure 29.16. Adding alignment constraints for the heart button

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 `TripCollectionViewCell` and set it as a subclass of `UICollectionViewCell`. Once the class is created, open up `TripCollectionViewCell.swift` and update the code to the following:

```
class TripCollectionViewCell: 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"), for: .normal)
            } else {
                likeButton.setImage(UIImage(named: "heart"), for: .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 `TripCollectionViewCell`. Right click the Cell in Document Outline. Connect each of the outlet variables to the corresponding visual

element.

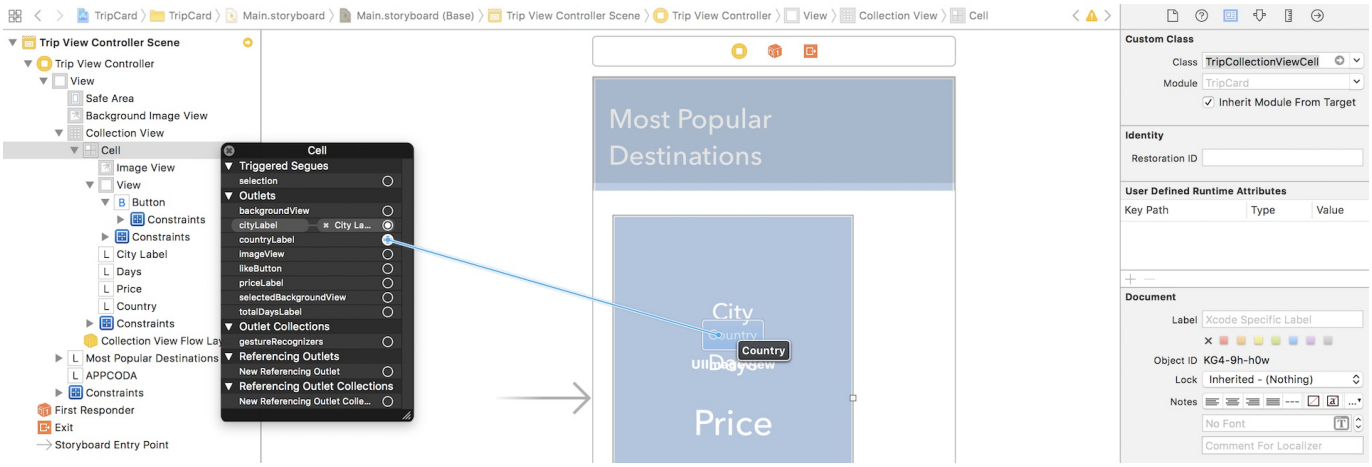


Figure 29.17. Connecting the outlets

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 UIKit

struct Trip {
    var tripId = ""
    var city = ""
    var country = ""
    var featuredImage: UIImage?
    var price:Int = 0
    var totalDays:Int = 0
    var isLiked = false
}
```

The `Trip` structure 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.

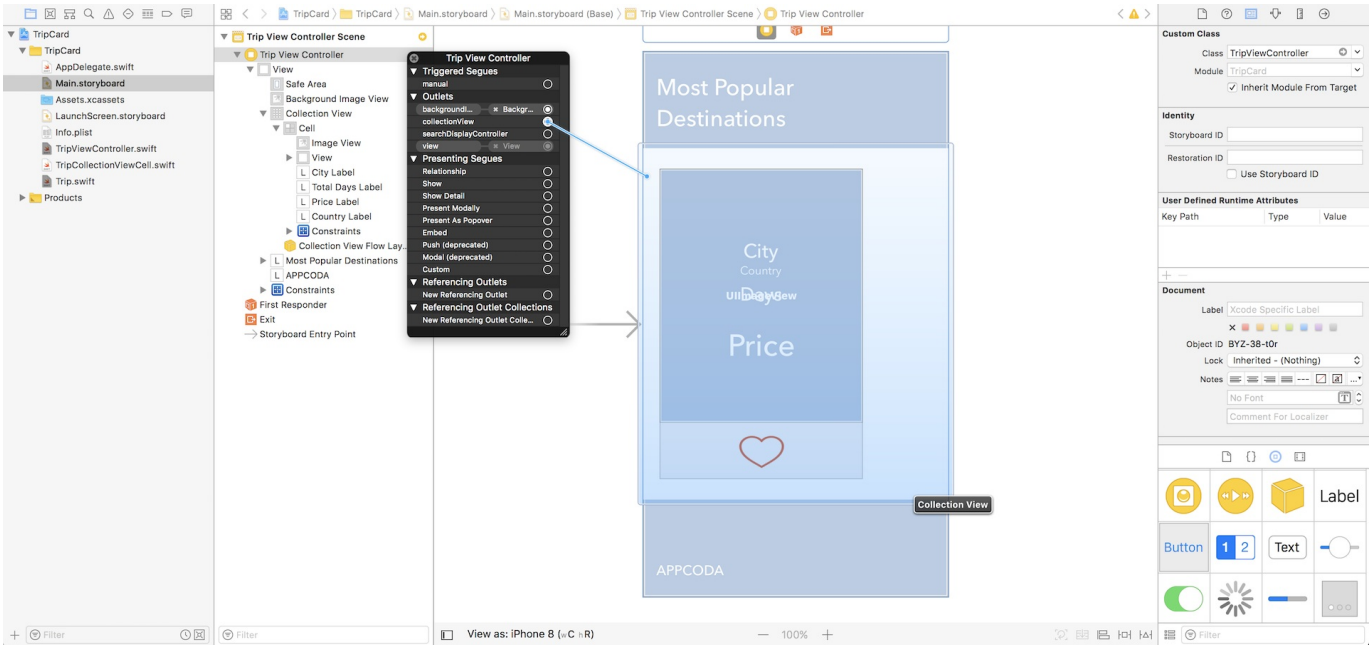


Figure 29.18. Connecting the collection view outlet

Furthermore, control-drag from collection view to trip view controller to connect the data source and delegate.

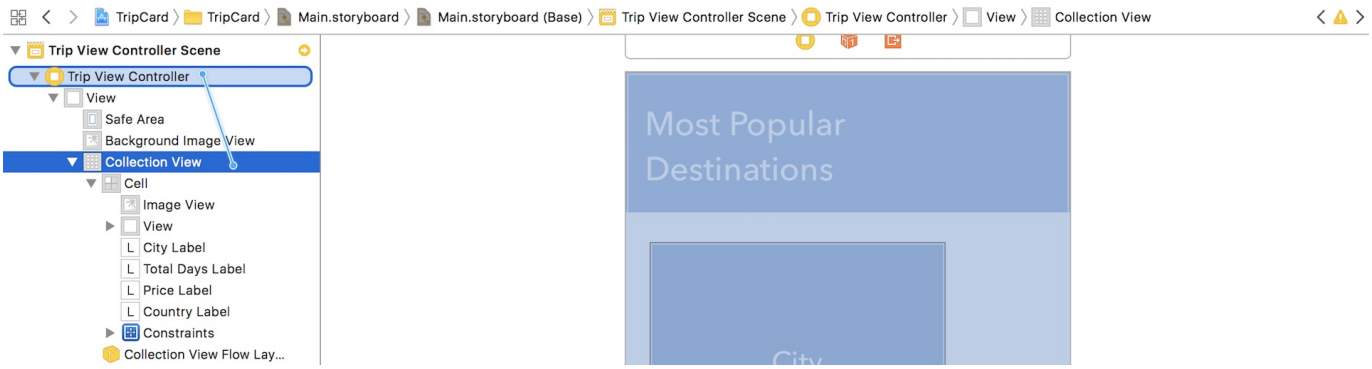


Figure 29.19. Control-drag from the collection view to the trip view controller

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", fe
aturedImage: 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", featuredI
mage: 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", featuredIma
ge: UIImage(named: "santorini"), price: 1800, totalDays: 7, isLiked: false),
    Trip(tripId: "NewYork001", city: "New York", country: "United States", feature
dImage: UIImage(named: "newyork"), price: 900, totalDays: 3, isLiked: false),
    Trip(tripId: "Kyoto001", city: "Kyoto", country: "Japan", featuredImage: UIIma
ge(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. Similar to what we have done before, we will use an extension to adopt these protocols:

```
extension TripViewController: UICollectionViewDelegate, UICollectionViewDataSource
{
    func numberOfSections(in collectionView: UICollectionView) -> Int {
        return 1
    }

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

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPat
h: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "Cell",
for: indexPath) as! TripCollectionViewCell

        // 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.clear
```

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 4.7-inch display. If you run the app on iPhone SE, however, parts of the collection view are blocked.

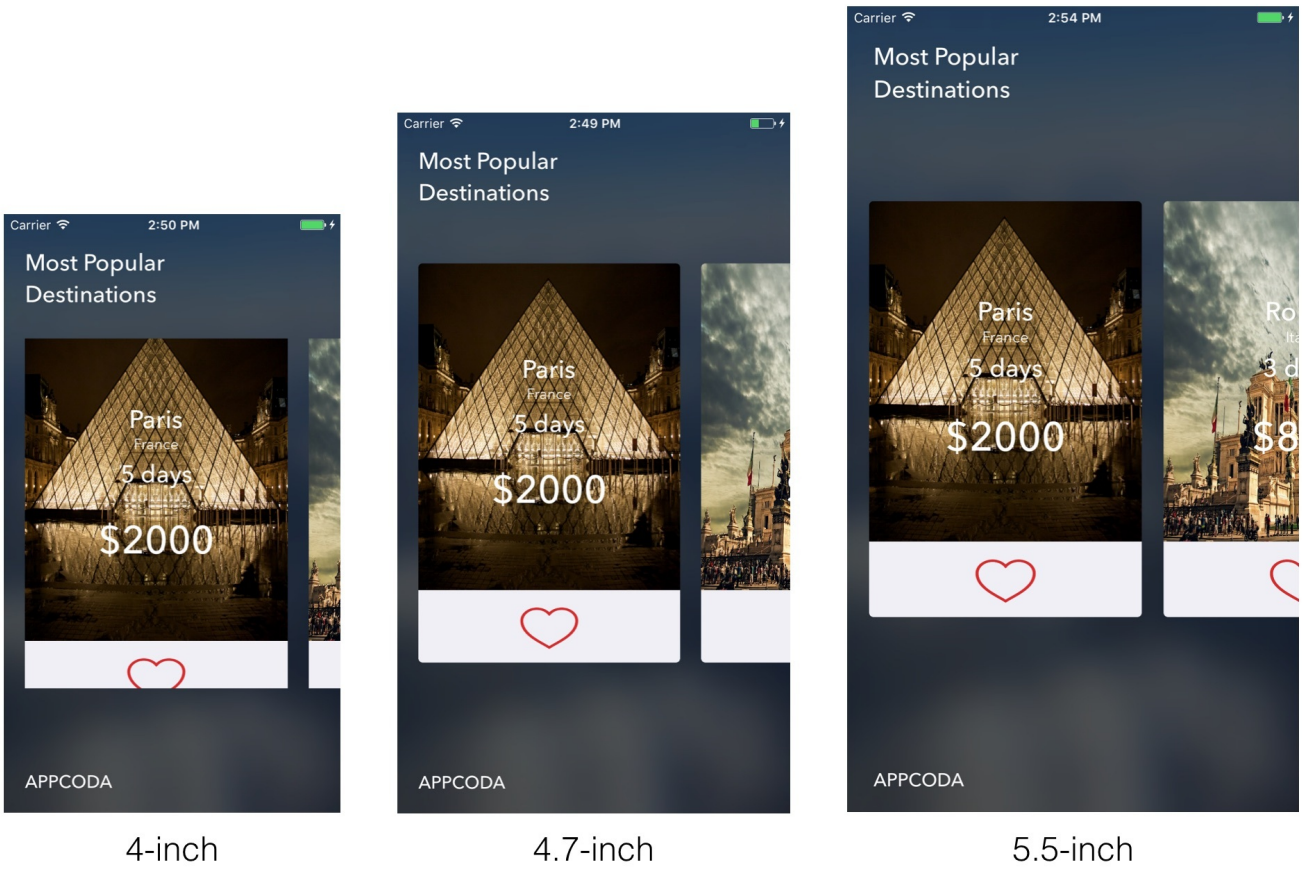


Figure 29.20. The demo app on different screen sizes

We have to reduce the height of the collection view for 4-inch devices. Insert the following block of code in the `viewDidLoad` method of `TripViewController.swift`:

```
if UIScreen.main.bounds.size.height == 568.0 {
    let flowLayout = self.collectionView.collectionViewLayout as! UICollectionView
FlowLayout
    flowLayout.itemSize = CGSize(width: 250.0, height: 330.0)
}
```

Base on the screen height, we can deduce if the device has a 4-inch screen. If it meets the criteria, we adjust the height of the collection view from `380` points to `330` points. Once you have made the change, try to test the app on iPhone SE 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 use a delegate pattern to do the data passing. First, define a new protocol named `TripCollectionCellDelegate` in the `TripCollectionViewCell` class:

```
protocol TripCollectionCellDelegate {
    func didLikeButtonPressed(cell: TripCollectionViewCell)
}
```

Next, 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(cell: self)
}
```

Now go back to the storyboard to associate the heart button with this method. Select the heart button and go to the Connection inspector. Drag from *Touch Up Inside* to the Cell in the Document Outline. Select `likeButtonTappedWithSender:` when the popover appears.

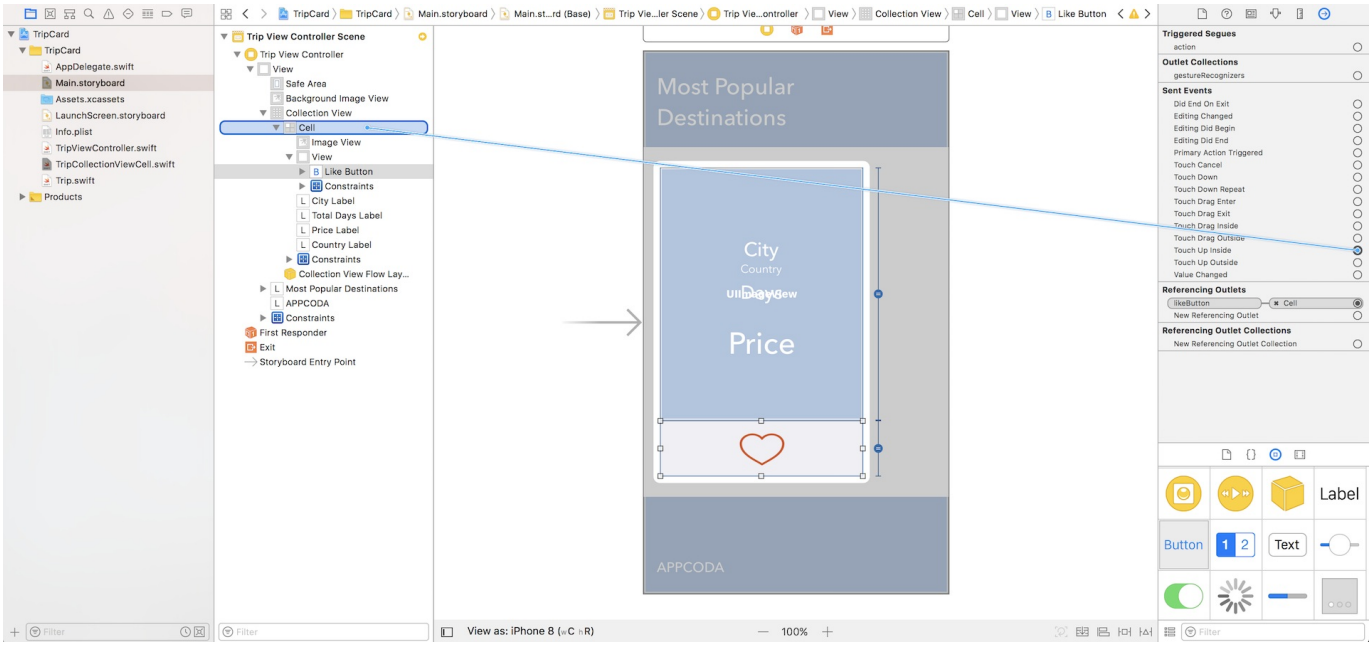


Figure 29.21. Connecting the Heart button with the action method

Now open `TripViewController.swift`. It is the object that adopts the `TripCollectionCellDelegate` protocol. Let's create an extension to implement the protocol:

```
extension TripViewController: TripCollectionCellDelegate {
    func didLikeButtonPressed(cell: TripCollectionViewCell) {
        if let indexPath = collectionView.indexPath(for: 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 `indexPath(for:)` method and toggle the status of `isLiked` accordingly.

Recall that we have defined a `didSet` observer for the `isLiked` property of `TripCollectionViewCell`. 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 `collectionView(_:cellForItemAt:)` 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.

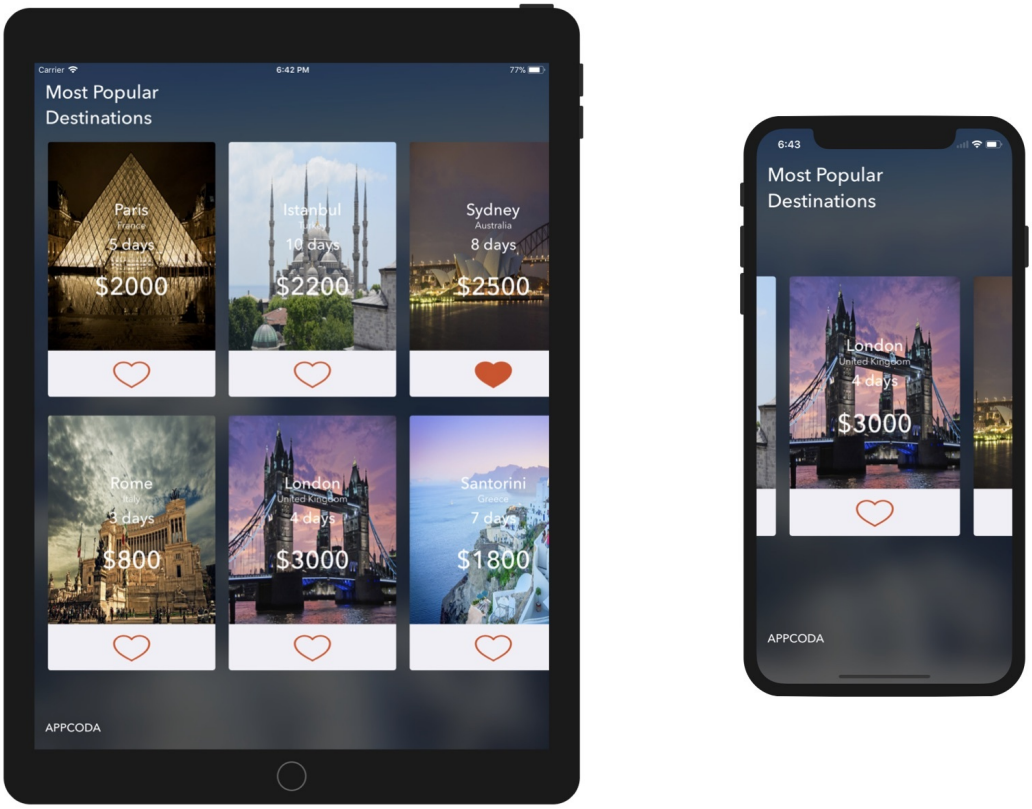
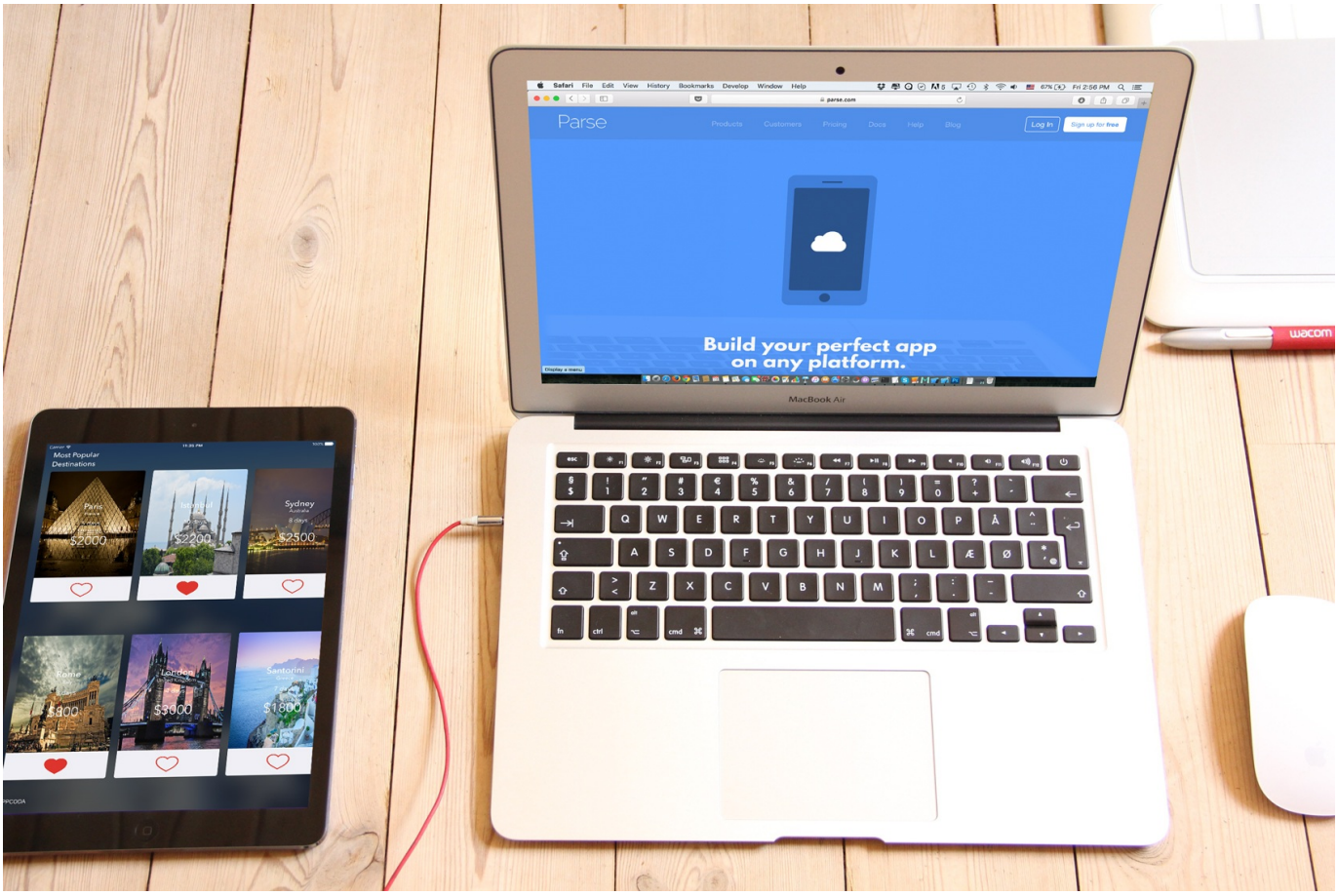


Figure 29.22. Tapping the heart button to bookmark the trip

For reference, you can download the final project from http://www.appcoda.com/resources/swift5/TripCard.zip.

Chapter 30
Working with Parse as Mobile Backends



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 13 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 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.

If you haven't heard of Parse, it is better to understand its history.

Parse was acquired by Facebook in late April 2013. Since then, it has grown into one of the most popular mobile backends. Unfortunately, Facebook considered to shut down the service and no longer provides the Parse cloud to developers. For now, you can still use Parse as your mobile backend. It comes down to these two solutions:

1. **Install and host your own Parse servers** - Although the Parse's hosted service will be retired on January 28, 2017, Facebook released an open source version of the Parse backend called Parse Server. Now everyone can install and host their own Parse servers on AWS and Heroku. The downside of this approach is that you will have to manage the servers yourself. For indie developers or those who do not have any backend management experience, this is not a perfect option.

2. **Use Parse hosting service** - Some companies such as SashiDo.io and Back4App now offers managed Parse servers. In other words, they help you install the Parse servers, and host them for you. You do not need to learn AWS/Heroku or worry about the server infrastructure. These companies just manage the Parse cloud servers for you. It is very similar to the Parse hosted backend provided by Facebook but delivered by third-party companies. In this tutorial, I will use Back4App's Parse hosting service, simply because it is free to use. After you understand how

In this chapter, I will walk you through the integration process of Parse using Back4app. 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 http://www.appcoda.com/resources/swift5/ParseDemoStarter.zip.

If you haven't read chapter 29, I highly recommend you to check it out first. It will be better to have some basic understandings of the demo app before you move on.

I hope I have made everything clear. Let's get started.

Creating Your App on Parse

First, you have to sign up for a free account on http://back4app.com. Once you sign up the account, you'll be brought to a dashboard. From there, click the `Build new Parse app` button to create a new application. Simply use `TripCard` as the app name and click *Create*.

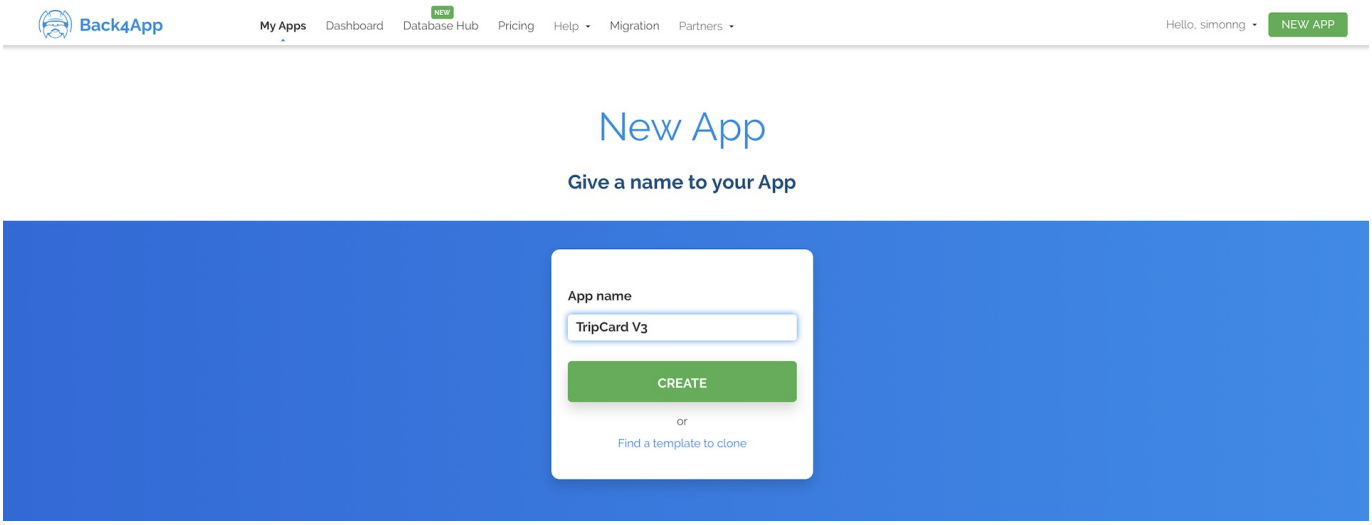


Figure 30.1. Back4app - Creating a new application

Once the app is created, you will be brought to the main screen in which you can find all the available features of Back4App. Like the Parse cloud, Back4app offers various backend services including data and push notification.

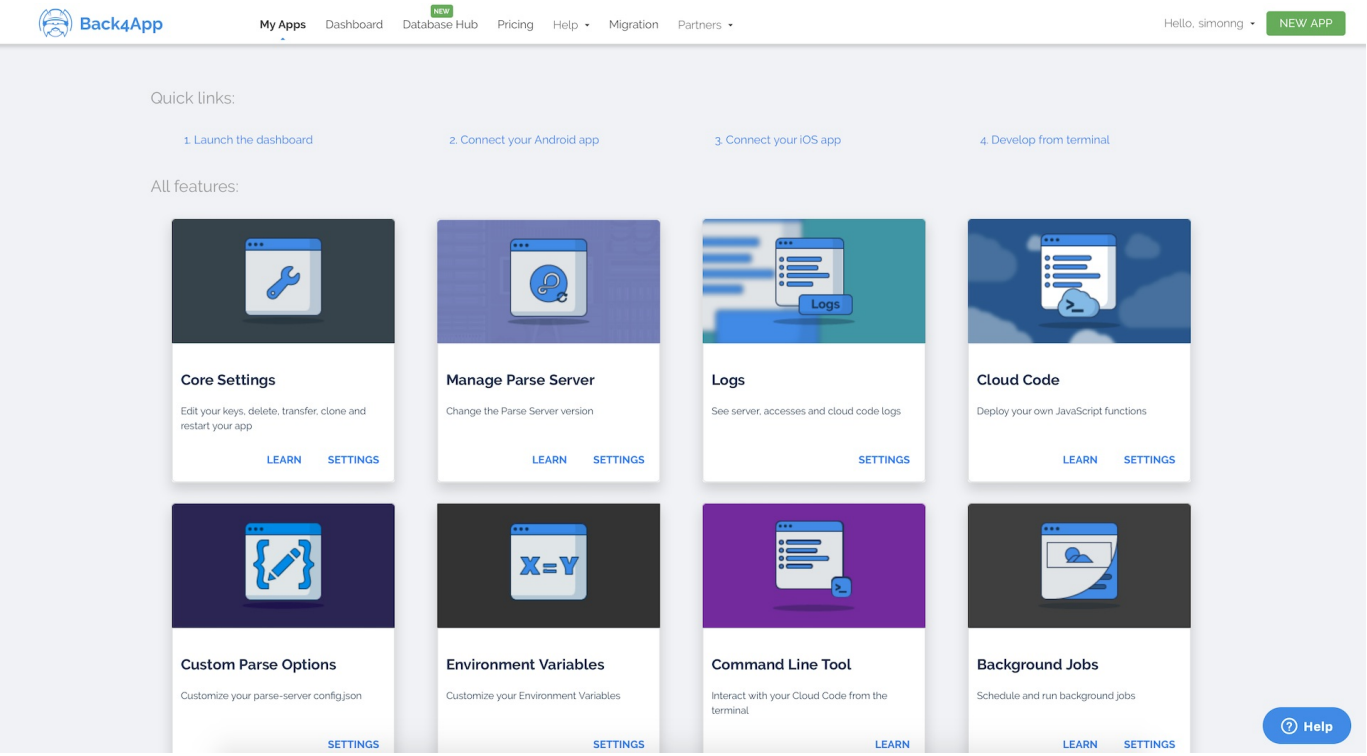


Figure 30.2. Your Parse app - main screen

What we will focus on in this chapter is the data service. To manage the data of your Parse app, click *Launch the dashboard* to access the Parse dashboard.

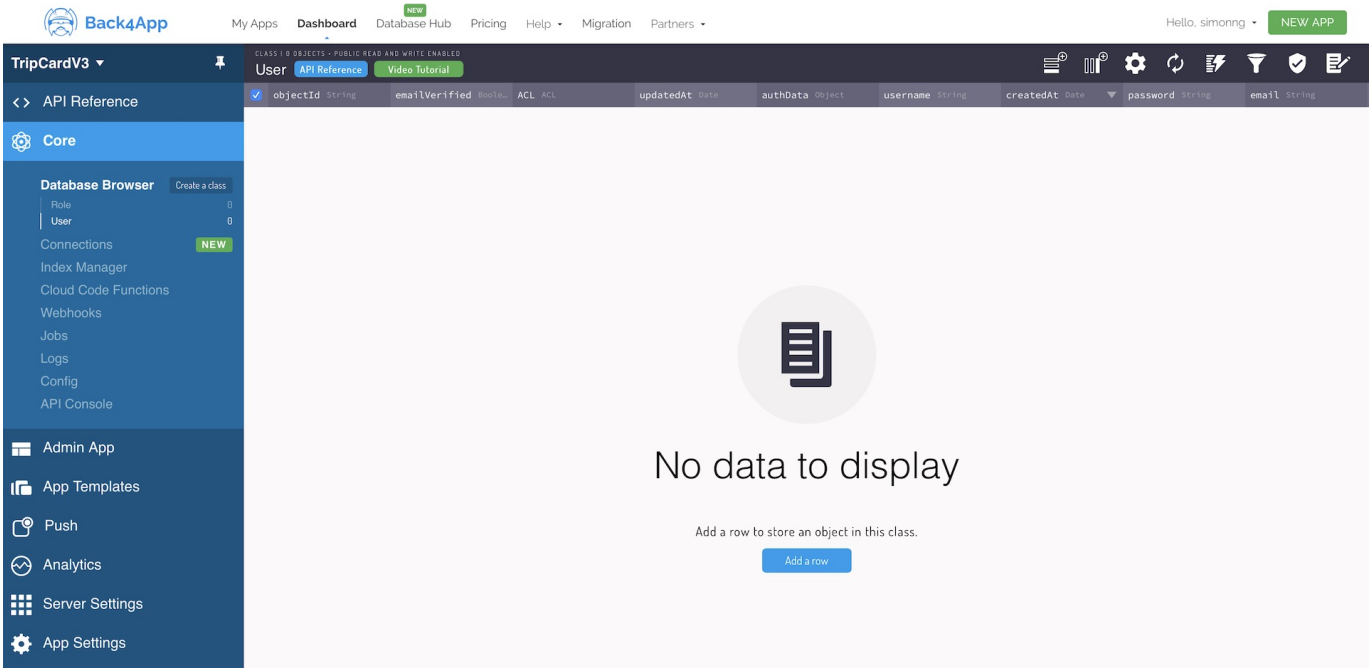


Figure 30.3. The Parse Dashboard

Setting up Your Data

The Parse dashboard lets developers manage their Parse app and data in a graphical UI. By default, the data browser shows no data. It is quite obvious because 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 `Create a class` button on the side menu to create a new class. Set the name to `Trip` and type to `Custom`, and then click `Create class` to proceed. Once created, you should see the class under the *Browser* section of the sidebar menu.

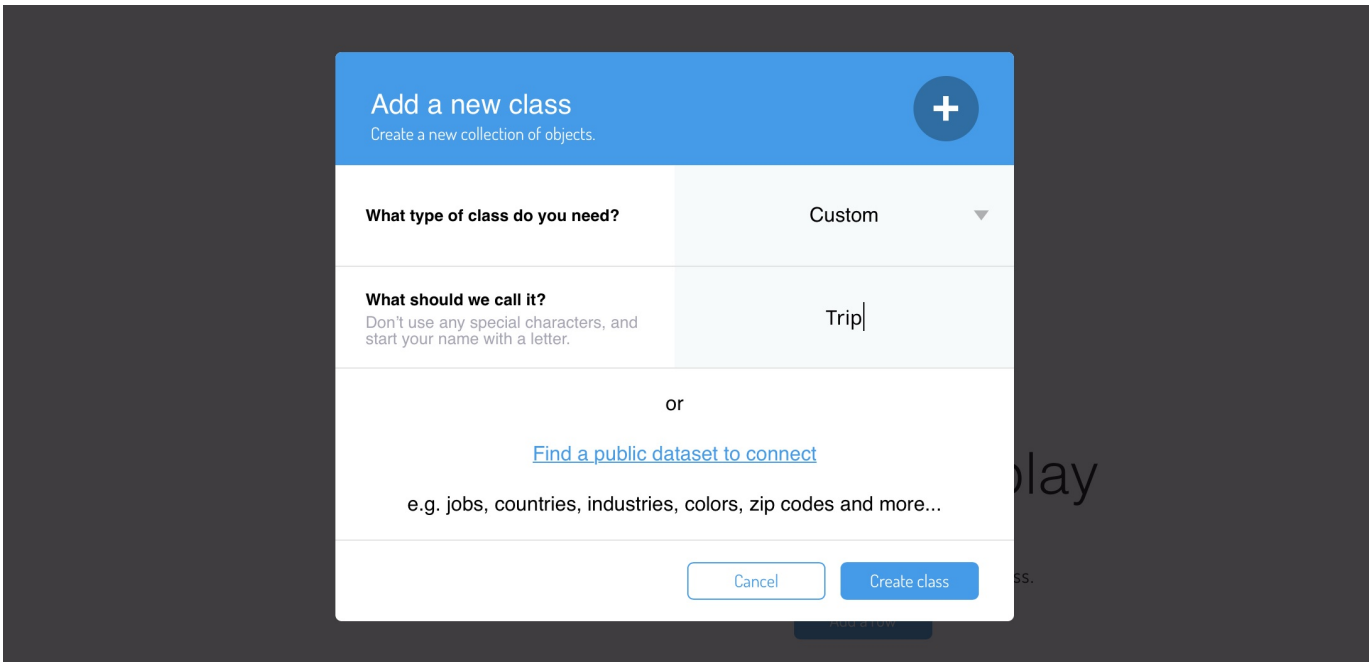


Figure 30.4. Creating a new class in the Parse app

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 `Add a new column` button to add a new column.

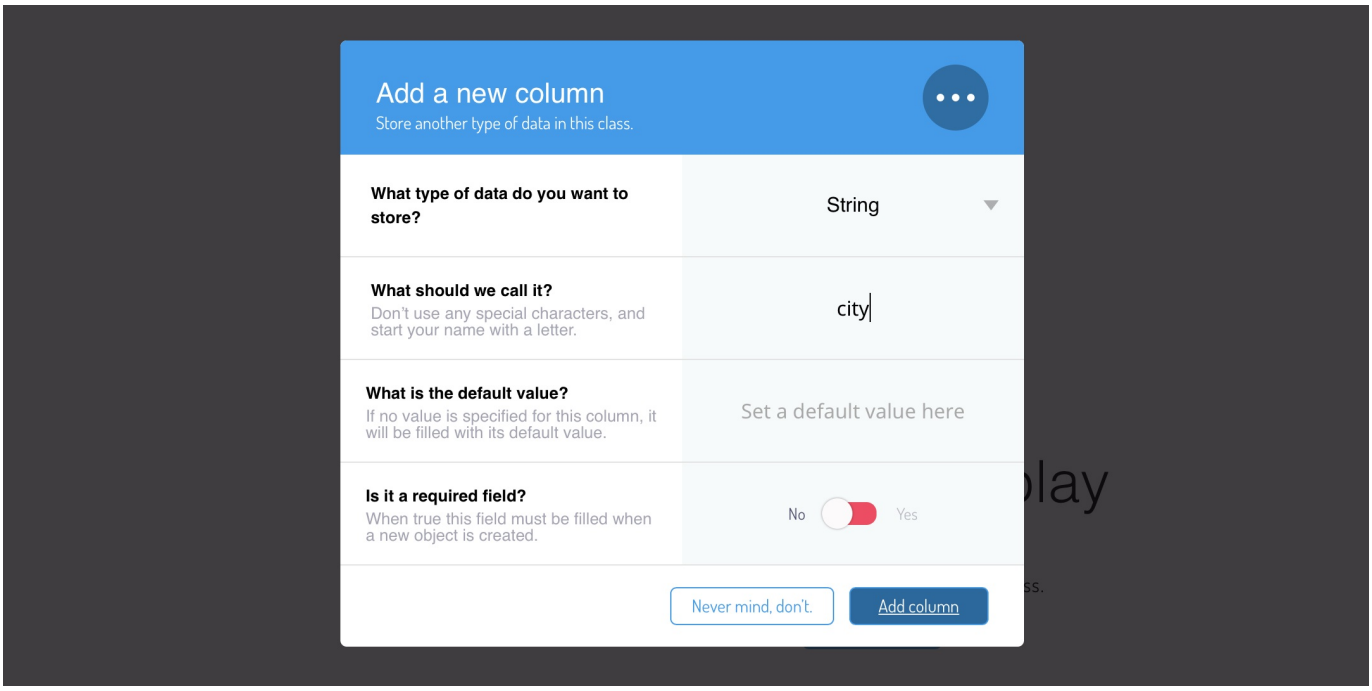


Figure 30.5. Adding a new column

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`, type to `Boolean`, and the default value to `false`.

Once you have added the columns, your table should look similar to the screenshot below.

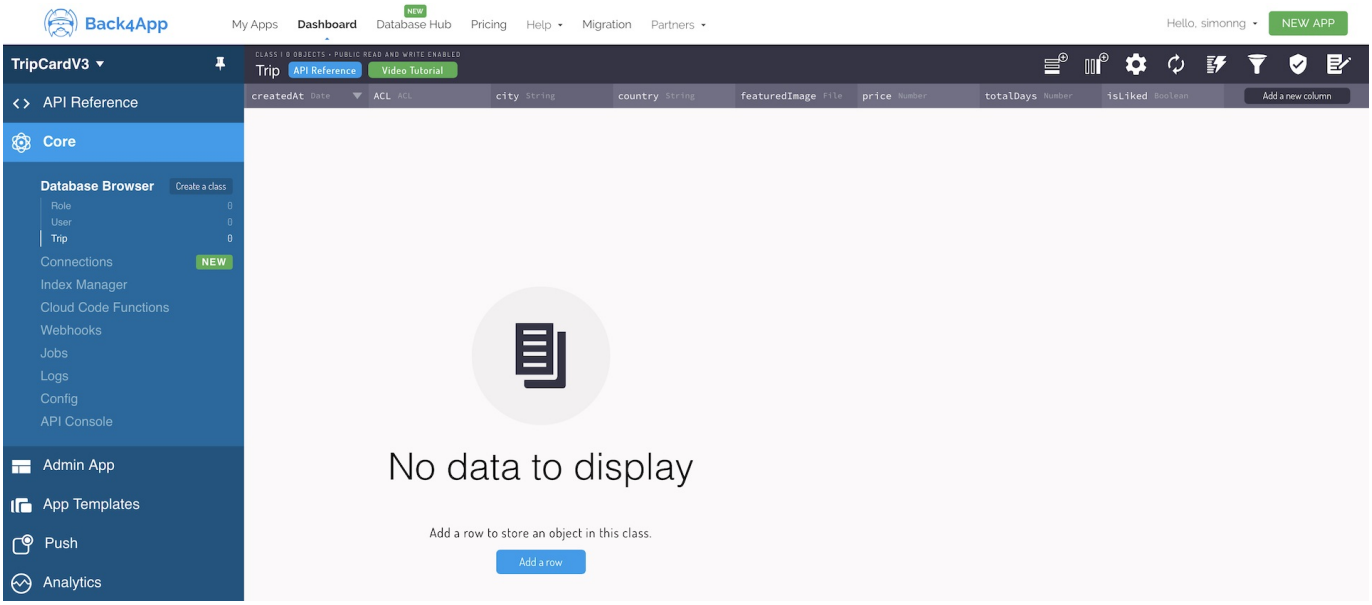


Figure 30.6. New columns added to the Trip class

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 `Add a 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", fe
aturedImage: UIImage(named: "paris"), price: 2000, totalDays: 5, isLiked: false),
                     Trip(tripId: "Rome001", city: "Rome", country: "Italy", featu
redImage: UIImage(named: "rome"), price: 800, totalDays: 3, isLiked: false),
                     Trip(tripId: "Istanbul001", city: "Istanbul", country: "Turke
y", featuredImage: UIImage(named: "istanbul"), price: 2200, totalDays: 10, isLiked
: false),
                     Trip(tripId: "London001", city: "London", country: "United Ki
ngdom", featuredImage: UIImage(named: "london"), price: 3000, totalDays: 4, isLike
d: false),
                     Trip(tripId: "Sydney001", city: "Sydney", country: "Australia"
, featuredImage: UIImage(named: "sydney"), price: 2500, totalDays: 8, isLiked: fal
se),
                     Trip(tripId: "Santorini001", city: "Santorini", country: "Gre
ece", featuredImage: UIImage(named: "santorini"), price: 1800, totalDays: 7, isLik
ed: false),
                     Trip(tripId: "NewYork001", city: "New York", country: "United
 States", featuredImage: UIImage(named: "newyork"), price: 900, totalDays: 3, isLi
ked: false),
                     Trip(tripId: "Kyoto001", city: "Kyoto", country: "Japan", fea
turedImage: 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:

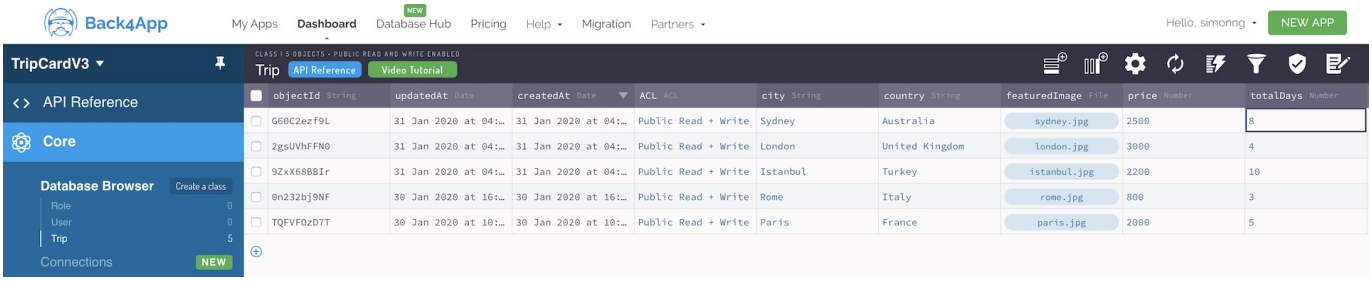


Figure 30.7. Adding the first data item

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.

> **Note:** You can find the images in the *TripCard/Assets.xcassets* folder of the ParseDemo project.

Repeat the above procedures and add the rest of the trip data. You will end up with a screen similar to this:

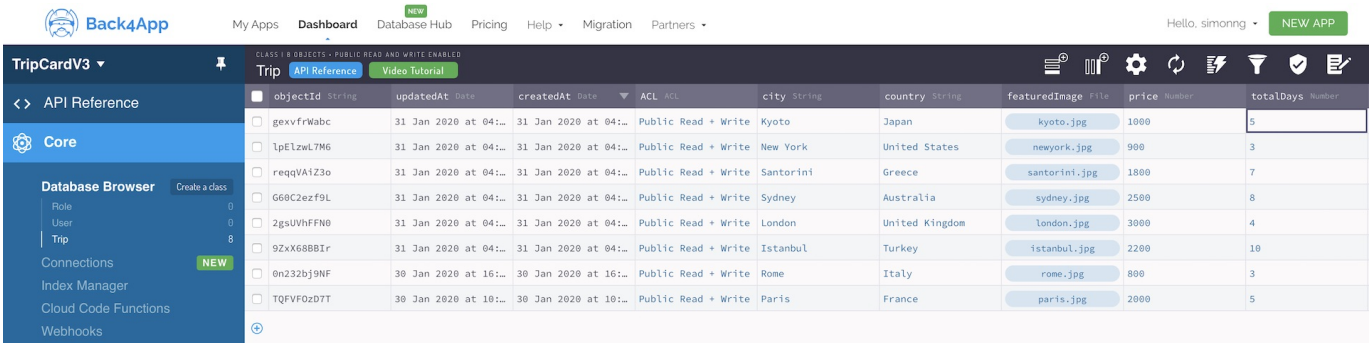


Figure 30.8. Trip data in the data browser

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. The very first thing to do is to install the Parse SDK. You can either install it using CocoaPods (which is the recommended way) or manually.

Using CocoaPods

Assuming you have CocoaPods installed on your Mac, open Terminal app and change to your *TripCard* project folder. Type `pod init` to create the Podfile and edit it like this:

```
target 'TripCard' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for TripCard
  pod 'Parse'

end
```

To install the Parse SDK using CocoaPods, you just need to specify the `Parse` pod in the configuration file. Save the file, go back to Terminal and type:

```
pod install
```

CocoaPods should automatically download the Parse SDK and all the required libraries.

Manual Installation

For some reasons, if you prefer to install the SDK manually, you can download the Parse SDK for iOS from https://github.com/parse-community/Parse-SDK-iOS-OSX/releases/download/1.17.3/Parse-iOS.zip. 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.

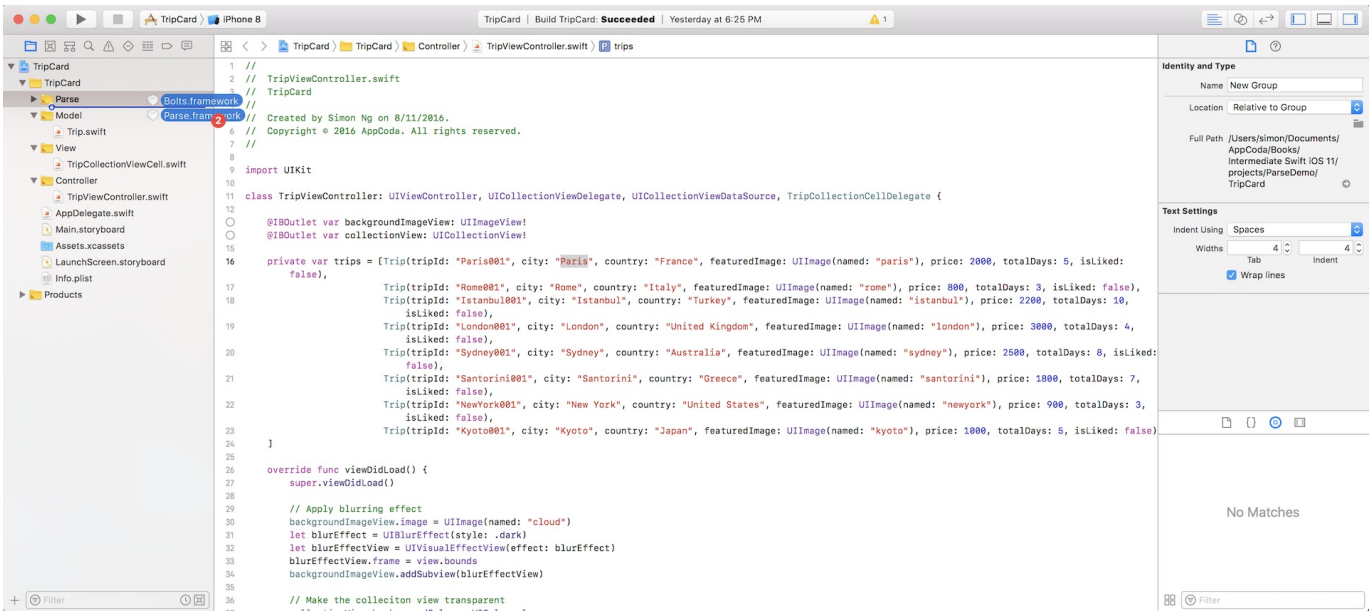


Figure 30.9. Adding the frameworks to the TripCard project

The Parse SDK depends on other frameworks in iOS SDK. You will need to add the following libraries to the project:

- AudioToolbox.framework
- CFNetwork.framework
- CoreGraphics.framework
- CoreLocation.framework
- MobileCoreServices.framework
- QuartzCore.framework
- Security.framework
- StoreKit.framework
- SystemConfiguration.framework
- libz.tbd
- libsqlite3.tbd

  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.

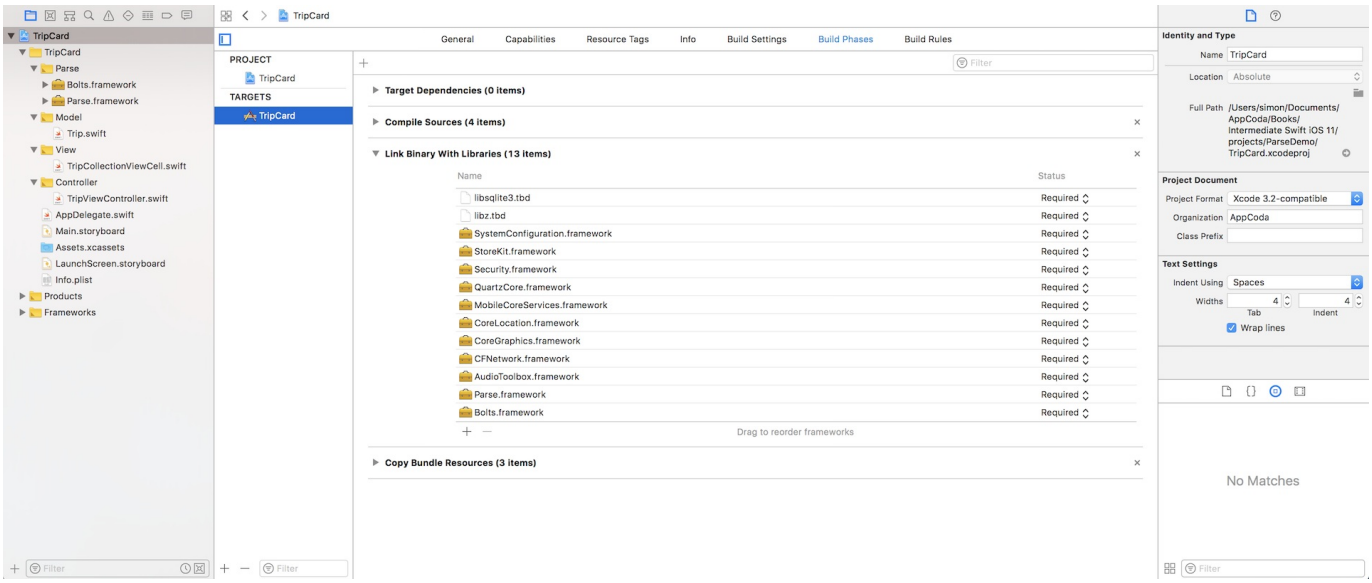


Figure 30.10 Adding the required libraries to the project

Connecting with Parse

To access your app data on Parse, you first need to find out the Application Key and the Client Key. In the dashboard, choose *Security & Keys* under App Settings.

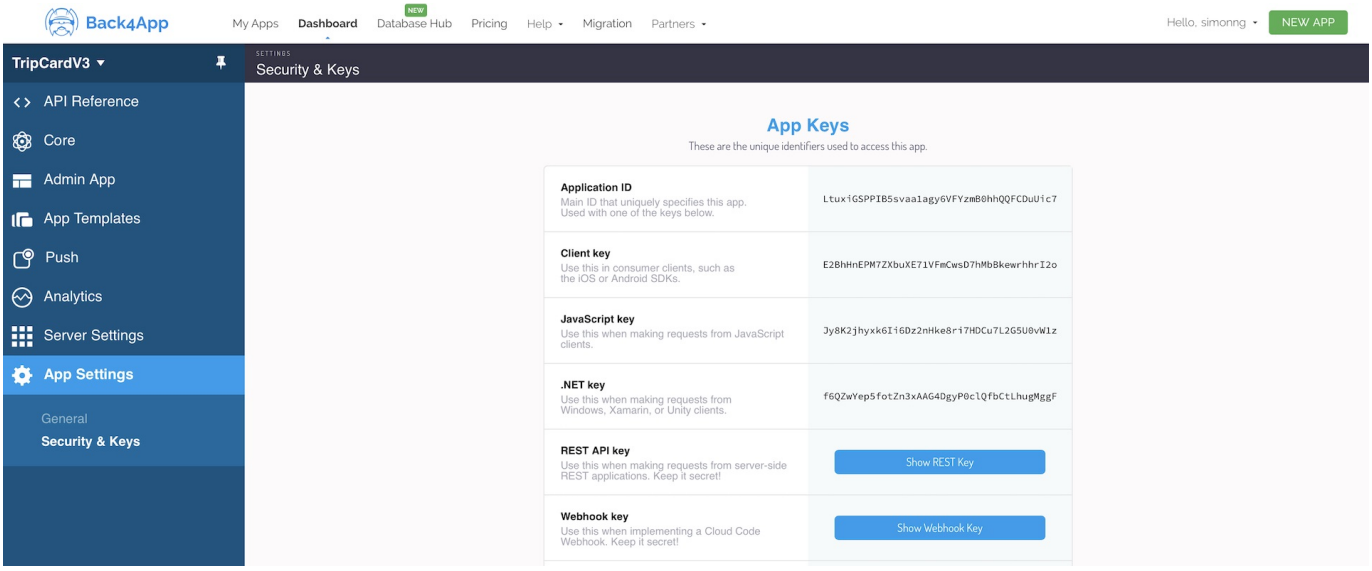


Figure 30.11. Revealing the App Id and Client Key

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.swift` and add an import statement at the very beginning to import the Parse framework:

```
import Parse
```

Next, add the following code in the `application(_:didFinishLaunchingWithOptions:)` method to initialize Parse:

```
// Initialize Parse.
let configuration = ParseClientConfiguration {
    $0.applicationId = "dKvSyQFjk9U2vrWgUT7tYrrMkAaWZcI9i0HDgCjP"
    $0.clientKey = "PQkB57nWFnVI5x45IzSixJcSqL3SsBzXhnnuIfHZ"
    $0.server = "https://parseapi.back4app.com"
}
Parse.initialize(with: configuration)
```

Note that you should replace the Application ID and the Client Key with your own keys. With just a couple lines 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

Now we are ready to modify the TripCard app to pull data from the Parse cloud. We'll first 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.findObjectsInBackground { (objects, error) in
    if let error = error {
        print("Error: \(error) \(error.localizedDescription)")
        return
    }

    if let objects = objects {
        // Do something
    }
}
```

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 `findObjectsInBackground` 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` structure (i.e. `Trip.swift`), you may notice that the `featuredImage` property is of the type `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 will 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 `PFFileObject`. Now open `Trip.swift` and update it to the following:

```
import UIKit
import Parse

struct Trip {
    var tripId = ""
    var city = ""
    var country = ""
    var featuredImage: PFFileObject?
    var price:Int = 0
    var totalDays:Int = 0
    var isLiked = false

    init(tripId: String, city: String, country: String, featuredImage: PFFileObject
!, 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? PFFileObject
        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 helper method called `toPFObject`. In the 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 insert the following import statement:

```
import Parse
```

Then add the following method:

```
func loadTripsFromParse() {
    // Clear up the array
    trips.removeAll(keepingCapacity: true)
    collectionView.reloadData()

    // Pull data from Parse
    let query = PFQuery(className: "Trip")
    query.findObjectsInBackground { (objects, error) -> Void in

        if let error = error {
            print("Error: \(error) \(error.localizedDescription)")
            return
        }

        if let objects = objects {
            for (index, object) in objects.enumerated() {
                // Convert PFObject into Trip object
                let trip = Trip(pfObject: object)
                self.trips.append(trip)

                let indexPath = IndexPath(row: index, section: 0)
                self.collectionView.insertItems(at: [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 the Parse cloud 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 `insertItems(at:)` method.

For the `collectionView(_:cellForItemAt:)` method, you will need to change the following line of code from:

```
cell.imageView.image = trips[indexPath.row].featuredImage
```

to:

```
// Load image in background
cell.imageView.image = UIImage()
if let featuredImage = trips[indexPath.row].featuredImage {
    featuredImage.getDataInBackground(block: { (imageData, error) 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 the background. Parse stores files (such as images, audio, and documents) in the cloud in the form of `PFFile`. We use `PFFile` to reference the featured image. The class provides the `getDataInBackground` 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.

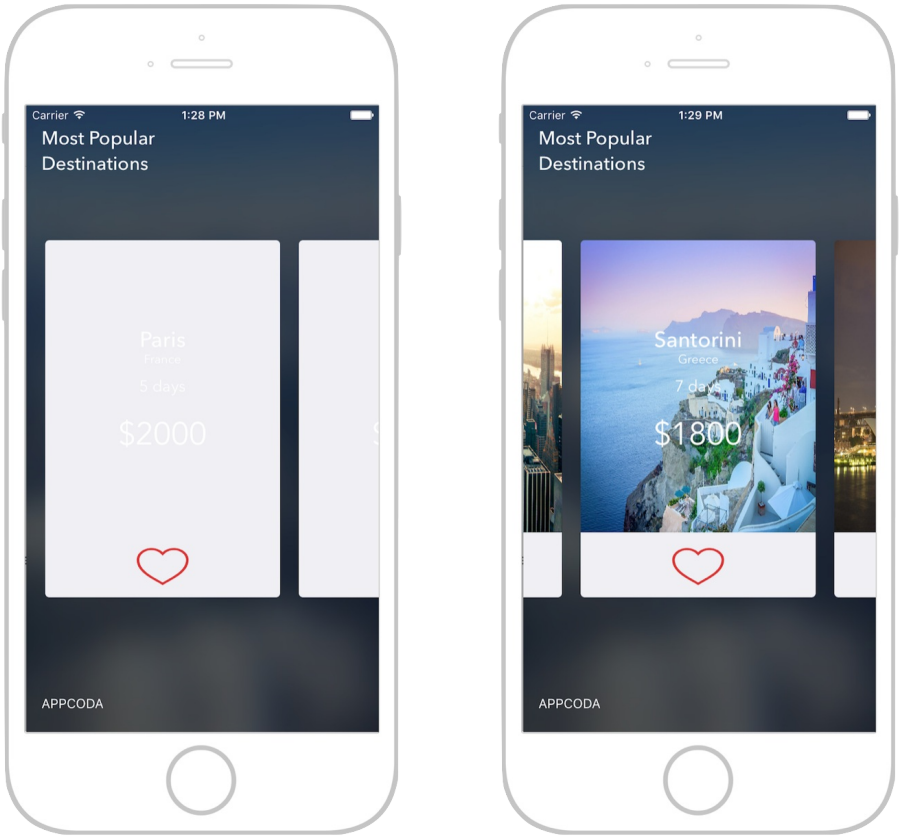
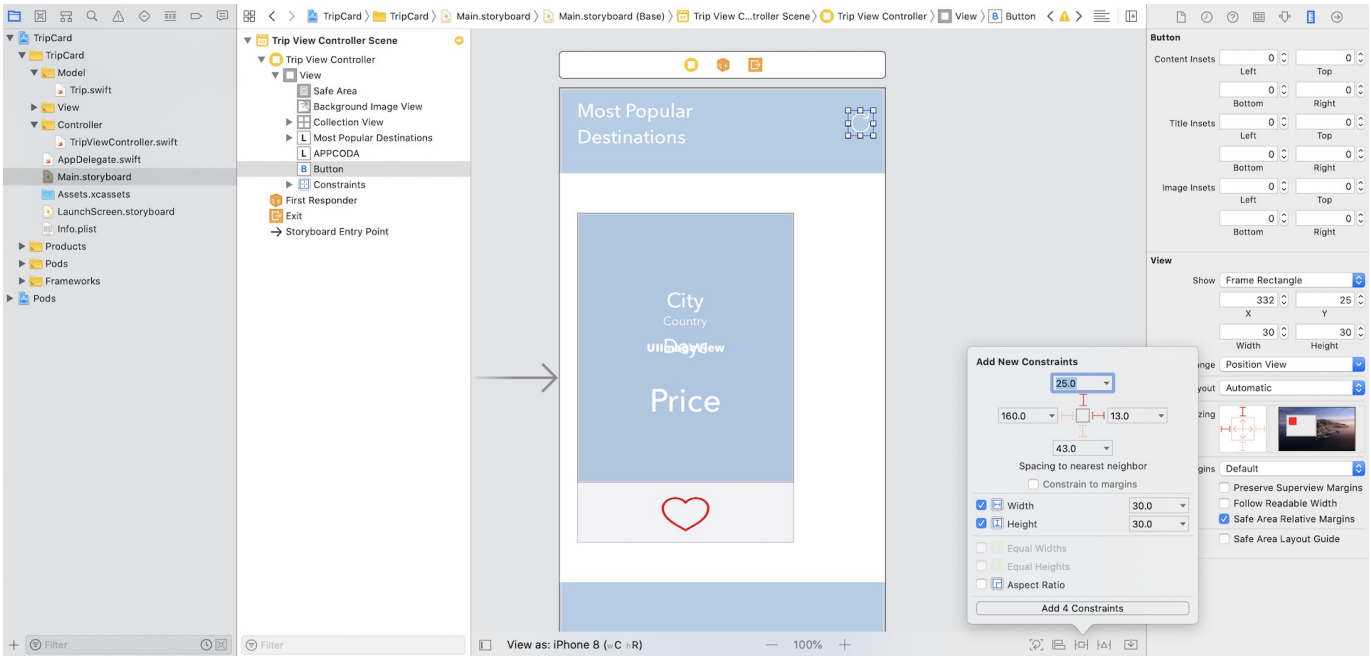


Figure 30.12. The TripCard app now loads data from the Parse cloud

Refreshing Data

Currently, there is no way to refresh the data. Let's add a button to the Trip View Controller in the 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. 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 `Add New Constraints` button of the auto layout menu to add the layout constraints (see figure 30.13).



Next, insert an action method in `TripViewController.swift`:

```
@IBAction func reloadButtonTapped(sender: Any) {
    loadTripsFromParse()
}
```

Go back to the storyboard and associate the refresh button with this action method. Control-drag from the `reload` button to the first responder button in the dock. After releasing the buttons, select `reloadButtonTappedWithSender:`.

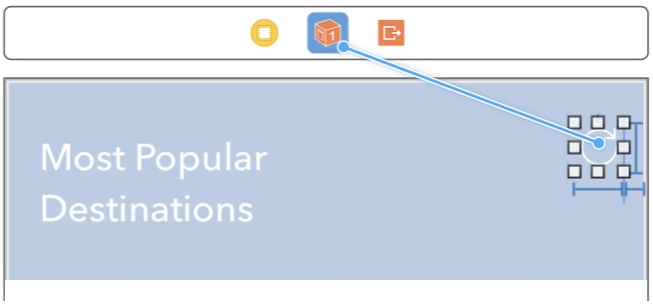


Figure 30.14. Connecting the button with the action method

Now run the app again. Once it's launched, go to the Parse dashboard and add/remove a new trip. Your app should now retrieve the new trip when the refresh button is tapped.

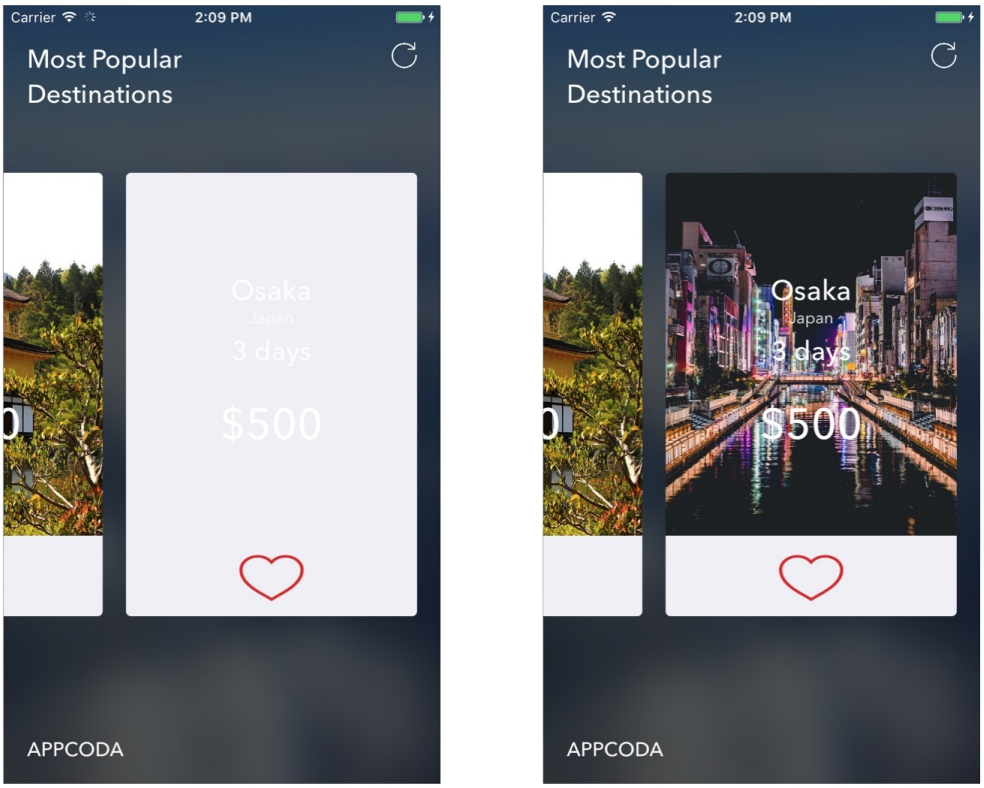


Figure 30.15. Adding a new record in the Parse cloud, then click the Reload button to refresh the data

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 the Parse backend. 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:

```
017-12-20 17:39:14.360195+0800 TripCard[30346:3122327] [Error]: The Internet conne
ction appears to be offline. (Code: 100, Version: 1.15.3)
2017-12-20 17:39:14.360488+0800 TripCard[30346:3122327] [Error]: Network connectio
n failed. Making attempt 1 after sleeping for 1.816037 seconds.
2017-12-20 17:39:16.364712+0800 TripCard[30346:3122334] [Error]: The Internet conn
ection appears to be offline. (Code: 100, Version: 1.15.3)
2017-12-20 17:39:16.365201+0800 TripCard[30346:3122334] [Error]: Network connectio
n failed. Making attempt 2 after sleeping for 3.632074 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 if the Internet access is not available, your app can load the result from local 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. However, you can easily enable it by writing a single line of code. Add the following code to 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 because we haven't written any code for pushing the updates to the cloud.

With the Parse SDK, it is pretty simple to update a `PFObject`. 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 `saveInBackground` 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: TripCollectionViewCell) {
    if let indexPath = collectionView.indexPath(for: 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().saveInBackground(block: { (success, erro
r) -> Void in
            if (success) {
                print("Successfully updated the trip")
            } else {
                print("Error: \(error?.localizedDescription ?? "Unknown error")")
            }
        })
    }
}
```

In the `if let` block, the first line of code is to set the `isLiked` property of the corresponding `Trip` object to `true` when a user taps the heart button.

To upload the update to the Parse cloud, 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`. This is how Parse identifies the object to update.

Once we have the `PFObject`, we simply call the `saveInBackground` 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`.

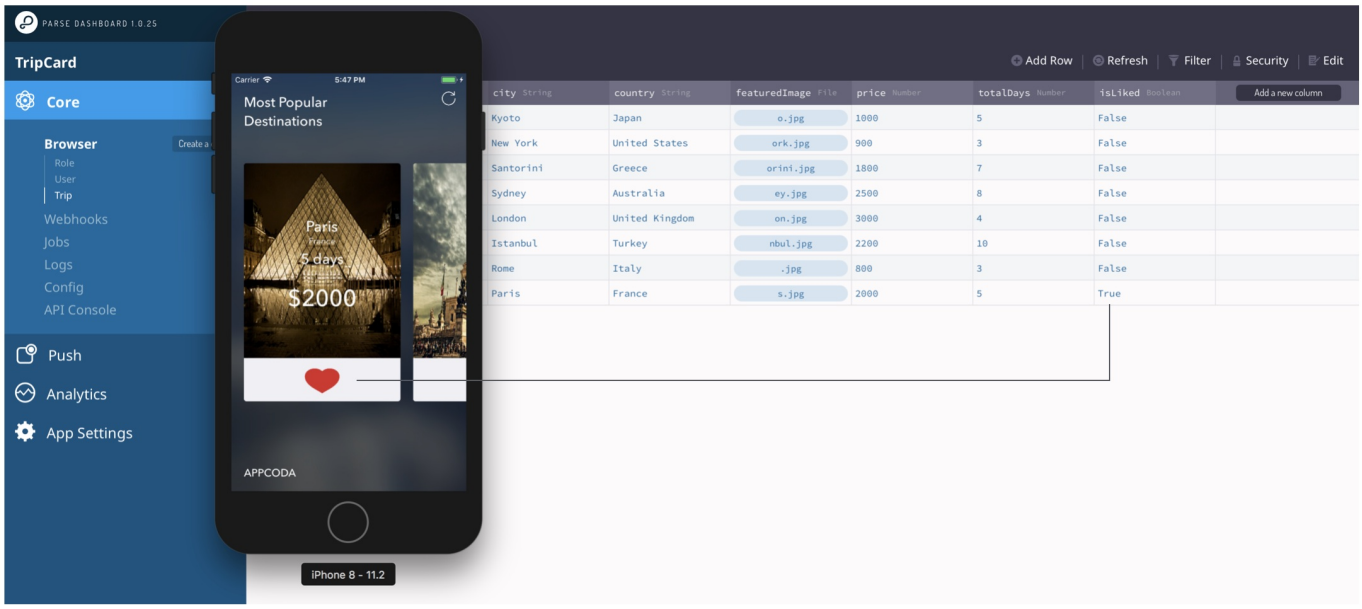


Figure 30.16. Tapping the heart button of the Paris card will update the isLiked property of the corresponding record on the Parse cloud

Deleting Data from Parse

Similarly, `PFObject` provides various methods for object deletion. In short, you call up the `deleteInBackground` method of the `PBObject` 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: #selector(h
andleSwipe(gesture:)))
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, we create an extension to implement the protocol:

```
extension TripViewController: UIGestureRecognizerDelegate {

    @objc func handleSwipe(gesture: UISwipeGestureRecognizer) {
        let point = gesture.location(in: self.collectionView)

        if (gesture.state == UIGestureRecognizer.State.ended) {
            if let indexPath = collectionView.indexPathForItem(at: point) {
                // Remove trip from Parse, array and collection view
                trips[indexPath.row].toPFObject().deleteInBackground(block: { (suc
cess, error) -> Void in
                    if (success) {
                        print("Successfully removed the trip")
                    } else {
                        print("Error: \(error?.localizedDescription ?? "Unknown er
ror")")
                        return
                    }

                    self.trips.remove(at: indexPath.row)
                    self.collectionView.deleteItems(at: [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 `location(in:)` 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 `indexPathForItem(at:)` method. Once we have the index path of the cell to be removed, we call the `deleteInBackground` method to delete it from Parse, and remove the item from the collection view.

Great! You've implemented the delete feature. Hit the Run button to launch the app and try to delete a record from Parse.

Summary

I hope that this chapter gave you an idea about how to connect your app to the cloud. In this chapter, we use Back4app.com as the Parse backend, which frees you from configuring and managing your own Parse servers. It is not a must to use back4app.com. There are quite a number of Parse hosting service providers you can try it out such as SashiDo.io.

The startup cost of using a cloud is nearly zero. And, with the Parse SDK, it is very simple to add a cloud backend for your apps. If you think it's too hard to integrate your app with the cloud, think again! And begin to consider implementing your existing apps with some cloud features.

For reference, you can download the final project from http://www.appcoda.com/resources/swift5/ParseDemo.zip.

Chapter 31
Parsing CSV and Preloading 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 13 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 http://www.appcoda.com/resources/swift5/CoreDataPreloadDemoStarter.zip and have a trial run.

I have already built the data model and provided the implementation of the table view. You can look into the `MenuItemTableViewController` class and `CoreDataPreloadDemo.xcdatamodeld` for details. The data model is pretty simple. I have defined a `MenuItem` entity, which includes three attributes: *name*, *detail*, and *price*.

If you open `AppDelegate.swift`, you will see the following code snippet:

```
lazy var persistentContainer: NSPersistentContainer = {
    /*
     The persistent container for the application. This implementation
     creates and returns a container, having loaded the store for the
     application to it. This property is optional since there are legitimate
     error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "CoreDataPreloadDemo")
    container.loadPersistentStores(completionHandler: { (storeDescription, error)
in
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error appropria
tely.
            // fatalError() causes the application to generate a crash log and ter
minate. You should not use this function in a shipping application, although it ma
y be useful during development.

            /*
             Typical reasons for an error here include:
             * The parent directory does not exist, cannot be created, or disallow
s writing.
             * The persistent store is not accessible, due to permissions or data
protection when the device is locked.
             * The device is out of space.
             * The store could not be migrated to the current model version.
             Check the error message to determine what the actual problem was.
             */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

// MARK: - Core Data Saving support

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
```

```
            try context.save()
        } catch {
            // Replace this implementation with code to handle the error appropria
tely.
            // fatalError() causes the application to generate a crash log and ter
minate. You should not use this function in a shipping application, although it ma
y be useful during development.
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

It already comes with the code required for loading the Core Data model (i.e. CoreDataPreloadDemo.xcdatamodeld).

In the `MenuTableViewController.swift` file, I also implemented the required code for retrieving the menu items:

```
// Load menu items from database
if let appDelegate = (UIApplication.shared.delegate as? AppDelegate) {
    let request: NSFetchRequest<MenuItem> = MenuItem.fetchRequest()
    let context = appDelegate.persistentContainer.viewContext
    do {
        menuItems = try context.fetch(request)
    } catch {
        print("Failed to retrieve record")
        print(error)
    }
}
```

The demo is a very simple app showing a list of food. By default, the starter project comes with an empty database. If 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.

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.

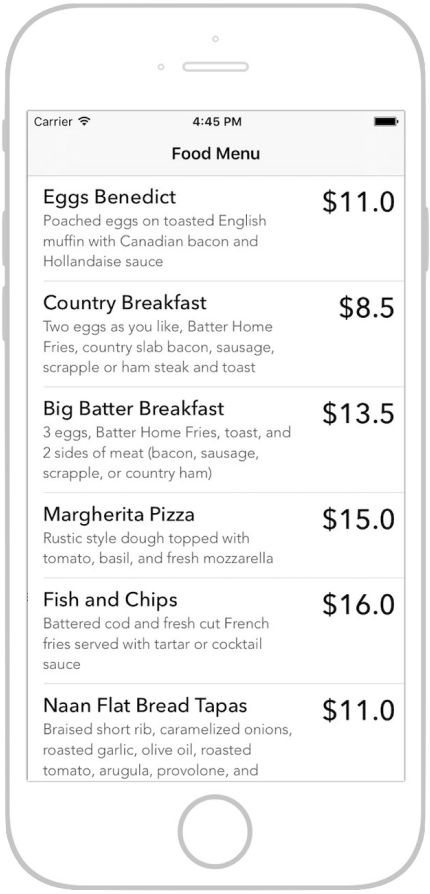


Figure 31.1. Demo app with food menu items

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 by 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 Holl
andaise sauce",11.0
Country Breakfast,"Two eggs as you like, Batter Home Fries, country slab bacon, sa
usage, 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 mozzarel
la",15.0
Fish and Chips,Battered cod and fresh cut French fries served with tartar or cockt
ail 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 by 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 in 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 first create an extension of `AppDelegate` with a method for parsing the CSV file:

```
extension AppDelegate {
    func parseCSV (contentsOfURL: URL, encoding: String.Encoding) -> [(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(contentsOf: contentsOfURL, encoding: encoding
)
            items = []
            let lines: [String] = content.components(separatedBy: .newlines)

            for line in lines {
                var values:[String] = []
                if line != "" {
                    // For a line with double quotes
                    // we use NSScanner to perform the parsing
                    if line.range(of: "\"") != nil {
                        var textToScan:String = line
                        var value:NSString?
                        var textScanner:Scanner = Scanner(string: textToScan)
                        while textScanner.string != "" {

                            if (textScanner.string as NSString).substring(to: 1) =
= "\"" {
                                textScanner.scanLocation += 1
                                textScanner.scanUpTo("\"", into: &value)
                                textScanner.scanLocation += 1
                            } else {
                                textScanner.scanUpTo(delimiter, into: &value)
                            }

                            // Store the value into the values array
                            if let value = value {
                                values.append(value as String)
```

```
                            }

                            // Retrieve the unscanned remainder of the string
                            if textScanner.scanLocation < textScanner.string.count
 {
                                textToScan = (textScanner.string as NSString).subs
tring(from: textScanner.scanLocation + 1)
                            } else {
                                textToScan = ""
                            }
                            textScanner = Scanner(string: textToScan)
                        }

                        // For a line without double quotes, we can simply separat
e the string
                        // by using the delimiter (e.g. comma)
                    } else  {
                        values = line.components(separatedBy: 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 `components(separatedBy:)` 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, sa
usage, scrapple or ham steak and toast", 8.5
```

In this case, we cannot simply use the `components(separatedBy:)` method to separate the field values. Instead, we use `Scanner` 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 `scanUpTo` 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:

1. First, we will remove all the existing data from the database. This operation is optional if you can ensure the database is empty.
2. 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` extension:

```
func preloadData() {

```

```
    // Load the data file. For any reasons it can't be loaded, we just return
    guard let contentsOfURL = Bundle.main.url(forResource: "menudata", withExtensi
on: "csv") else {
        return
    }

    // Remove all the menu items before preloading
    removeData()

    // Parse the CSV file and import the data
    if let items = parseCSV(contentsOfURL: contentsOfURL, encoding: String.Encoding
.utf8) {

        let context = persistentContainer.viewContext

        for item in items {
            let menuItem = MenuItem(context: context)
            menuItem.name = item.name
            menuItem.detail = item.detail
            menuItem.price = Double(item.price) ?? 0.0

            do {
                try context.save()
            } catch {
                print(error)
            }
        }

    }
}

func removeData() {
    // Remove the existing items
    let fetchRequest = NSFetchRequest<MenuItem>(entityName: "MenuItem")
    let context = persistentContainer.viewContext

    do {

        let menuItems = try context.fetch(fetchRequest)

        for menuItem in menuItems {
            context.delete(menuItem)
```

```
            }

            saveContext()

        } 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 `delete` 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:

```
Bundle.main.url(forResource: "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 into the database.

Lastly, update the `application(_:didFinishLaunchingWithOptions:)` method like this to call the `preloadData()` method:

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

    let defaults = UserDefaults.standard
    let isPreloaded = defaults.bool(forKey: "isPreloaded")
    if !isPreloaded {
        preloadData()
        defaults.set(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 downloads 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://drive.google.com/uc?export=download&id=0ByZhaKOAvtNGelJOMEdhRFo2c28
```

```
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://drive.google.com/uc?export=download&id=[folder_id]

Please replace [folder_id] 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 preloadData() {

    // Load the data file from a remote URL
    guard let remoteURL = URL(string: "https://drive.google.com/uc?export=download
&id=0ByZhaKOAvtNGelJOMEdhRFo2c28") else {
        return
    }

    // Remove all the menu items before preloading
    removeData()

    // Parse the CSV file and import the data
    if let items = parseCSV(contentsOfURL: remoteURL, encoding: String.Encoding.ut
f8) {

        let context = persistentContainer.viewContext

        for item in items {
            let menuItem = MenuItem(context: context)
            menuItem.name = item.name
            menuItem.detail = item.detail
            menuItem.price = Double(item.price) ?? 0.0

            do {
                try context.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 launchO
ptions: [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.

Figure 31.3. The demo app now preloads the menu item from a remote location

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/CoreDataPreloadDemo.zip.

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 in a poor user experience. Apparently, you want to pre-filled the database beforehand and bundle it directly into 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 http://www.appcoda.com/resources/swift5/CoreDataPreloadDemoStarter.zip. 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 launc
hOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {

    let urls = FileManager.default.urls(for: .documentDirectory, in: .userDomainMa
sk)
    print(urls[0])

    return true
}
```

The SQLite database is generated under the application's document directory. To find the file path, we use `FileManager` to retrieve the document directory of the application.

Now run the app again. You should see an output in the console window showing the full path of the document directory like this:

```
file:///Users/simon/Library/Developer/CoreSimulator/Devices/7DC35502-54FD-447A-B10F-2B7B0FC5BDEF/data/Containers/Data/Application/505CF334-9CC4-404A-9236-4B88436F0808/Documents/
```

Copy the file path and go to Finder. In the menu select Go > Go to Folder... and then paste the path (without file://) in the pop-up. Click `Go` to confirm.

Once you open the document folder in Finder, you will find the `Library` folder at the same level. Go into the `Library` folder > `Application Support`. You will see three files: *CoreDataPreloadDemo.sqlite*, *CoreDataPreloadDemo.sqlite-wal* and *CoreDataPreloadDemo.sqlite-shm*.

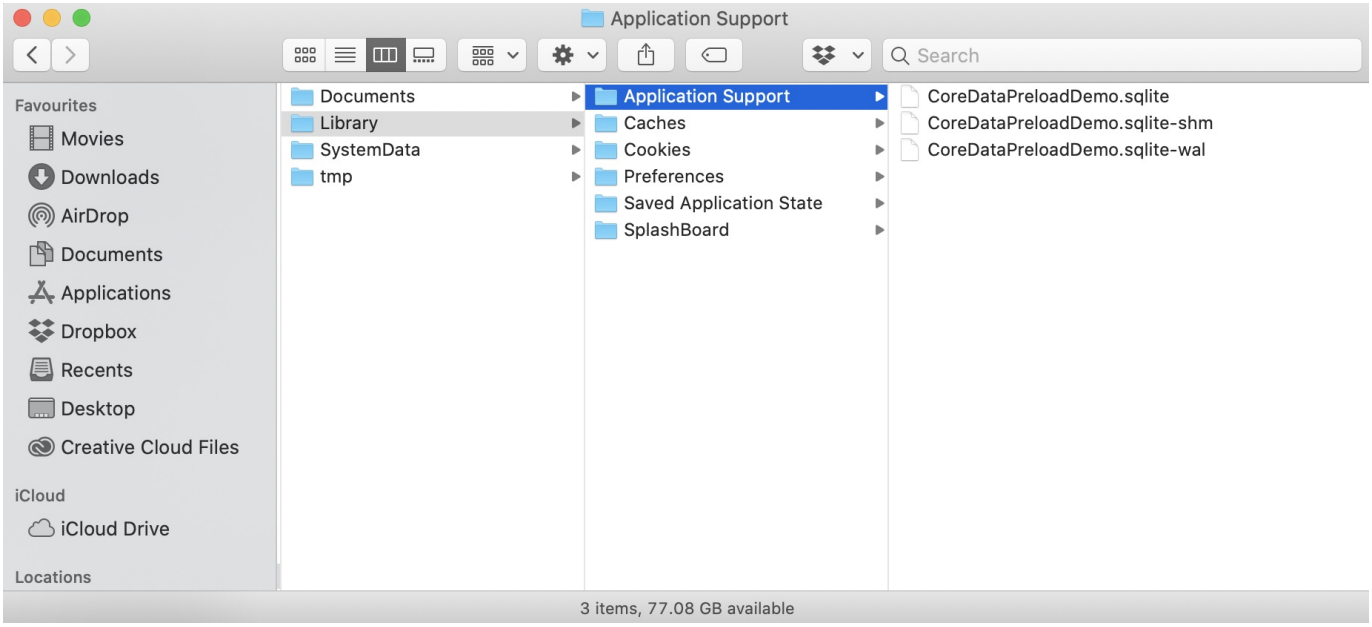


Figure 31.4. Revealing the SQLite files in Finder

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 to copy these three files. If you just copy the CoreDataDemo.sqlite file, you will probably end up with an empty database.

Now go back the starter project you just downloaded. Drag these three files to the project navigator.

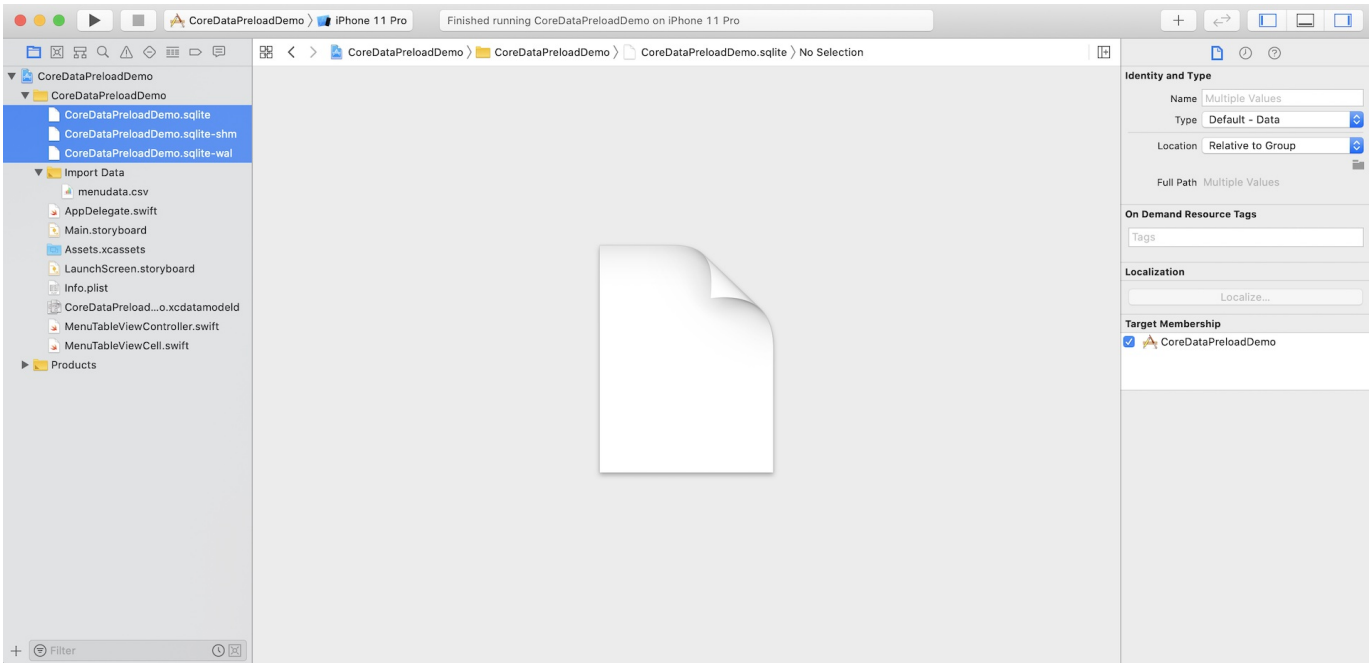


Figure 31.5. Adding the database files to the project

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, this database will be embedded in the app when you build the project. 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, modify the declaration of the persistentStoreCoordinator variable like this:

```
lazy var persistentContainer: NSPersistentContainer = {

    let directoryUrls = FileManager.default.urls(for: .documentDirectory, in: .use
rDomainMask)
    let applicationDocumentDirectory = directoryUrls[0]
    let storeUrl = applicationDocumentDirectory.appendingPathComponent("CoreDataPr
eloadDemo.sqlite")

    // Load the existing database
    if !FileManager.default.fileExists(atPath: storeUrl.path) {
        let sourceSqliteURLs = [Bundle.main.url(forResource: "CoreDataPreloadDemo"
, withExtension: "sqlite")!, Bundle.main.url(forResource: "CoreDataPreloadDemo", w
ithExtension: "sqlite-wal")!, Bundle.main.url(forResource: "CoreDataPreloadDemo",
withExtension: "sqlite-shm")!]
        let destSqliteURLs = [applicationDocumentDirectory.appendingPathComponent(
"CoreDataPreloadDemo.sqlite"), applicationDocumentDirectory.appendingPathComponent(
"CoreDataPreloadDemo.sqlite-wal"), applicationDocumentDirectory.appendingPathCompo
nent("CoreDataPreloadDemo.sqlite-shm")]

        for index in 0..<sourceSqliteURLs.count {
            do {
                try FileManager.default.copyItem(at: sourceSqliteURLs[index], to:
destSqliteURLs[index])
            } catch {
                print(error)
            }
        }
    }

    // Prepare the description of the Persistent Store
    let description = NSPersistentStoreDescription()
    description.url = storeUrl

    /*
     The persistent container for the application. This implementation
     creates and returns a container, having loaded the store for the
     application to it. This property is optional since there are legitimate
     error conditions that could cause the creation of the store to fail.
    */
    let container = NSPersistentContainer(name: "CoreDataPreloadDemo")
    container.persistentStoreDescriptions = [description]
```

```
    container.loadPersistentStores(completionHandler: { (storeDescription, error)
in
        if let error = error as NSError? {
            // Replace this implementation with code to handle the error appropria
tely.
            // fatalError() causes the application to generate a crash log and ter
minate. You should not use this function in a shipping application, although it ma
y be useful during development.

            /*
             Typical reasons for an error here include:
             * The parent directory does not exist, cannot be created, or disallow
s writing.
             * The persistent store is not accessible, due to permissions or data
protection when the device is locked.
             * The device is out of space.
             * The store could not be migrated to the current model version.
             Check the error message to determine what the actual problem was.
             */
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()
```

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 `copyItem(at:)` method of `FileManager`.

We then create another description object of the persistent store and specify the URL of the database. When we instantiate the `NSPresistentContainer` object, it uses the specified store description to create the store.

```
let container = NSPersistentContainer(name: "CoreDataPreloadDemo")
container.persistentStoreDescriptions = [description]
```

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 http://www.appcoda.com/resources/swift5/CoreDataExistingDB.zip.

Chapter 32
Gesture Recognizers, Multiple Annotations with Polylines and Routes

In earlier chapters, we discussed how to get directions and draw routes on maps. Now you should understand how to use the `MKDirections` API to retrieve the route-based directions between two annotations and display the route on a map view.

What if you have multiple annotations on a map? How can you connect those annotations together and even draw a route between all those points?

This is one of the common questions from my readers. In this chapter, I will walk you through the implementation by building a working demo. Actually, I have covered the necessary APIs in chapter 8, so if you haven't read the chapter, I recommend you to check it out first.

The Sample Route App

We will build a simple route demo app that lets users pin multiple locations by a simple press. The app then allows the user to display a route between the locations or simply connect them through straight lines.

You can start with this project template (http://www.appcoda.com/resources/swift5/RouteDemoStarter.zip). If you build the starter project, you should have an app showing a map view.

The `RouteViewController` is the view controller class associated with the view controller in the storyboard. And if you look into `RouteViewController.swift`, you will notice that I have connected the map view with the `mapView` outlet variable.

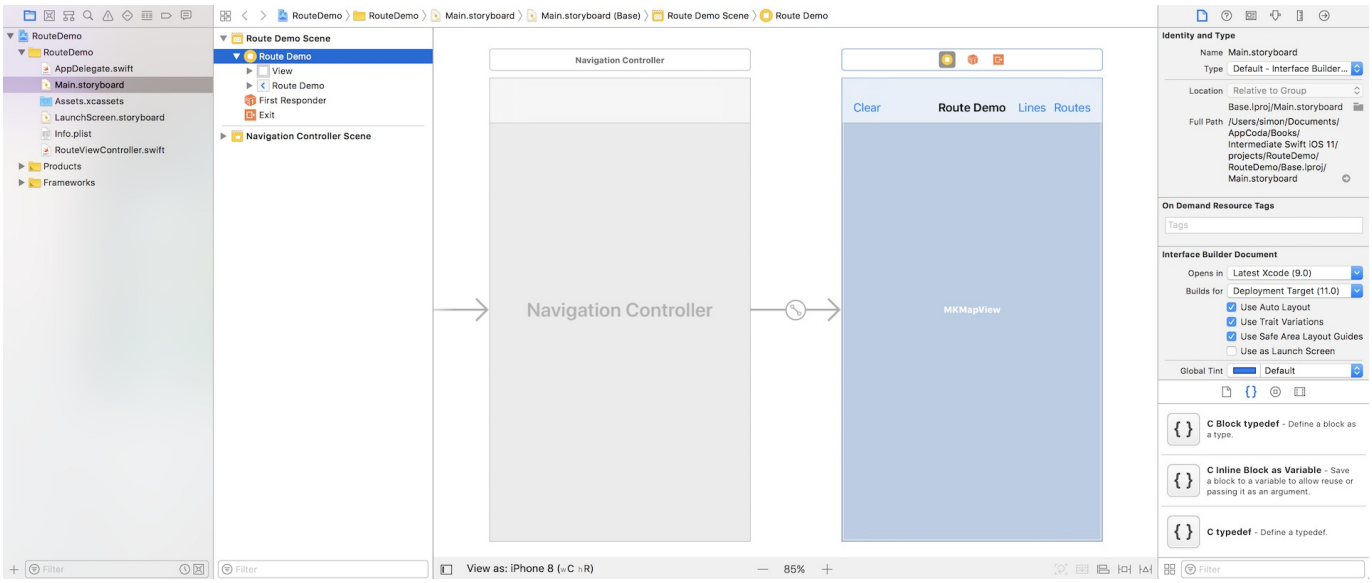


Figure 32.1. The storyboard of the starter project

That's it for the starter project. We will now build on top of it and add more features.

Detecting a Touch Using Gesture Recognizers

First things first, users can pin a location on the map by using a finger press in the app. Apple provides several standard gesture recognizers for developers to detect a touch including:

- `UITapGestureRecognizer` - for detecting a tap (or multiple taps)
- `UIPinchGestureRecognizer` - for detecting a pin (zoom-in and zoom-out)
- `UIPanGestureRecognizer` - for detecting a pan gesture
- `UISwipeGestureRecognizer` - for detecting a swipe gesture
- `UIRotationGestureRecognizer` - for detecting a rotation gesture (i.e. fingers moving in opposite directions)
- `UILongPressGestureRecognizer` - for detecting a "touch and hold" gesture

So which gesture recognizer should we use in our Route demo app? The obvious choice is to utilize `UITapGestureRecognizer` because it is responsible to detect a tap. However, if you've used the built-in Maps app before, you should know that you can zoom in the map by double tapping the screen. The problem of using `UITapGestureRecognizer` in this situation is that you have to find a way to differentiate between a single tap and a double tap.

To keep things simple, we will use `UILongPressGestureRecognizer` instead. This gesture recognizer will only trigger an action when the user presses a finger on a view and hold them there for a certain period of time. While its name suggests that the class is designed to look for long-press gestures, the duration of a press doesn't need to be over 10 seconds in order to be considered as a long press. You can actually configure the duration of the press through the `minimumPressDuration` property. So the duration of a long press can be set to 0.1 second or even shorter.

Let's now see how to utilize the `UILongPressGestureRecognizer` class.

Applying Gesture Recognizers

All predefined gesture recognizer classes are very easy to use. You just need to write a few lines of code, and your app is ready to detect certain gestures. For the demo app, we implement the `UILongPressGestureRecognizer` like below. You can insert the code snippet in the `viewDidLoad` method of the `RouteViewController` class.

```
let longpressGestureRecognizer = UILongPressGestureRecognizer(target: self, action
: #selector(pinLocation))
longpressGestureRecognizer.minimumPressDuration = 0.3
mapView.addGestureRecognizer(longpressGestureRecognizer)
```

In the code above, we first instantiate an instance of `UILongPressGestureRecognizer` with a target and an action. When a long-press is recognized, the recognizer will trigger an action of a specific object. The `target` parameter tells the recognizer which object to connect with. And the action specifies the action method to call. Here we set the `target` to `self` (i.e. `RouteViewController`) and the `action` to `pinLocation`.

```
Note: We haven't implemented the `pinLocation` method yet, so
it is normal that Xcode indicates an error for the first line
of the above code. The #selector syntax was introduced in the
early version of Swift (2.2). By using #selector, it will check
your code at compile time to make sure the method you specify
actually exists.
```

As mentioned before, you can specify how long a finger must press on a screen for the gesture to be recognized. We simply set the minimum press duration to `0.3` seconds.

Lastly, you should associate the gesture recognizer with a specific view. To make the association, you simply call the view's `addGestureRecognizer` method and pass the corresponding gesture recognizer. Since the map view is the view that interacts with the user's touch, we associate the long-press recognizer with the map view.

Pinning a Location on the Map

When the user presses a specific location on the map view, the gesture recognizer we have created earlier will call the `pinLocation` method of the `RouteViewController` class.

The method is created for pinning the selected location on the map. Specifically, here is what the method will do:

1. Get the location of the press
2. Convert the location from a point to a coordinate
3. With the coordinate, annotate the location on the map

Now, implement the `pinLocation` like this:

```
@objc func pinLocation(sender: UILongPressGestureRecognizer) {
    if sender.state != .ended {
        return
    }

    // Get the location of the touch
    let tappedPoint = sender.location(in: mapView)

    // Convert point to coordinate
    let tappedCoordinate = mapView.convert(tappedPoint, toCoordinateFrom: mapView)

    // Annotate on the map view
    let annotation = MKPointAnnotation()
    annotation.coordinate = tappedCoordinate

    // Store the annotation for later use
    annotations.append(annotation)

    mapView.showAnnotations([annotation], animated: true)
}
```

When the method is called by the recognizer, it will pass a `UILongPressGestureRecognizer` object. The best practice is to check if the gesture has actually ended. All gesture recognizers provide a `state` property that stores the current state of the recognizer. We

verify if the state equals `.ended`, otherwise, we simply return from the method.

You can use `location(in:)` of a gesture recognizer to get the location of the press. The method returns a point (in the form of `CGPoint`) that identifies the touch. To annotate this location on the map, we have to convert it from a point to a coordinate. The `MKMapView` class provides a built-in method named `convert(_:toCoordinateFrom:)` for this purpose.

With the coordinate of the location, we can create a `MKPointAnnotation` object and display it on the map view by calling `showAnnotations`.

In the above code, we also add the current annotation to an array. The `annotations` array stores all the pinned locations. Later we will use the data to draw routes.

To make the app work, remember to declare the `annotations` variable in the `RouteViewController` class:

```
private var annotations = [MKPointAnnotation]()
```

Now run the app to have a quick test. Press anywhere of the map to pin a location.

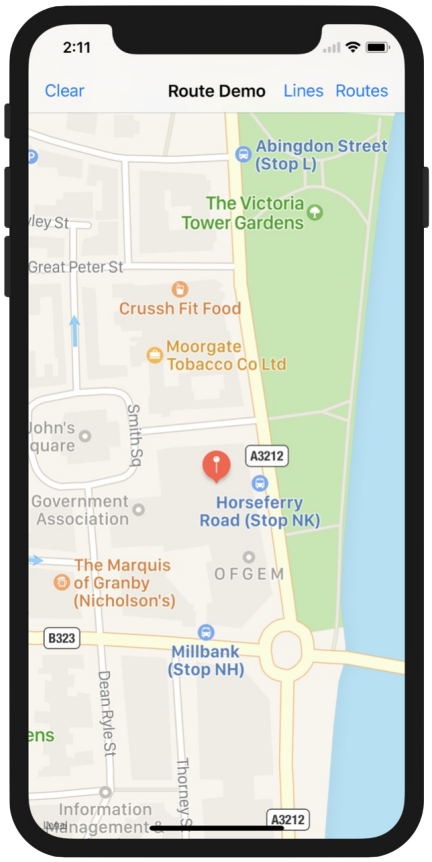


Figure 32.2. Press the map view to pin a location

Drop Pin Animation

> Beautiful, subtle animation pervades the iOS UI and makes the app experience more engaging and dynamic. Appropriate animation can: – Communicate status and provide feedback – Enhance the sense of direct manipulation – Help people visualize the results of their actions
>
> – iOS Human Interface Guidelines, Apple

The bar has been raised. More and more apps are well-crafted with polished and thoughtful animations to delight their users. Your app may fall short of your users' expectations if you do not put in any efforts in designing these meaningful animations.

I'm not talking about those big, 3D and fancy animations or effects. Instead, I'm referring to those subtle animations that set your app apart from the competition and better the user experience.

One great example is the hamburger button animation created by the CreativeDash team (https://dribbble.com/shots/1695411-Open-Close-Source). When you tap the hamburger button, it turns into a close button with a really nice transition. Though the animation is subtle, it helps maintain continuity and gives a meaningful transition.

Now let's take a look at the Route demo app again. When a user presses on the map to pin a location, it just shows an annotation right away. Wouldn't it be great if we add a drop pin animation?

To create the animation, we have to adopt the `MKMapViewDelegate` protocol. The protocol defines the `mapView(_:didAdd:)` method that is called when an annotation view is about to add to the map:

```
optional func mapView(_ mapView: MKMapView, didAdd views: [MKAnnotationView])
```

By implementing this method, we can provide a custom animation before the annotation appears on the map.

First, edit the class declaration to adopt the `MKMapViewDelegate` protocol:

```
class RouteViewController: UIViewController, MKMapViewDelegate {
```

Next, implement the method like this:

```
func mapView(_ mapView: MKMapView, didAdd views: [MKAnnotationView]) {
    let annotationView = views[0]
    let endFrame = annotationView.frame
    annotationView.frame = endFrame.offsetBy(dx: 0, dy: -600)
    UIView.animate(withDuration: 0.3, animations: { () -> Void in
        annotationView.frame = endFrame
    })
}
```

The `frame` property of the annotation provides the resulting position of the pin. In order to create a drop pin animation, we first change the position of the frame (i.e. pin) by offsetting its vertical position. The start position of the pin is now a bit higher than the resulting position. We then call the `animate(withDuration:animations:)` method of `UIView` to create the drop pin animation.

Lastly, insert the following line of code in the `viewDidLoad` method to specify the delegate of the map view:

```
mapView.delegate = self
```

Run the app again. Press on the map using a single finger, and then release it. The app should display a pin with an animation.

Connecting Annotations with Polylines

Now that your users should be able to pin multiple locations on the map, the next thing we are going to do is to connect the annotations with line segments. Technically speaking, it means we need to create an `MKPolyline` object from a series of points or coordinates. `MKPolyline` is a class that can be used to represent a connected sequence of line segments. You can create a polyline by constructing a `MKPolyline` object with a series of end-points.

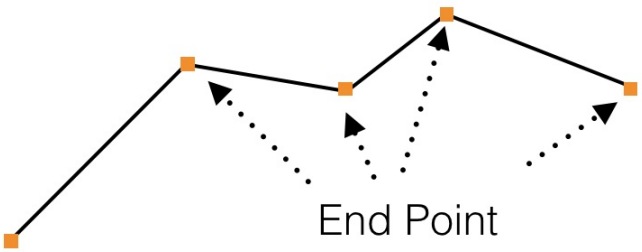


Figure 32.3. A polyline

Now let's create a `drawPolyline` action method in the `RouteViewController` class:

```
@IBAction func drawPolyline() {
    mapView.removeOverlays(mapView.overlays)

    var coordinates = [CLLocationCoordinate2D]()
    for annotation in annotations {
        coordinates.append(annotation.coordinate)
    }

    let polyline = MKPolyline(coordinates: &coordinates, count: coordinates.count)

    mapView.addOverlay(polyline)
}
```

You can create a `MKPolyline` object by specifying the series of map points or coordinates. In this case, we use the latter option. So we first retrieve all the coordinates of the annotations and store them into the `coordinates` array. Then we use the coordinates to construct a `MKPolyline` object. To display a shape or line segments on a map, you use overlays to layer the content over the map. Here the `MKPolyline` object is the overlay object. You can simply call the `add` method of a map view to add the overlay object.

The `drawPolyline` method will be called when the user taps the *Lines* button. We haven't associated the *Lines* button with the `drawPolyline` method yet. Now, go to `Main.storyboard`. Control-drag from the *Lines* button to the view controller icon of the dock. In the pop-over menu, select `drawPolyline` to connect with the method.

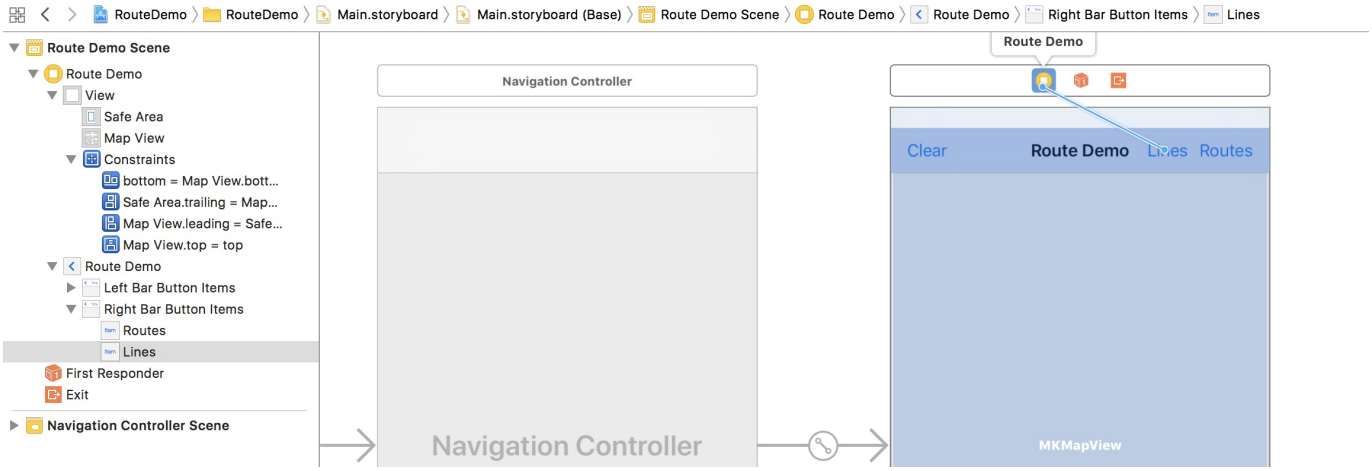


Figure 32.4. Connecting the Lines button with the action method

Before moving on, let's do a quick test. Run the app, pin several locations on the map, and tap the *Lines* button. If you expect the app connects the locations with lines, you will be disappointed.

Why? Did we miss anything?

The overlay object (i.e. `MKPolyline` object) added to the map view is actually a data object. It only contains the points needed to provide the location of the overlay on the map. The `add` method is not responsible to draw the overlay's content onto the screen.

Instead, the presentation of an overlay is handled by an overlay renderer object, which is an instance of the `MKOverlayRenderer` class. Every time when an overlay moves onscreen, the map view calls the `mapView(_:rendererFor:)` method of its delegate to ask for the corresponding overlay renderer object. Since we haven't implemented the method, the map view has no ideas how to render the overlay.

Now implement the required method like this:

```
func mapView(_ mapView: MKMapView, rendererFor overlay: MKOverlay) -> MKOverlayRen
derer {
    let renderer = MKPolylineRenderer(overlay: overlay)
    renderer.lineWidth = 3.0
    renderer.strokeColor = UIColor.purple
    renderer.alpha = 0.5

    return renderer
}
```

Before drawing the line segments, we first remove all the existing overlays on the map view. `MKPolylineRenderer`, a subclass of `MKOverlayRenderer`, provides the visual representation for the `MKPolyline` overlay object. The renderer object has several properties for developers to customize the rendering. In the above code, we change the line width, stroke color, and alpha value.

Now re-run the project. This time the map view should be able to draw the overlay on the screen.

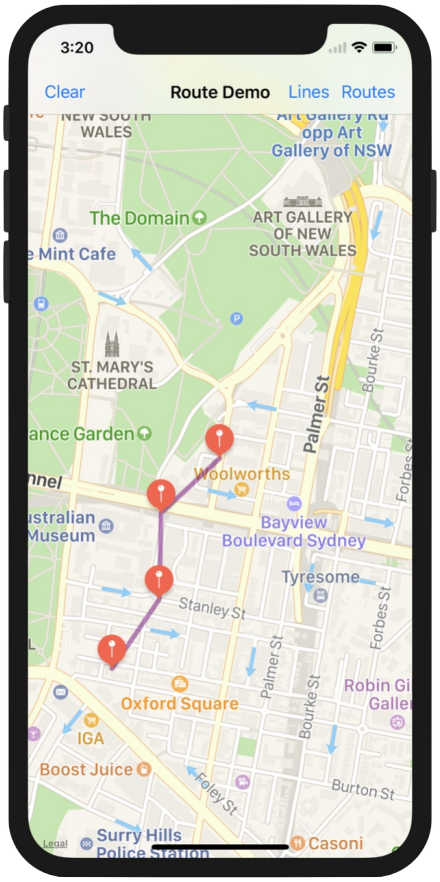


Figure 32.5. The demo app can now connect the dots with lines

However, there is a problem with the current implementation. You have to manually zoom out the map in order to view the route. To offer a better user experience, insert the following lines of code in the above method (place them before `return renderer`):

```
let visibleMapRect = mapView.mapRectThatFits(renderer.polyline.boundingMapRect, ed
gePadding: UIEdgeInsets(top: 50, left: 50, bottom: 50, right: 50))
mapView.setRegion(MKCoordinateRegion(visibleMapRect), animated: true)
```

Base on the given polyline, the `mapRectThatFits` method computes the new viewable area of the map that fits the polyline. Optionally, you can add a padding to the new map rectangle. Here, we set the padding value for each side to `50` points. With the new map rectangle, we call `setRegion` of the map view to change the visible region accordingly.

Connecting Annotations with Routes

It's pretty easy to connect the points with line segments, right? But that doesn't give users a lot of information. Instead, we want to display the actual routes between the annotations.

In the earlier chapter, we've explored the `MKDirection` API that allows developers to access the route-based direction data from Apple's server. To draw the actual routes, here is what we're going to do:

1. Assuming we have three annotations on the map, we will first search for the route between point 1 and point 2 using the `MKDirection` API.
2. Display the route on the map using overlay.
3. Repeat the above steps for point 2 and point 3.
4. If there are more than three annotations, just keep repeating the steps for the rest of the annotations.

Let's first create the method for computing the direction and drawing between two coordinates. Insert the following method into the `RouteViewController` class:

```
func drawDirection(startPoint: CLLocationCoordinate2D, endPoint: CLLocationCoordin
ate2D) {

    // Create map items from coordinate
    let startPlacemark = MKPlacemark(coordinate: startPoint, addressDictionary: nil
)
    let endPlacemark = MKPlacemark(coordinate: endPoint, addressDictionary: nil)
    let startMapItem = MKMapItem(placemark: startPlacemark)
    let endMapItem = MKMapItem(placemark: endPlacemark)

    // Set the source and destination of the route
    let directionRequest = MKDirections.Request()
    directionRequest.source = startMapItem
    directionRequest.destination = endMapItem
    directionRequest.transportType = MKDirectionsTransportType.automobile

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

    directions.calculate { (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)
    }
}
```

The `drawDirection` method takes in the coordinates of the two annotations. It then converts the coordinate into `MKMapItem` and creates a direction request for the two map items. For demo purpose, we just set the transportation type to auto mobile. To initiate

the request, we call the `calculate(completionHandler:)` method, which creates an asynchronous request for directions and calls the completion handler when the operation completes. Once the route information computed, we display it on the map as an overlay.

Now that we have a function for calculating the direction between two points, let's create the action method that loops through all the annotations:

```
@IBAction func drawRoute() {
    mapView.removeOverlays(mapView.overlays)

    var coordinates = [CLLocationCoordinate2D]()
    for annotation in annotations {
        coordinates.append(annotation.coordinate)
    }

    var index = 0
    while index < annotations.count - 1 {
        drawDirection(startPoint: annotations[index].coordinate, endPoint: annotat
ions[index + 1].coordinate)
        index += 1
    }
}
```

This method is called when the user taps the *Routes* button in the navigation bar. It first removes the existing overlay, and then retrieve all the annotations on the map view. Lastly, it makes use of the `drawDirection` method, that we have just created, to calculate the routes between the annotations.

We haven't associated the *Routes* button with the `drawRoute` method. Open `Main.storyboard` and control-drag from the *Routes* button to the view controller icon. Select `drawRoute` from the pop-over menu to make the connection.

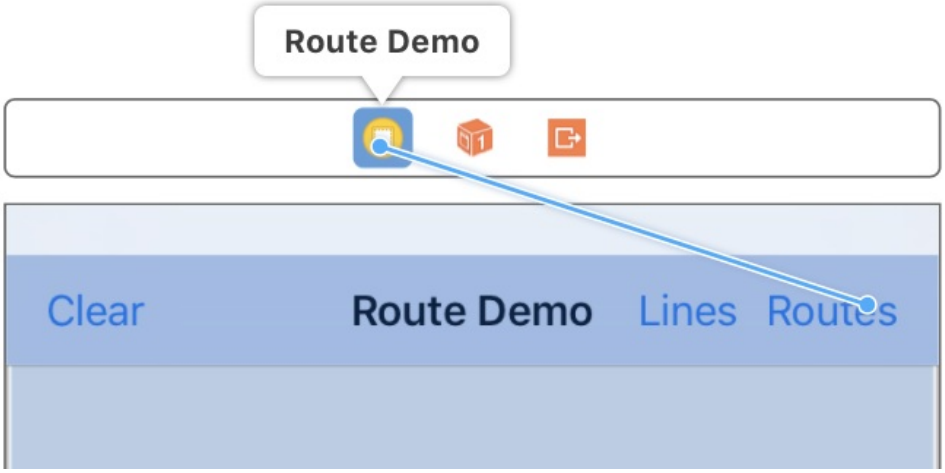


Figure 32.6. Connecting the Routes button with the action method

You're now ready to test the app again. Pin a few locations and tap the Routes button. The app should compute the directions for you.

One thing you should notice is that the map didn't zoom out to fit the routes automatically. You can make the app even better by inserting the following code snippet in the `drawRoute` method (just put it right before `var index = 0`):

```
let polyline = MKPolyline(coordinates: &coordinates, count: coordinates.count)
let visibleMapRect = mapView.mapRectThatFits(polyline.boundingMapRect, edgePadding
: UIEdgeInsets(top: 50, left: 50, bottom: 50, right: 50))
self.mapView.setRegion(MKCoordinateRegion(visibleMapRect), animated: true)
```

Like before, we estimate the preferred size of the map by creating a polyline object for all annotations. Next, we calculate the new aspect ratio of the map view and set the padding for each side by calling the `mapRectThatFits` method. With the new map rectangle, we invoke the `setRegion` method of the map view to adjust the scale.

After the changes, the map should zoom out automatically to display the route within the screen real estate.

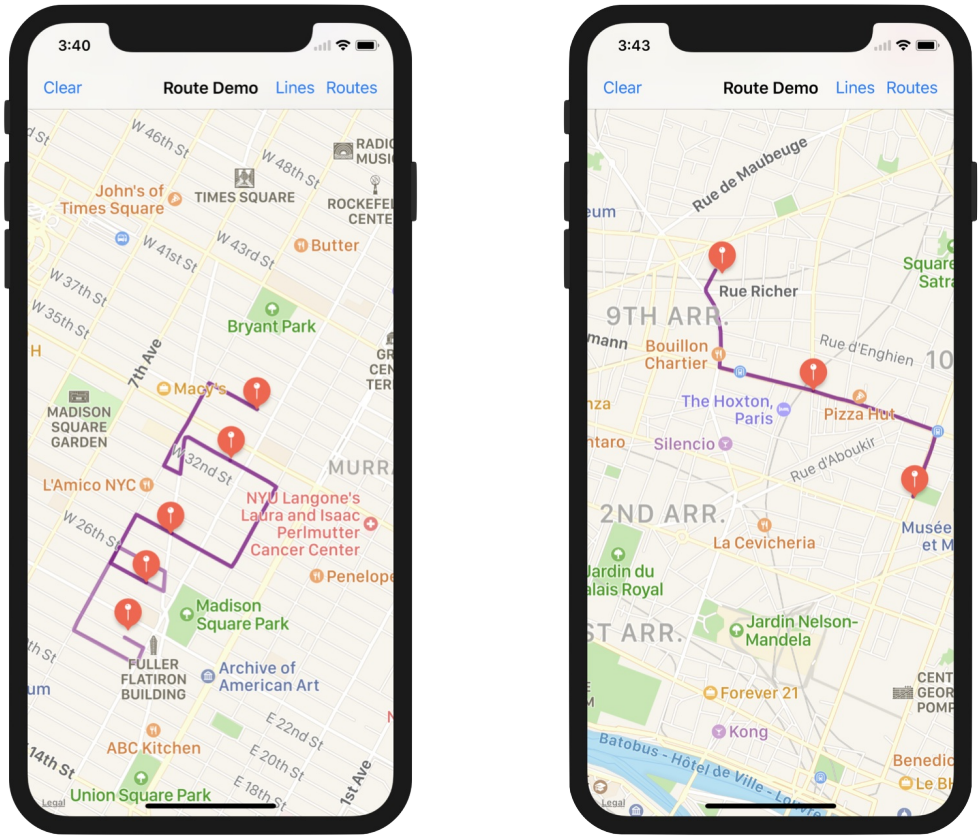


Figure 32.7. Automatically display the route within the screen real estate

Removing Annotations

The app is almost complete. Currently, there is no way for users to clear the annotations. So, insert the `removeAnnotations` method in the `RouteViewController` class:

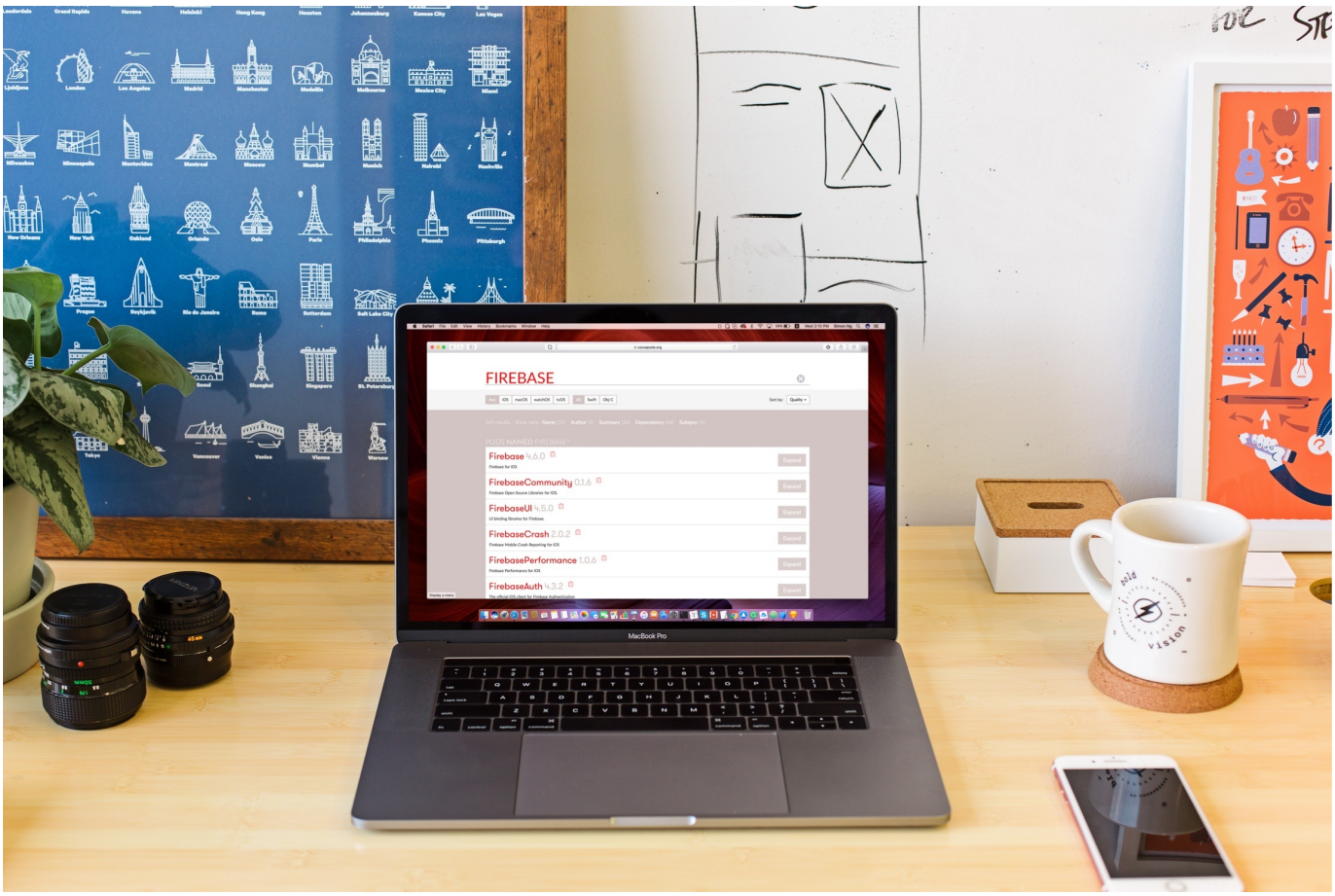
```
@IBAction func removeAnnotations() {

    // Remove annotations and overlays
    mapView.removeOverlays(mapView.overlays)
    mapView.removeAnnotations(annotations)

    // Clear the annotation array
    annotations.removeAll()
}
```

To remove all the annotations from the map view, simply call the `removeAnnotations` method with the `annotations` array. Since the annotations are removed, we have to reset the `annotations` array and clear the overlays accordingly.

Lastly, go to storyboard and connect the *Clear* button with the `removeAnnotations` method. That's it. The demo app is now complete. Test it again on the simulator or a real device.

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/RouteDemo.zip.

Chapter 33
Using CocoaPods in Swift Projects



Understanding CocoaPods, a dependency manager for Swift and Objective-C projects, is a critical skill every iOS developer should have. If you have no experience with CocoaPods, this chapter is written for you.

We're going to take a look at what CocoaPods is, why you should start using it, and how to setup a project with cocoa pods installed!

What is CocoaPods?

First things first, what is CocoaPods? CocoaPods is the dependency manager for your Xcode projects. It helps developers manage the library dependencies in any Xcode projects.

> The dependencies for your projects are specified in a single text file called a Podfile. CocoaPods will resolve dependencies between libraries, fetch the resulting source code, then link it together in an Xcode workspace to build your project.
>
> https://guides.cocoapods.org/using/getting-started.html

You may be confused what this means to you. Let's consider an example.

Suppose you want to integrate your app with Google AdMob for monetisation. AdMob uses the Google Mobile Ads SDK (which is now part of the Firebase SDK). To display ad banners in your apps, the first thing you need is to install the SDK into your Xcode project.

The old fashioned way of doing the integration is to download the Google Mobile Ads SDK from Google and install it into your Xcode project manually. The problem is that the whole procedures are complicated, and the SDK also depends on other frameworks to function properly. Just take a look at the manual procedures as provided by Google:

1. Find the desired SDK in the list.
2. Make sure you have an Xcode project open in Xcode.
3. In Xcode, hit `⌘-1` to open the Project Navigator pane. It will open on left side of the Xcode window if it wasn't already open.
4. Drag each framework from the "Analytics" directory into the Project Navigator pane. In the dialog box that appears, make sure the target you want the framework to be added to has a checkmark next to it, and that you've selected "Copy items if needed". If you already have Firebase frameworks in your project, make sure that you replace them with the new versions.
5. Drag each framework from the directory named after the SDK into the Project Navigator pane. Note that there may be no additional frameworks, in which case this directory will be empty. For instance, if you want the Database SDK, look in the Database folder for the required frameworks. In the dialog box that appears, make sure the target you want this framework to be added to has a checkmark next to it, and that you've selected "Copy items if needed."
6. If the SDK has resources, go into the Resources folders, which will be in the SDK folder. Drag all of those resources into the Project Navigator, just like the frameworks, again making sure that the target you want to add these resources to has a checkmark next to it, and that you've selected "Copy items if needed".
7. Add the -ObjC flag to "Other Linker Settings": a. In your project settings, open the Settings panel for your target b. Go to the Build Settings tab and find the "Other Linking Flags" setting in the Linking section. c. Double-click the setting, click the '+' button, and add "-ObjC"
8. Drag the `Firebase.h` header in this directory into your project. This will allow you to `#import "Firebase.h"` and start using any Firebase SDK that you have.
9. If you're using Swift, or you want to use modules, drag module.modulemap into your project and update your User Header Search Paths to contain the directory that contains your module map.
10. You're done! Compile your target and start using Firebase.

CocoaPods is the dependency manager that saves you from doing all the above manual procedures. It all comes down to a single text file called *PodFile*. If you use CocoaPods to install the Google Mobile Ads SDK, all you need is create a PodFile under your Xcode project with the following content:

```
source 'https://github.com/CocoaPods/Specs.git'

platform :ios, '9.0'

target 'BannerExample' do
  pod 'Firebase/Core'
  pod 'Firebase/AdMob'
end
```

When you run the `pod install` command, CocoaPods will download and install the specified libraries/dependencies for you.

This is why CocoaPods has its place. It simplifies the whole process by automatically finding and installing the frameworks, or dependencies require. You will experience the power of CocoaPods in a minute.

Setting Up CocoaPods on Your Mac

CocoaPods doesn't come with the macOS. However, setting up CocoaPods is pretty simple and straightforward. To install Cocoapods, navigate to the terminal and type the following commands:

```
sudo gem install cocoapods
```

This line of command installs the CocoaPods gem on your system. CocoaPods was built with Ruby, so it relies on the default Ruby available on macOS for installation. If you're familiar with Ruby, gems in Ruby are similar to pods in CocoaPods.

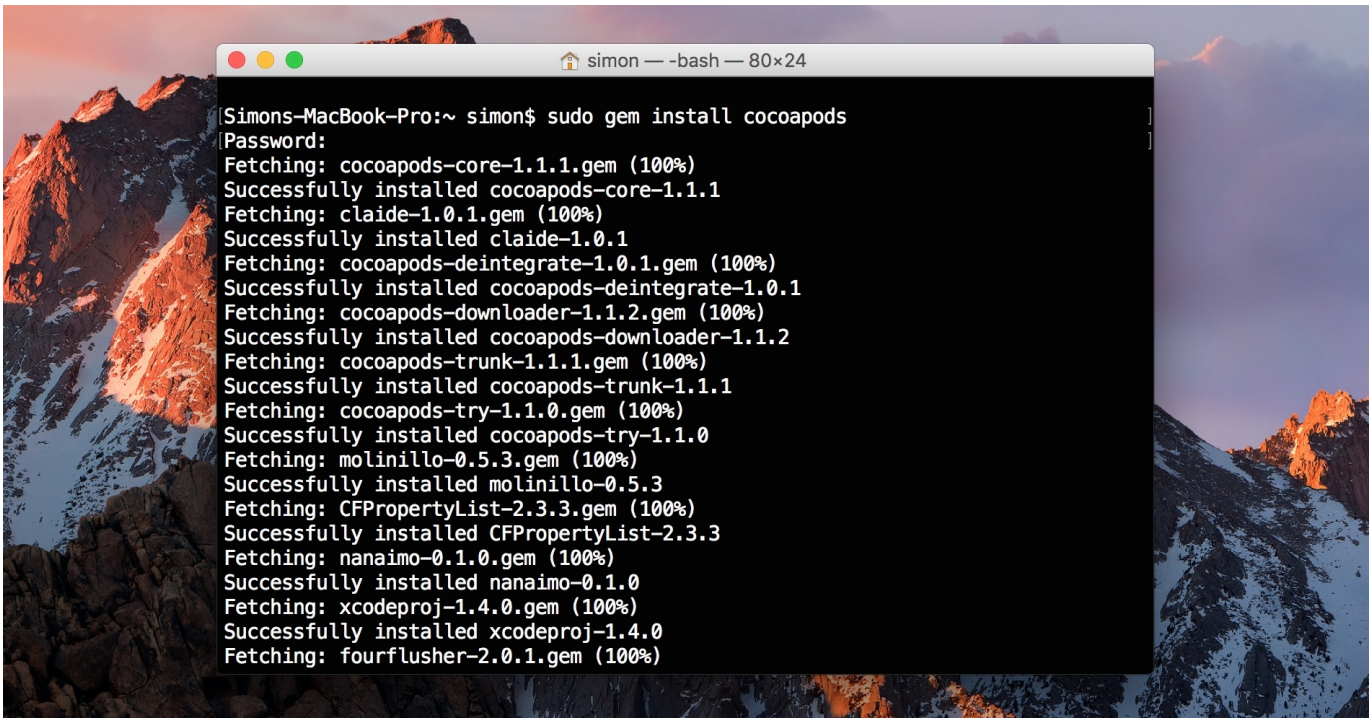


Figure 33.1. Installing CocoaPods

It'll take several minutes to finish the installation. Just be patient, grab a cup of coffee, and wait the whole process to complete.

Using CocoaPods for Xcode Projects

Once you have CocoaPods installed on your Mac, let's see how to use it. We will create a sample project and demonstrate how you can install the Google Mobile Ads SDK in the project using CocoaPods.

First, create a new Xcode project using the Single View Application template and name it *GoogleAdDemo*. Close the project and back in terminal. Use the cd (change directory) command to navigate to your new Xcode project. Assuming you save the project under Desktop, here is the command:

```
cd ~/Desktop/GoogleAdDemo
```

Next, we need to create what's called a *Podfile*. A Podfile is a file that lives in the root of your project and is responsible for keeping track of all the pods you want to install. When you ask CocoaPods to install any pods or updates to your existing pods, CocoaPods will look to the Podfile for instructions.

To create a Podfile, simply type the following command:

```
pod init
```

CocoaPods then generates the Podfile like this:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '13.0'

target 'GoogleAdDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for GoogleAdDemo

end
```

That's the basic structure of a Podfile. All you need to do is edit the file and specify your required pods. To use the Google Mobile Ads SDK, you have to edit the file like this:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '13.0'

target 'GoogleAdDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for GoogleAdDemo
  pod 'Firebase/Core'
  pod 'Firebase/AdMob'
end
```

We simply specify the `Firebase/Core` and `Firebase/AdMob` pods for the GoogleAdDemo project.

Before we move to the next step, let us go through the above configuration file:

- The Podfile describes the dependencies of the targets of your Xcode project. Therefore, we have to specify the target, which is GoogleAdDemo for this app project.
- The `use_frameworks` option tells CocoaPods to use frameworks instead of static libraries. This is required for Swift projects.
- The couple of lines that we have just inserted lets CocoaPods know that we need to use the Core and AdMob pods. You may wonder how do you know the pod name. Normally you can look it up in the documentation of the pod (here it is Google) or simply search it on cocoapods.org.

Note: You can check out the Google documentation at https://firebase.google.com/docs/admob/ios/quick-start.

Now that you should have a better understanding of the pod file, type the following command in the terminal to complete the last step:

```
pod install
```

Cocoapods will now install the specified pods. After downloading the required pods, it creates a workspace file named `GoogleAdDemo.xcworkspace`. This workspace file bundles your original Xcode project, the Firebase/AdMob library, and its dependencies.

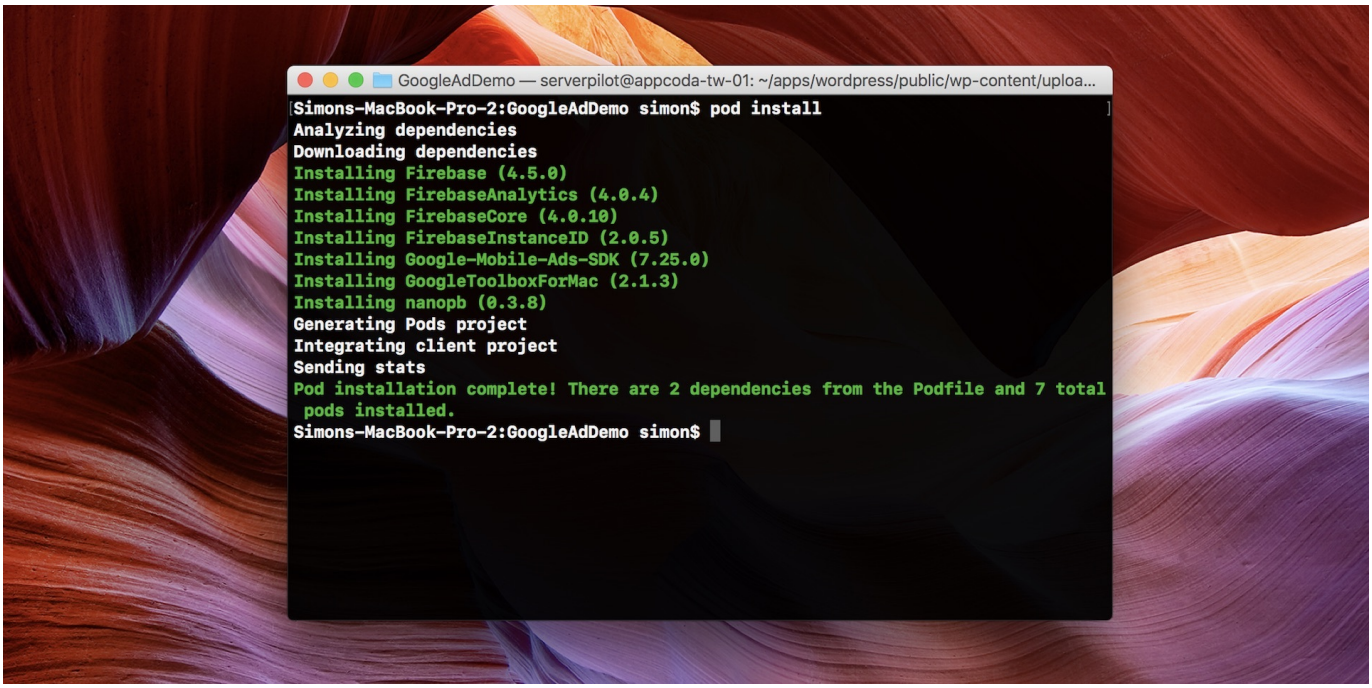


Figure 33.2. Installing the pods

Using the Xcode Workspace for Development

From now and onwards, make sure you use the generated workspace file for development. It is the file that bundles the original Xcode project and the Pod project with the required libraries.

If you open `GoogleAdDemo.xcworkspace` with Xcode, you should find both the *GoogleAdDemo* project and the *Pod* project, which contains the Firebase library.

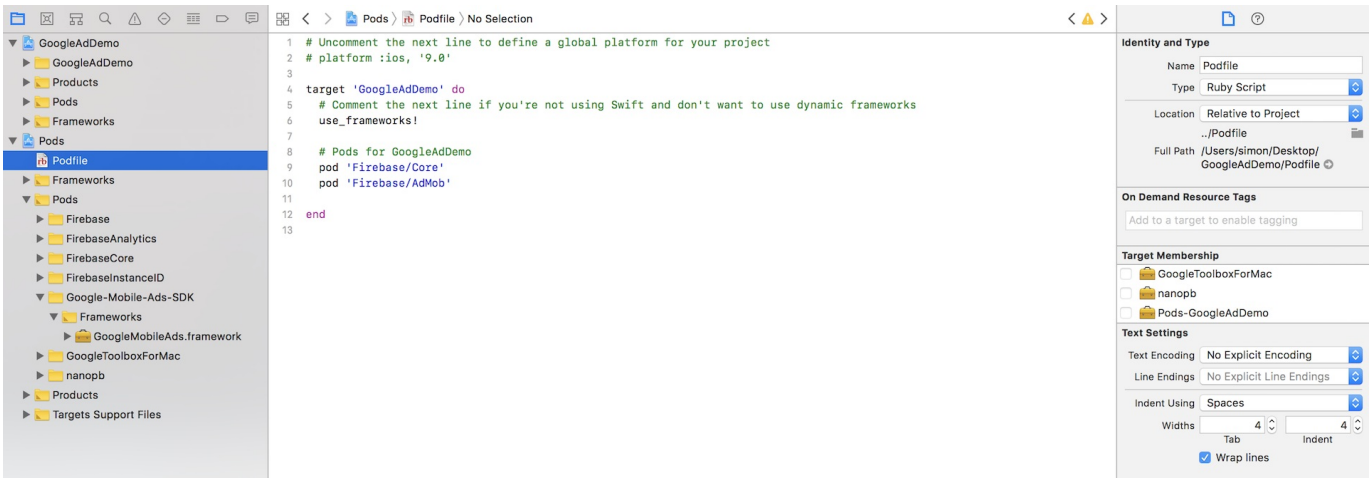
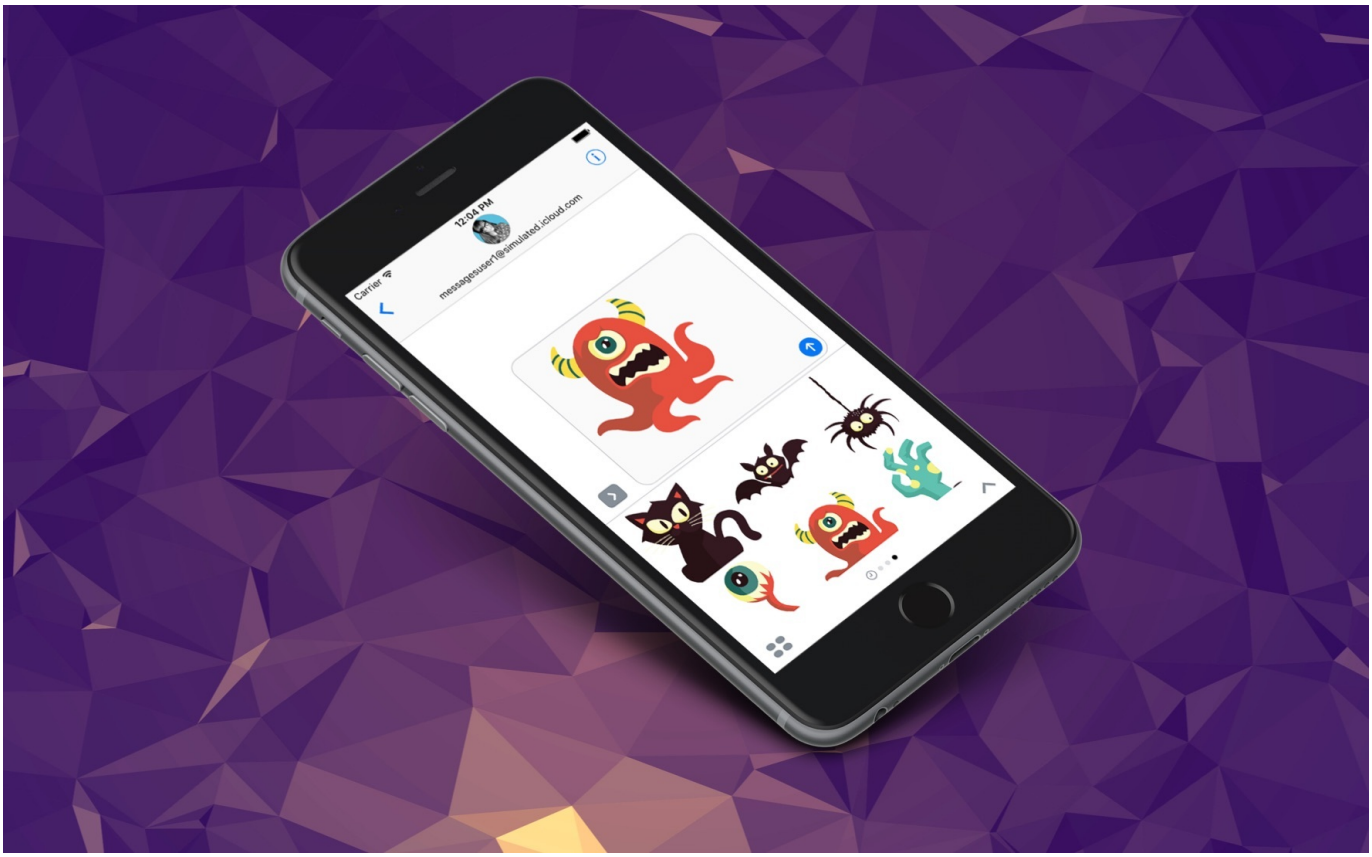


Figure 33.3. GoogleAdDemo.xcworkspace bundles both GoogleAdDemo and Pod projects

Wrapping Up

CocoaPods is an incredibly simple tool that every iOS developer should have in his/her backpocket. I hope this chapter is clear cut and easy to follow. In future, when you need to use any third party libraries, consider to use CocoaPods rather than manually install the libraries.

Chapter 34
Building a Simple Sticker App



Back in 2016, the introduction of the Message framework in iOS 10 was one of the biggest announcements. Developers can finally create app extensions for Apple's built-in Messages app. By building an app extension, you let users interact with your app right in the Messages app. For example, you can build a message sticker extension that allows your users to send stickers while communicating with his/her friends in Messages. Or if you already developed a photo editing app, you can now write an extension for users to edit photos without leaving the Messages app.

The support of extension opens up a lot of opportunities for app developers. Apple even introduced a separate App Store for iMessage. You can sell stickers and app extensions through the app store, which is dedicated for iMessage.

To build an app extension for Messages, you will need to make use of the new Message framework. The framework supports two types of app extensions:

- Sticker packs
- iMessage apps

In this chapter, I will focus on showing you how to build a sticker pack. For the chapter that follows, we will dive a little bit deeper to see how you can develop an iMessage app.

Before moving on, I have to say that Apple makes it very easy for everyone to build sticker packs. Even if you do not have any Swift programming experience, you'll be able to create your own sticker pack because it doesn't need you to write a line of code. Follow the procedures described in this chapter and learn how to create a sticker extension.

Preparing the Sticker Images

Creating a sticker app is a two-part process:

- First, you prepare the sticker images, that conforms to Apple's requirements.
- Secondly, you create a sticker app project using Xcode.

Let's start with the first part. Messages supports various sticker image formats including PNG, GIF, and JPG, with a maximum size of 500KB. That said, it is recommended to use images in PNG format.

Sticker images are displayed in a grid-based browser. Depending on the image size (*small*, *regular* or *large*), the browser presents the images in 2, 3 or 4 columns.

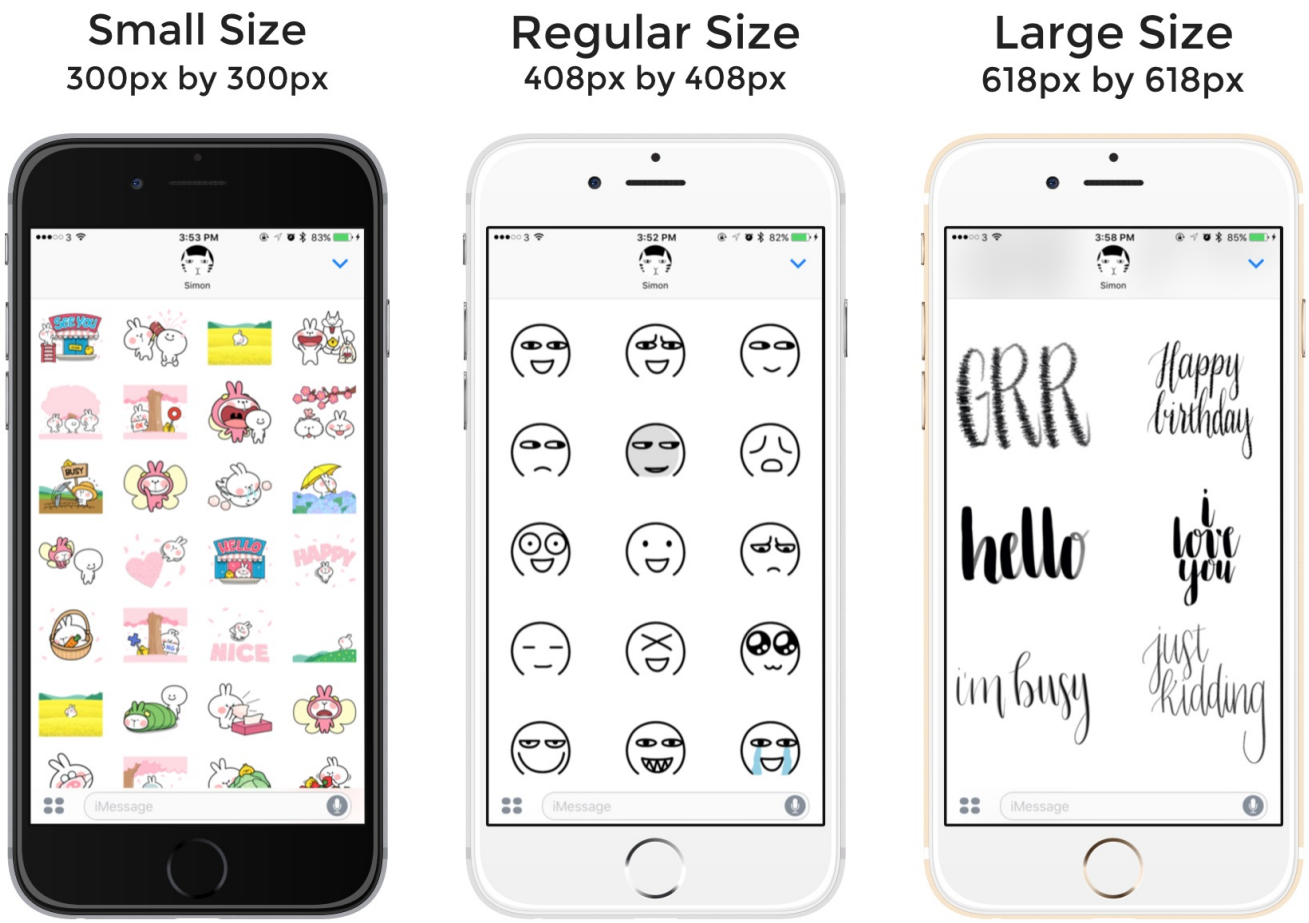


Figure 34.1. How the image size affects the sticker presentation

Other than size, the other thing you have to consider, while preparing your sticker images, is whether the images are static or animated. Messages supports both. For animated images, they should be either in GIF or APNG format.

We will discuss more on animated sticker images in the later section. So let's focus on the static ones first. Now choose your own images and resize them to a size that best fits your stickers.

If you don't want to prepare your own images, you can download this image pack from http://www.appcoda.com/resources/swift4/StickerPack.zip.

Building the Sticker Package Project Using Xcode

Assuming your sticker images are ready, we're now going to build the sticker app. Fire up Xcode and create a new project. To build a sticker pack, please choose iOS > Application and then select Sticker Pack Application.

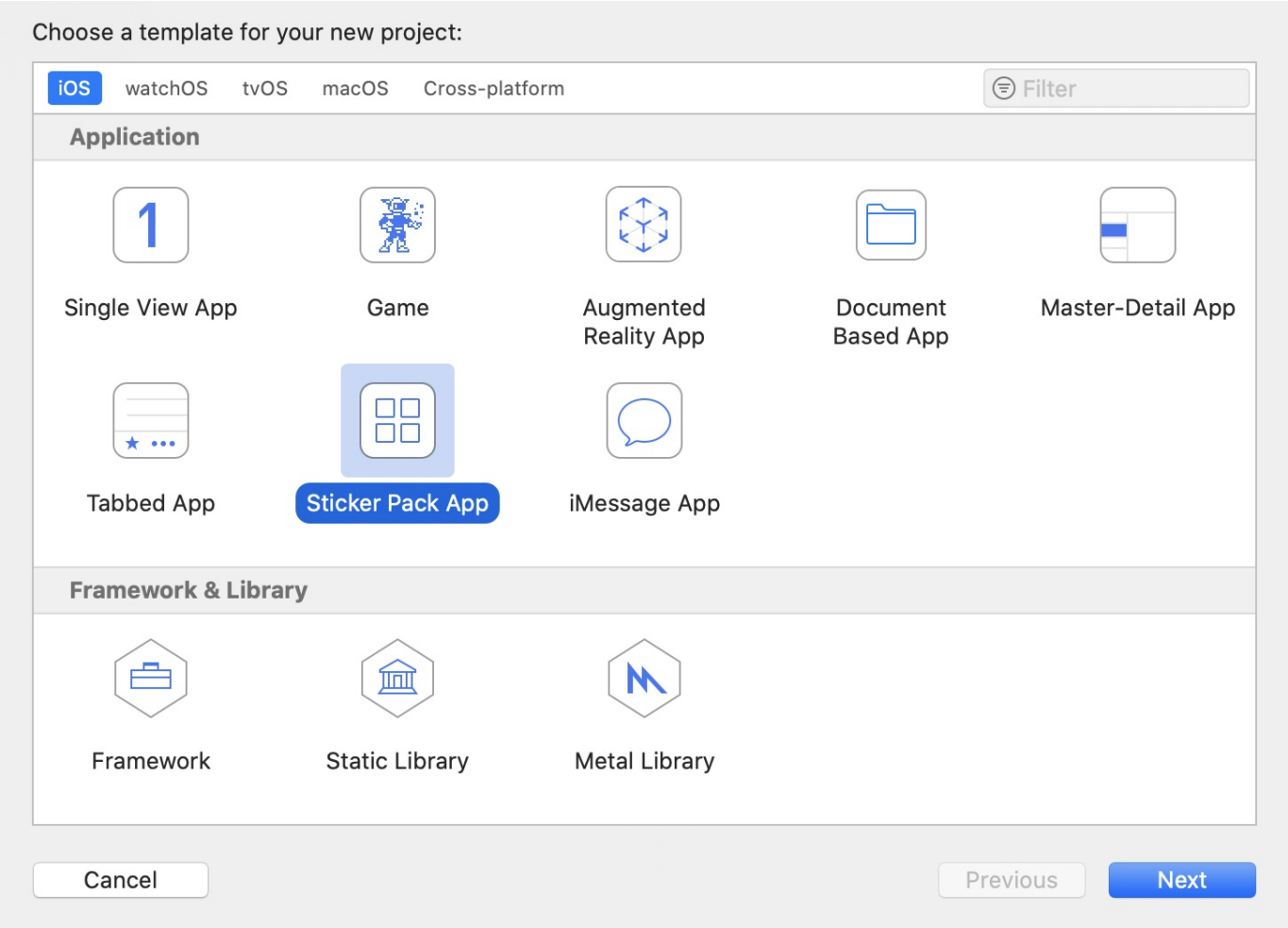


Figure 34.2. Choosing the sticker pack app template

Next, fill in the project name. For this demo, I use the name *CuteSticker* but you can choose whatever name you prefer.

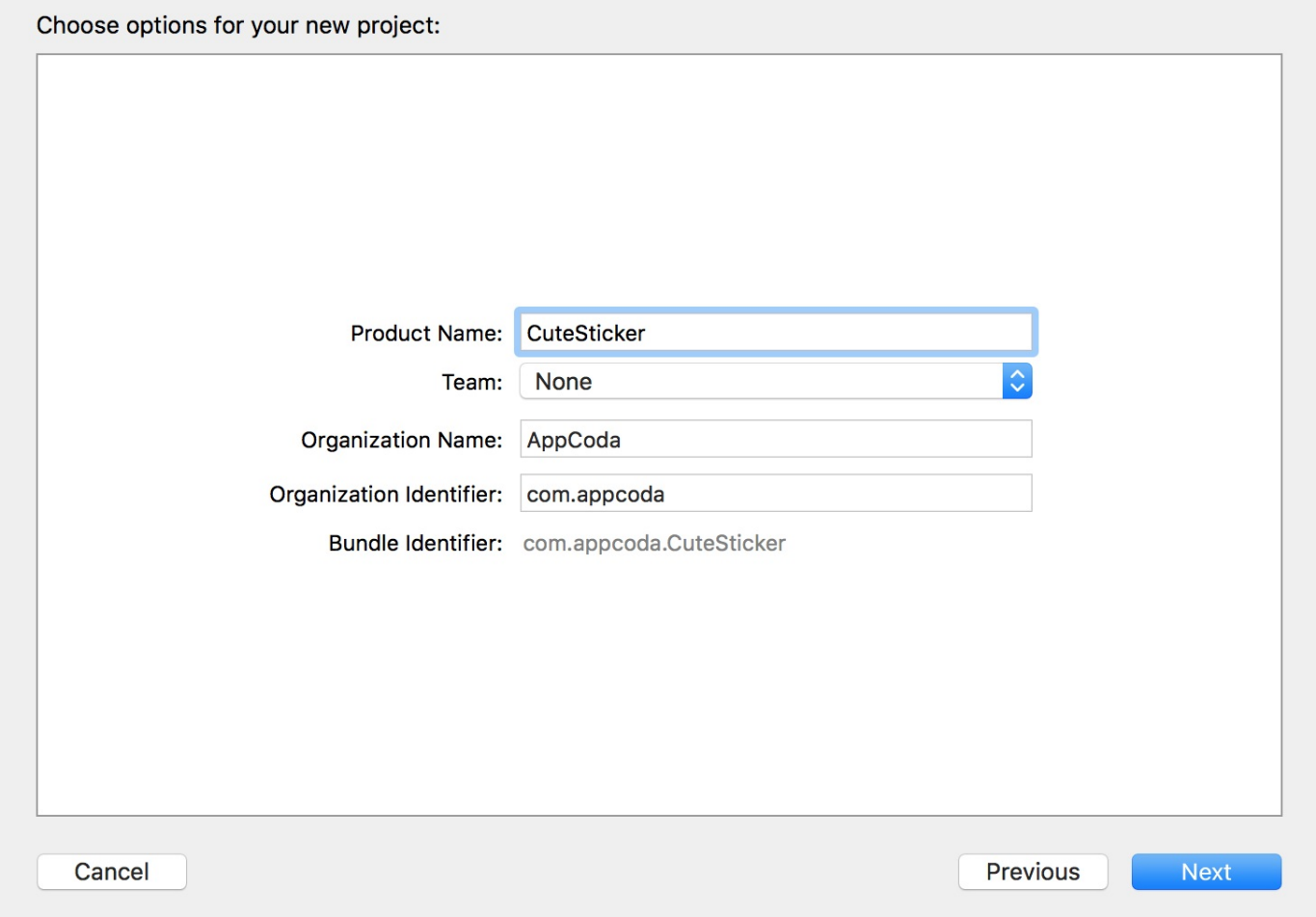


Figure 34.3. Filling in the project details

Click *Next* to continue and choose a folder to save your project. Xcode then generates a sticker app project for you.

Adding Images to the Sticker Pack

Once your Xcode project is created, you'll see two folders in the project navigator. Click `Stickers.xcstickers` and then select the Sticker Pack folder. This is where you put your image files.

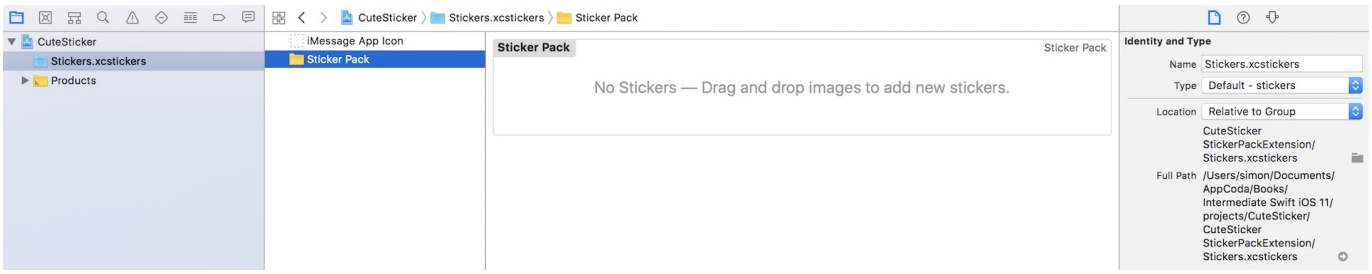


Figure 34.4. The sticker pack folder

Assuming you've downloaded our image pack, unzip it in Finder. Then select all the images and drag them into the Sticker Pack folder.

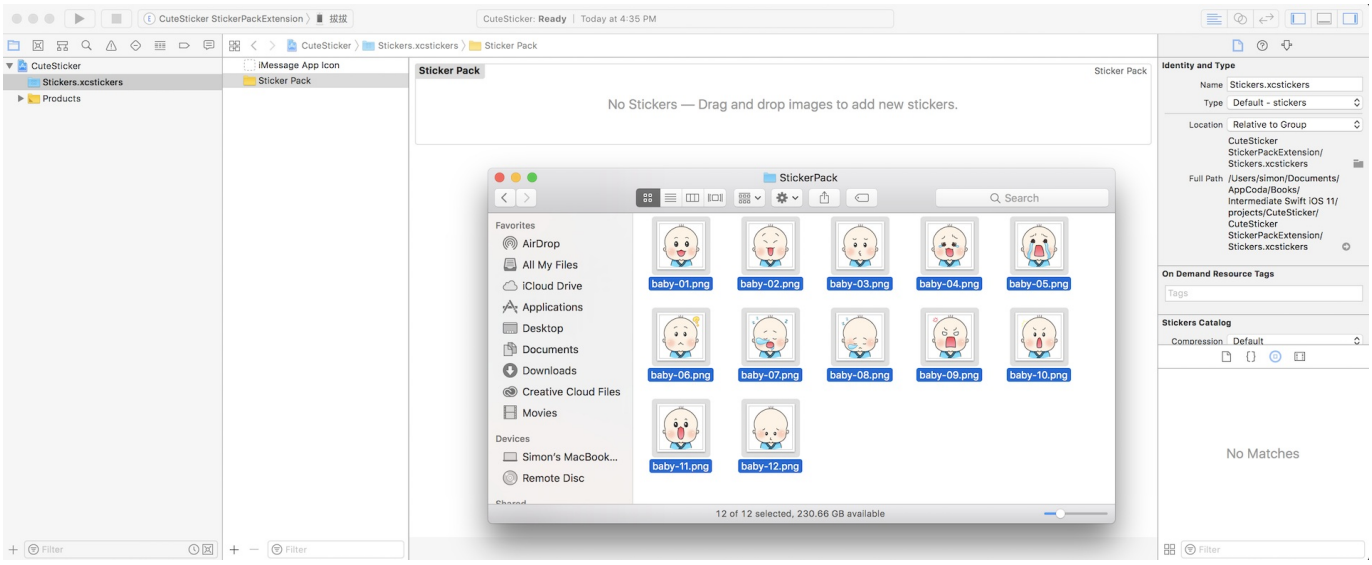


Figure 34.5. Drag and drop the sticker images

It's almost done. Now, select the *Sticker Pack* folder, and then choose the Attributes inspector. By default, the sticker size is set to *3 Column*. For the demo images, the size is 300px by 300px. So the best choice is to set the sticker size to *4 Column*, though you can still keep it to *3 Column* if you want.

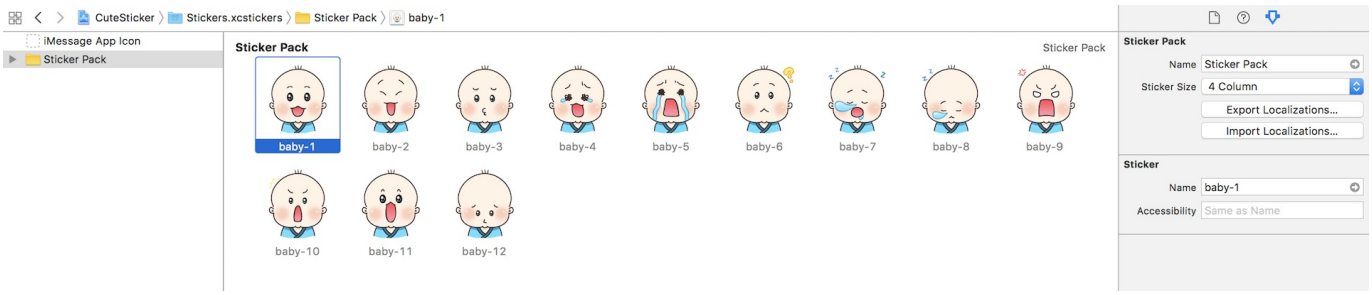


Figure 34.6. Set the appropriate sticker size

Adding App Icons

Lastly, your sticker pack must have an app icon. Again, for demo purpose, I have prepared a sample app icon and you can download the app icon from http://www.appcoda.com/resources/swift4/StickerAppIcon.zip. If you prefer to create your own icon, make sure you prepare the icon with different sizes:

- 1024×768 points (@1x) for Messages App Store
- 27×20 points (@1x, @2x, @3x) for Messages
- 32×24 points (@1x, @2x, @3x) for Messages
- 29×29 points (@1x, @2x, @3x) for iPhone/iPad Settings
- 60×45 points (@2x, @3x) for Messages (iPhone)
- 67×50 points (@1x, @2x) for Messages (iPad)
- 74×55 points (@2x) for Message (iPad Pro)

To simplify the icon preparation, you can download iMessage App Icon template (https://developer.apple.com/ios/human-interface-guidelines/resources/) from Apple.

After you download our demo app icon pack, unzip the file and drag all the app icon files to iMessage App Icon.

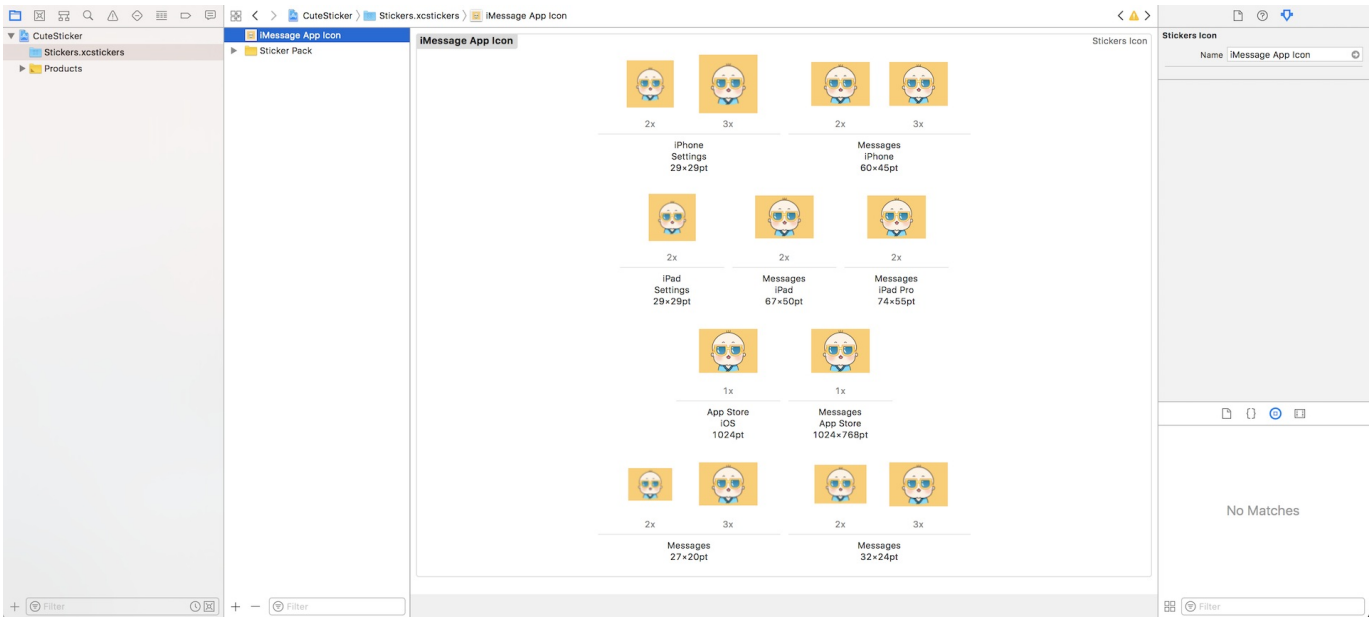


Figure 34.7. Adding the app icon

Testing the Sticker Pack

That's it! Now that you've created a sticker pack for Messages, it's time to test it. You do not need a real device to run the test. Xcode has a built-in simulator to test any iMessage app extension. Choose a simulated device (e.g. iPhone 8/11) and hit the Run button to start the testing.

Since the sticker pack is an app extension, you can't execute it as a standalone application. When you run the sticker pack, Xcode loads the sticker pack into the Messages app and automatically launches it on the simulator. In case if you don't see the sticker pack, click the lower left button (i.e. App Shelf button) to reveal the stickers.

Figure 34.8. Opening the sticker pack

The Messages app in the simulator has come with two simulated users: *Kate Bell* and *John Appleseed*. The default user is set to Kate. To send a sticker to John, choose a sticker from the message browser and press return key to send it. Then go to John. You should find the stickers you've just sent.

Figure 34.9. Sending and receiving the stickers in the built-in simulator

In some cases, you may experience the following error when launching the sticker app on the simulator:

```
2017-11-14 18:31:16.157871+0800 MobileSMS[30336:6142873] *** Terminating app due t
o uncaught exception 'NSInvalidArgumentException', reason: 'attempt to scroll to i
nvalid index path: <NSIndexPath: 0x600000429060> {length = 2, path = 0 - 922337203
6854775807}'
```

If you encounter the error, you can use the following workaround to test the sticker pack:

- In the simulator, go back to the home screen and tap the Messages icon to launch the Messages app.
- Click the `...` button in the App drawer.
- Then click the *Edit* button and flip the switch of the *CuteSticker* option to ON.

You will then see *CuteSticker* added in the app drawer.

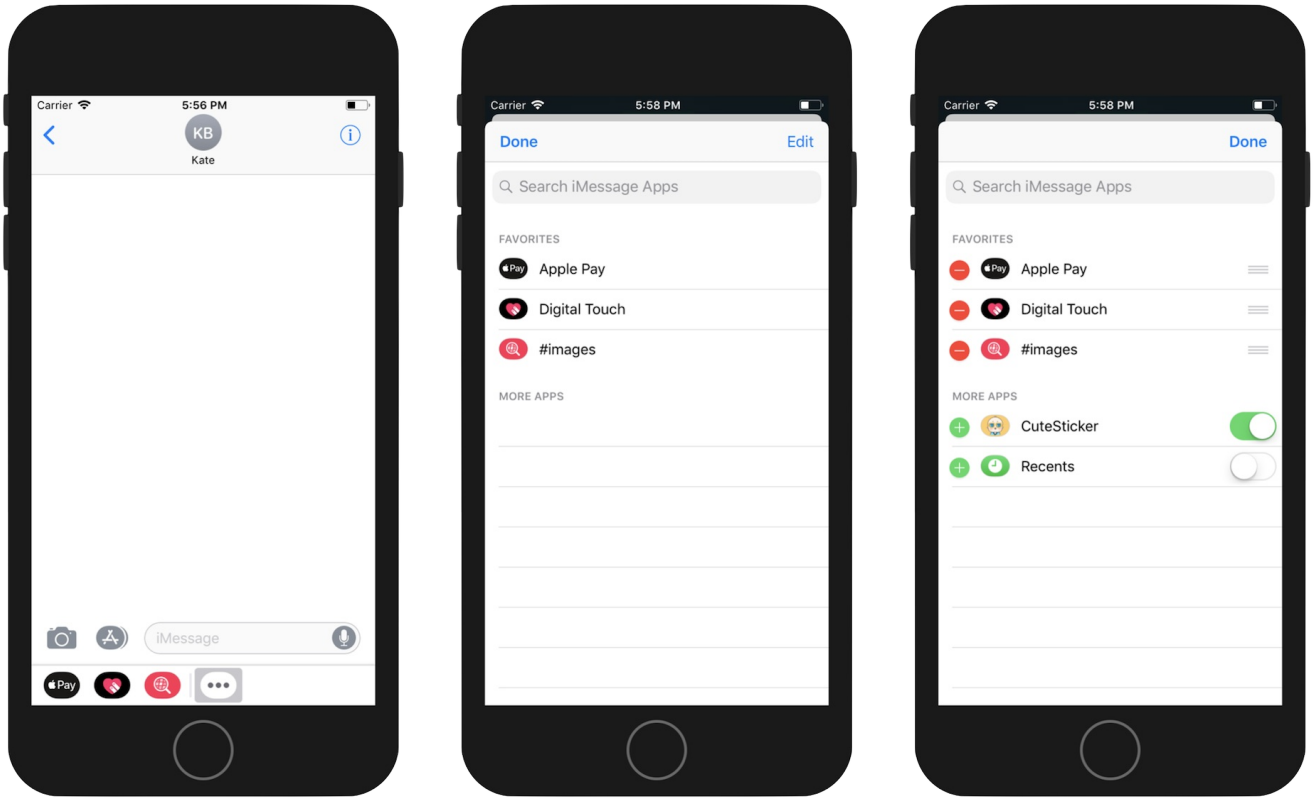


Figure 34.10. Enable CuteSticker by tapping the ... button in the app drawer

Enhancing Your Sticker Pack with Animated Images

As mentioned at the beginning, not only can you bundle static images in your sticker pack, the Messages app supports animated sticker images. If you already have some animated GIFs or APNGs, simply add the image to the stick pack. Xcode will recognize it and display the animation. As an example, you can download this free image (https://media.giphy.com/media/26BRyFg4vjuGG7HK8/giphy-downsized.gif) and add it to the sticker pack to have a test.

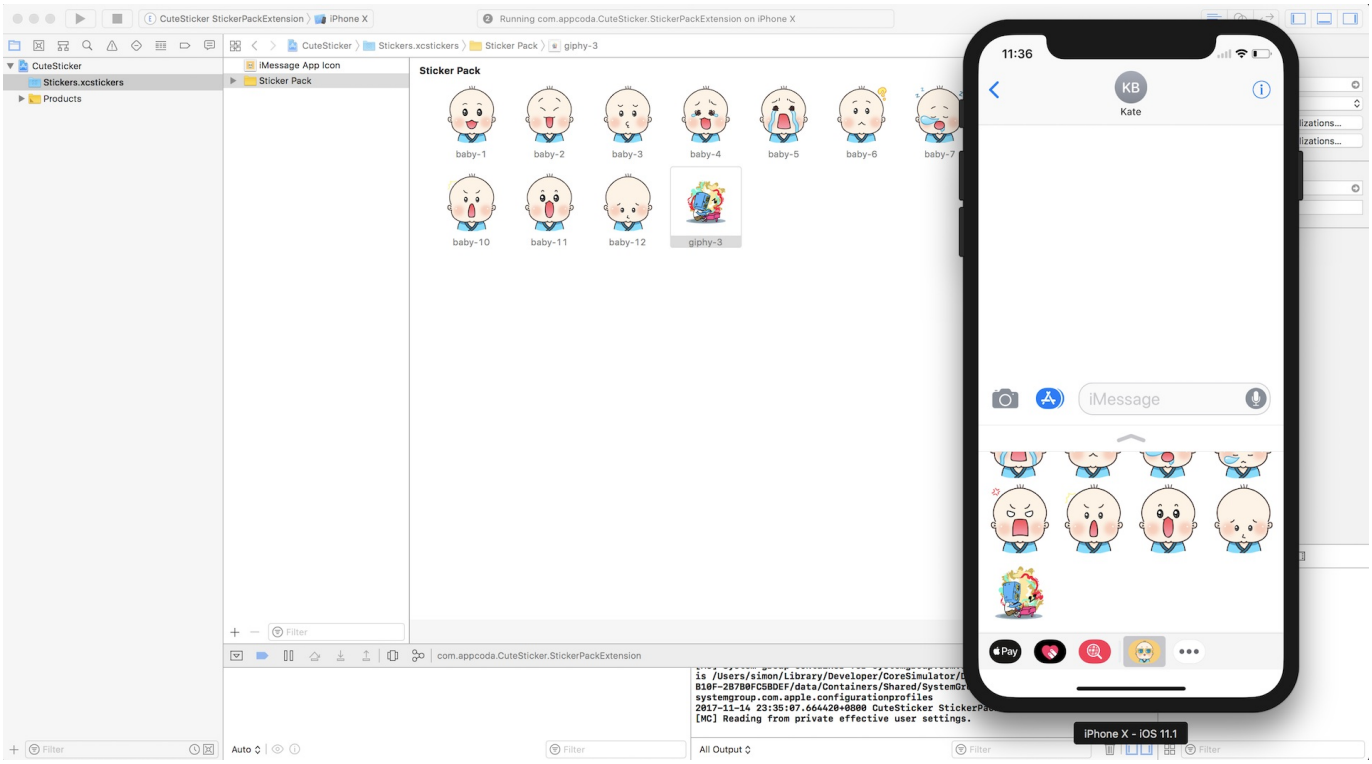


Figure 34.11. Using animated stick images

An alternative approach for creating animated images is to create a sticker sequence. Go back to your sticker pack. Right-click on any blank area to bring up the option menu. Choose *Add Assets* > *New Sticker Sequence*. This creates a sticker sequence for you to add a sequence of images.

As an example, you can download this image pack (http://www.appcoda.com/resources/swift4/StickerAnimatedImages.zip) to try it out. Unzip the pack, and drag all the images to the sticker sequence. Xcode allows you to preview the resulting animation right in the sticker pack. Place the cursor over the sticker sequence and click the Play button to preview the animation.

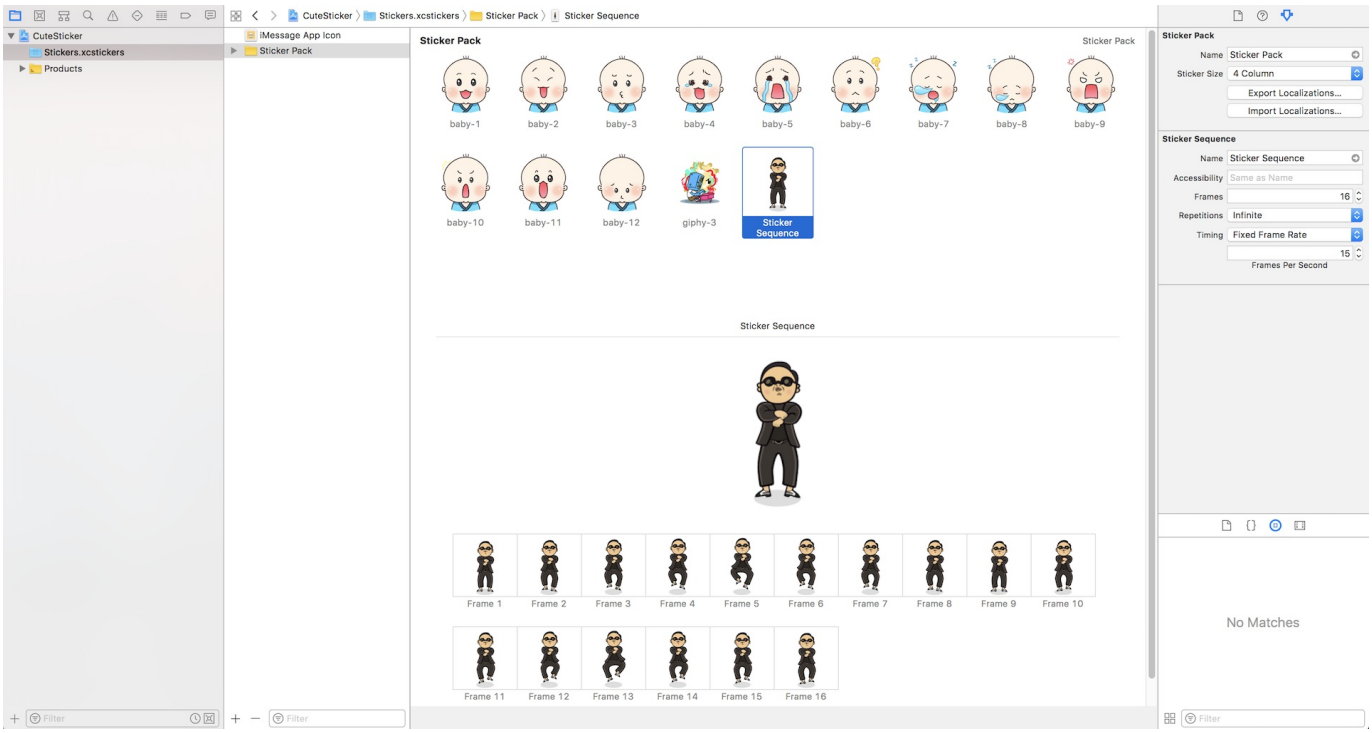


Figure 34.12. Using sticker sequences to create an animation

You can run the sticker pack in the simulator again. The Messages app will display both images as an animation.

Adding a Sticker Pack for an Existing App

Up till now, you've learned how to create an independent sticker app. What if you already have an existing app and want to bundle a sticker pack? How can you do that?

Xcode lets you build a sticker extension for any existing apps. Assuming you've opened an existing project in Xcode (e.g. VisualEffect), you can first select your project in the project navigator, and then go up to the menu. Select *Editor* > *Add Target*....

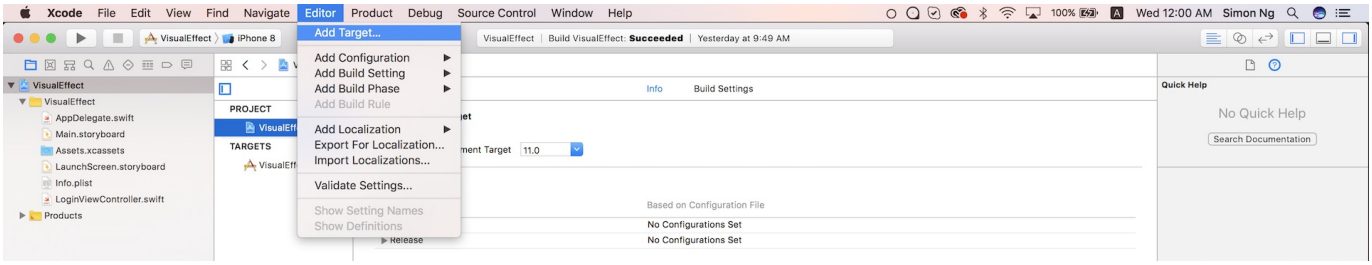


Figure 34.13. Adding a new target

You'll then be prompted to choose a project template. Again, pick the *Sticker Pack Extension* template to proceed.

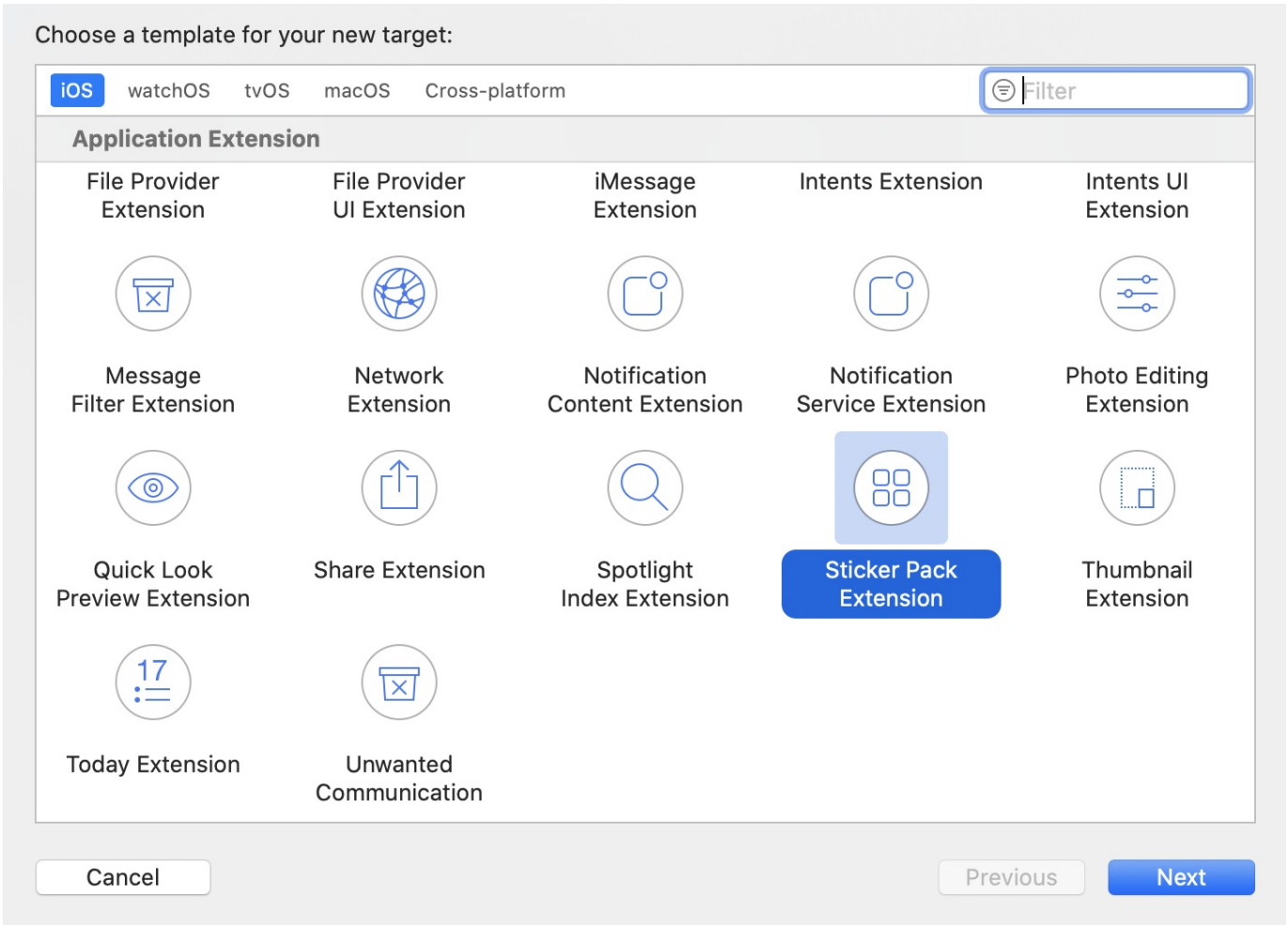


Figure 34.14. Choosing the Sticker Pack Application template

Next, follow on-screen instructions to give your product a name. This is the name of your sticker pack that will be shown in Messages. Finally, hit the *Activate* button when Xcode prompts you to activate the new scheme. Xcode will add the *Stickers.xcstickers* folder in your existing project. All you need to do is drag your sticker images into the sticker pack.

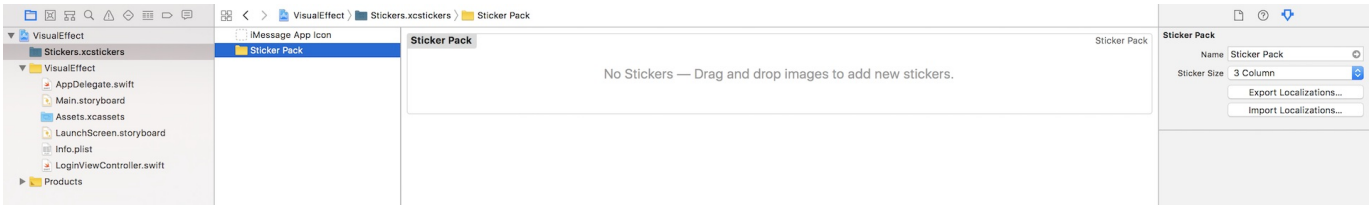


Figure 34.15. Sticker pack

To test the sticker app, you can choose the *StickerPackExtension* scheme and then run the app in any of the simulators.

Summary

You have just learned how to create an app extension for the Messages app in Xcode. As you see, you don't even need to write a line of code to create a sticker pack. All you need is prepare your own images (animated or static) and you're ready to build a sticker pack.

Even though the Message App Store has been launched for more than a year, it is still a good time to start building your own sticker packs, especially you have an existing game or some iconic characters for your brand. Having a sticker pack on the Message App Store will definitely give your app more exposure.

Sticker pack is just one type of the iMessage app extensions. In the next chapter, we will see how to create a more complicated extension for Messages.

For reference, you can download the demo project from http://www.appcoda.com/resources/swift5/CuteSticker.zip.

Chapter 35
Building iMessage Apps Using Messages Framework

As mentioned in the previous chapter, not only can you create a sticker pack, the Messages framework allows developers to build another kind of messaging extensions, known as iMessage apps, that let users interact with your app without leaving the Messages app.

Let me give an example, so you will better understand what an iMessage app can do for you.

You probably have used the Airbnb app before.Let's say, you're now planning the next vacation with your friends. You come across a really nice lodging place to stay, and you want to ask your friends for opinions. So what would you do to share the place?

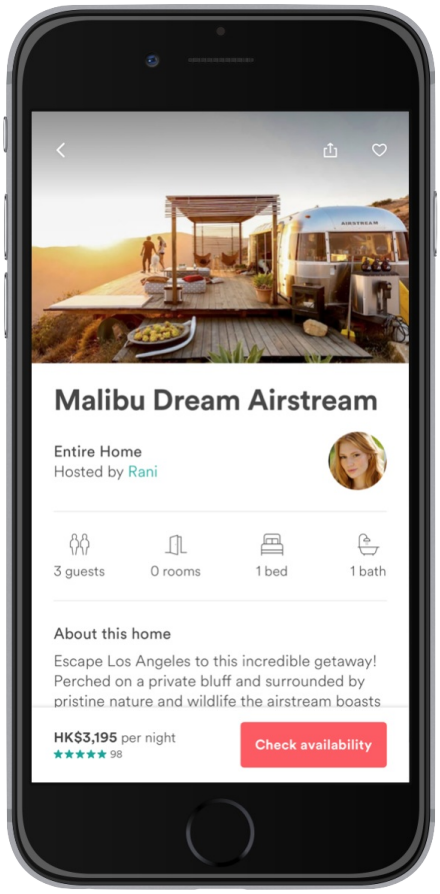


Figure 35.1. Airbnb App

For now, you may capture a screenshot and then send it over to your friends through Messages, Whatsapp or any messaging clients. Alternatively, you may use the built-in sharing option of the Airbnb app to share a link to your friends, so he/she can view the lodging place on airbnb.com.

Both ways are not perfect, however.

The screenshot may only show partial information of the lodging place. If you send the link to your friends over Messages, it should display the complete information of the place. But opening a link in iOS usually means switching to the mobile Safari browser. The user will need to view the details in Safari, and then switch back to the Messages app to reply the message. This isn't a big deal. That said, as developers, we always look for ways to improve the user experience of an app.

Starting from iOS 10, the Airbnb app comes with a message extension or what we called the iMessage app. The updated app lets you share any of the recently viewed hotels/lodging options right in the Messages app. Figure 35.2 displays the workflow.

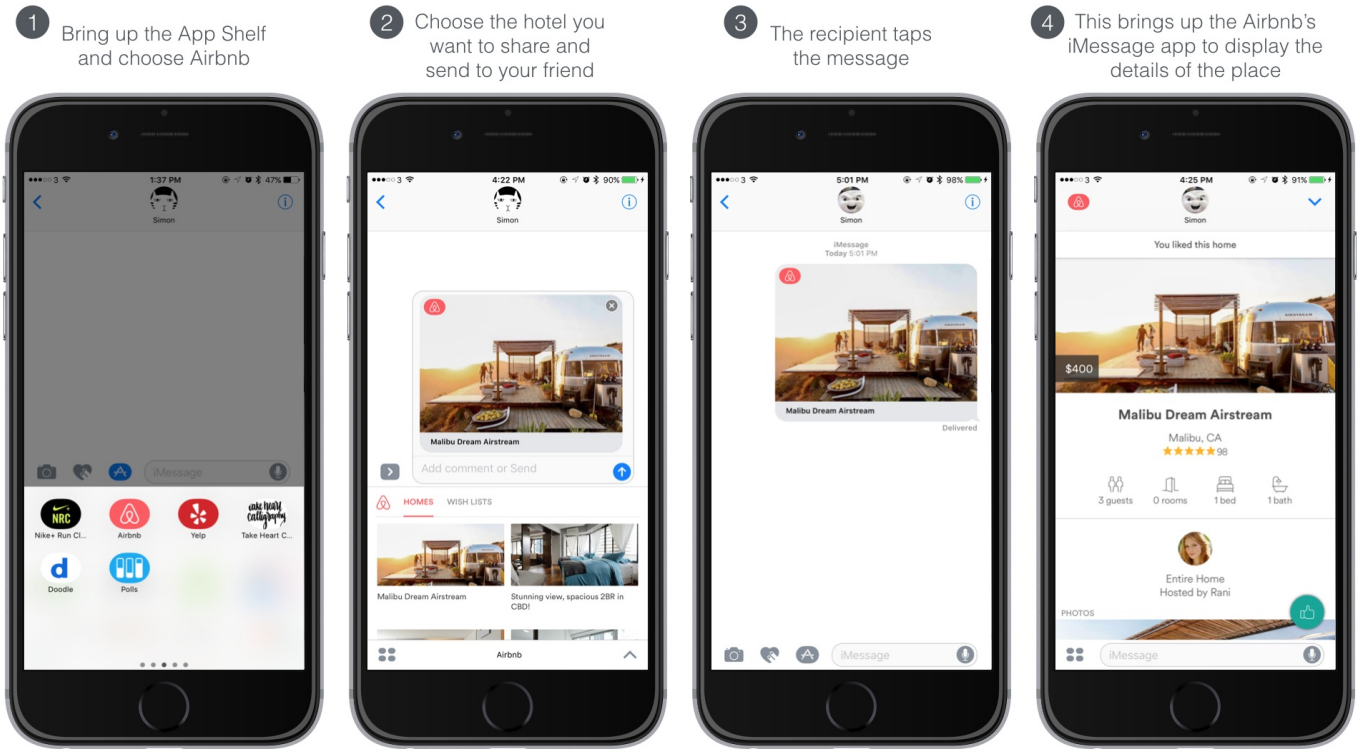


Figure 35.2. Sharing a lodging place using Airbnb's iMessage app

What's interesting is on the receiving side. Assuming the recipient's device has Airbnb installed, he/she can reveal the details of the lodging place right in the Messages app. Furthermore, if the recipient loves the place, he/she can tap the Like button and reply back.

Cool, right? Everything is now done right in Messages, without even launching the Airbnb app or switching to the mobile browser.

You may wonder what happens if the recipient doesn't have the Airbnb app installed?

Messages will bring up the App Store and suggest the user to download the Airbnb app. As you may realize, this is a new way to promote your app. When the recipient receives the message, it is likely he/she will install the app so as to view the message. Your app

user just helps you promote your app by sending messages.

Now that you should have a better idea of iMessage apps and why it is important to build for your existing app, let's dive into the implementation.

The Demo App

We will make use of the Icon Store app that we built in chapter 18/19 as a demo. If you haven't read those chapters, it is time to take a look. Even though it is not mandatory, the better you understand the chapter, the better you will understand what I'm going to discuss.

If you're ready to get started, download the Icon Store app from http://www.appcoda.com/resources/swift5/CollectionViewSelection.zip. Unzip it and compile the demo to see if it works.

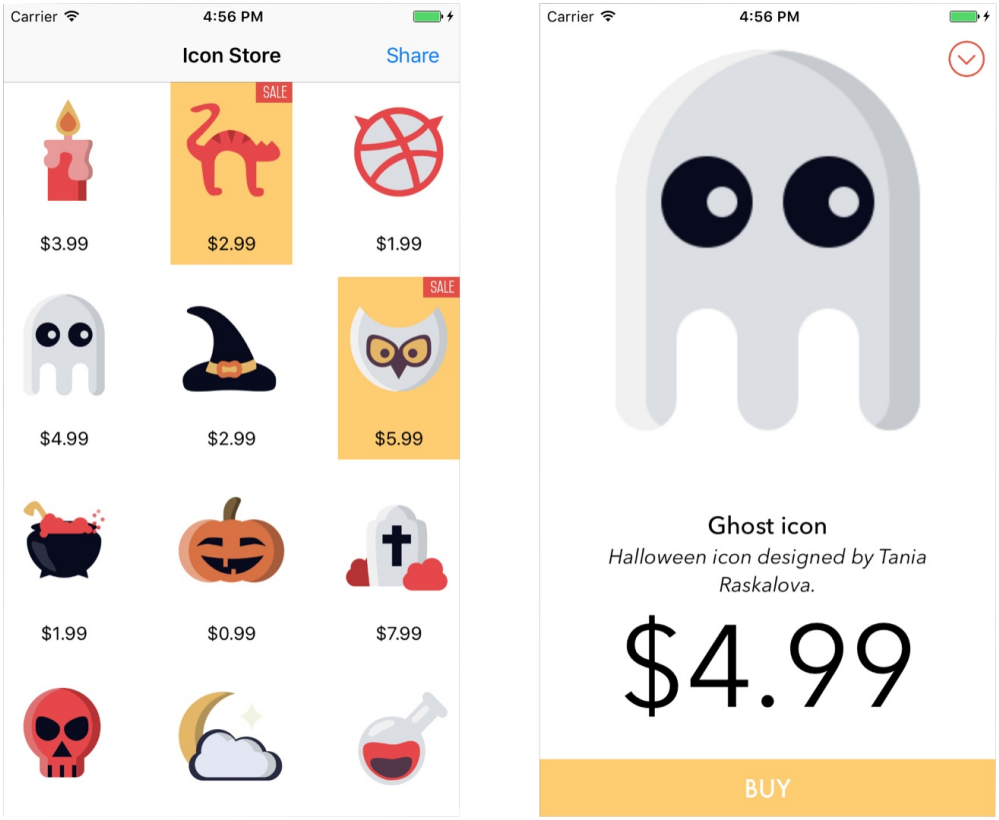


Figure 35.3. The Icon Store app

Meanwhile, if you want to share a favorite icon to another user, you probably do a screen capture and send the screenshot over Messages. What we are going to do is build an iMessage app such that users can access the icons right the Messages app. Users can pick an icon and send it to another user. On the receiving end, the recipient can reveal the icon details simply by tapping the message.

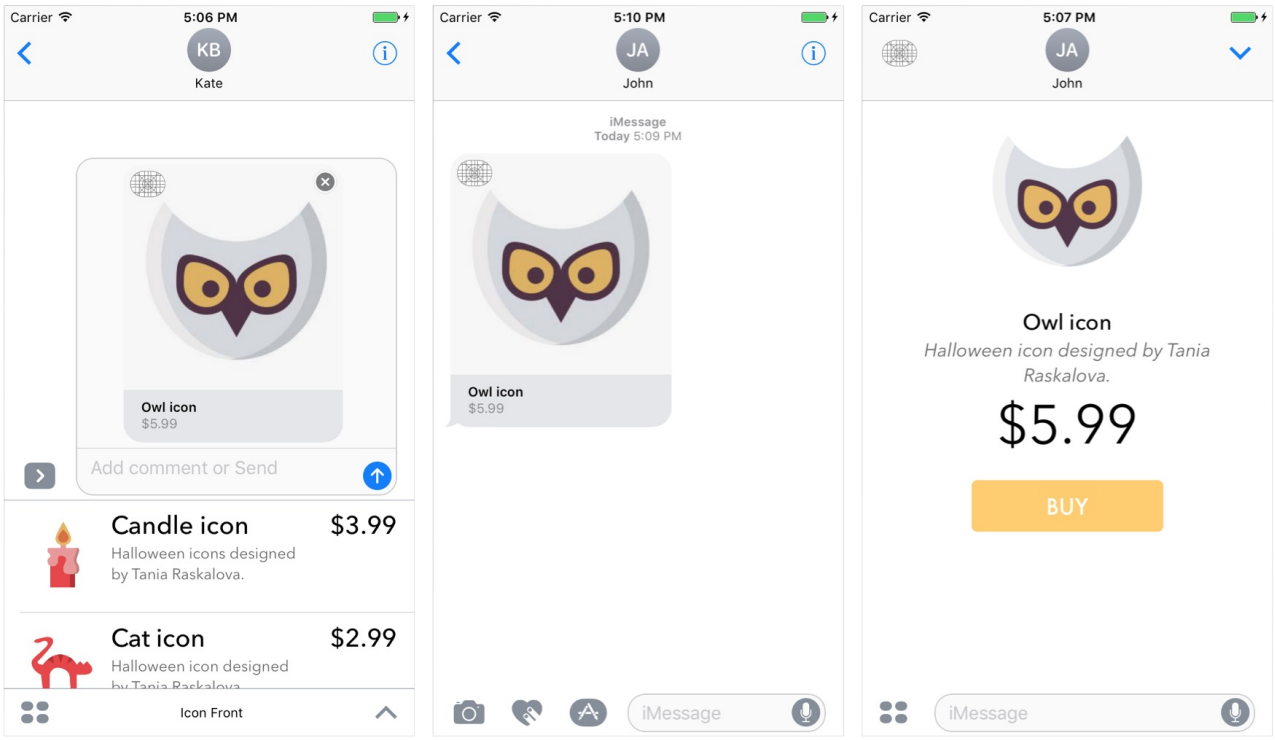


Figure 35.4. How the iMessage app of Icon Store works

Creating the Message Extension

Okay, let's get started. To create a message extension, you have to add a new target for the existing project. First, select *CollectionViewDemo* project file in the project navigator, and then go up to the Xcode menu. Select *Editor* > *Add Target*....

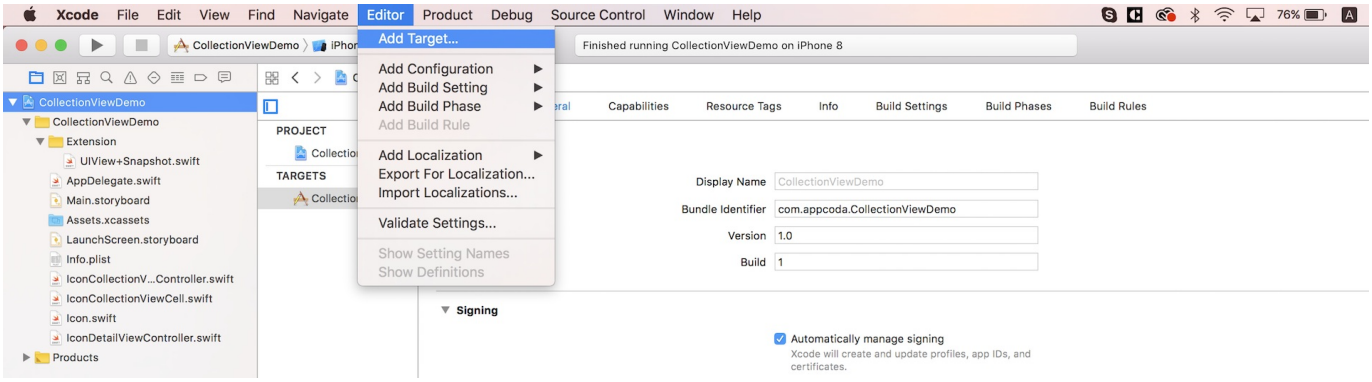


Figure 35.5. Adding a new target for the existing project

Next, choose *iMessage Extension* and confirm. Name the product *IconStore* and hit *Finish* to proceed.

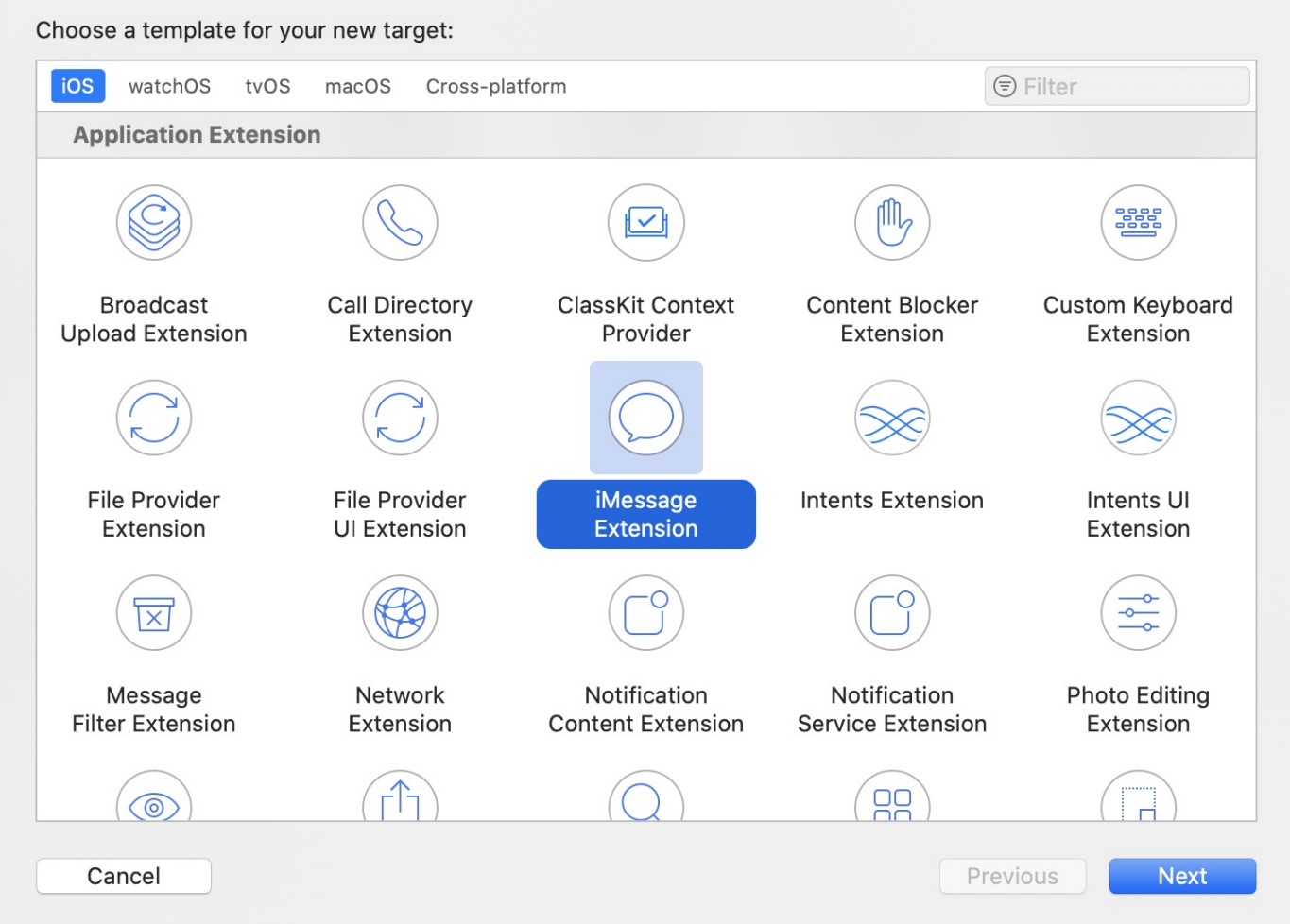


Figure 35.6. Choose iMessage Application to create a message extension

You will be prompted to activate the *MessagesExtension* scheme. Simply hit *Activate* to use the scheme for later testing and debugging.

Once Xcode created the message extension files, you will see a new folder in the project navigator:

- **IconStore** - contains the asset catalog for the message extension, plus the `.swift` files and storyboard for the message extension. This is where you place the app icon of the iMessage app.

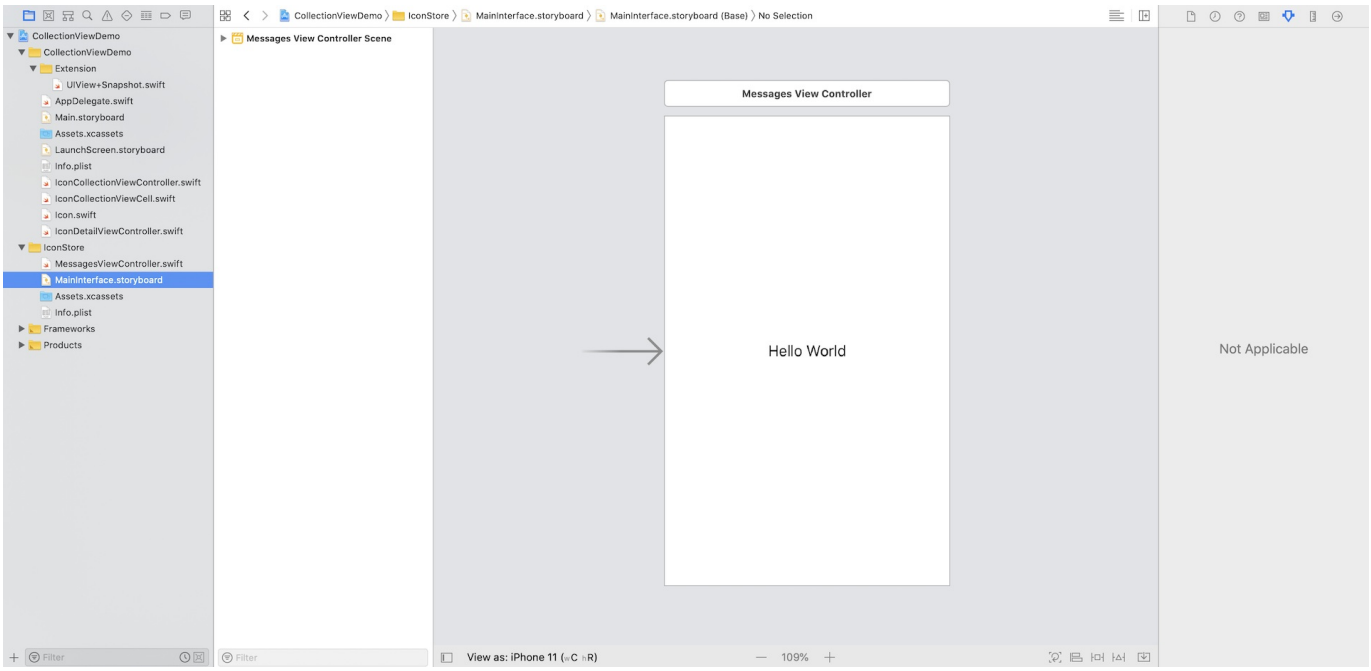


Figure 35.7. Choose iMessage Application to create a message extension

Now, if you select the *IconStore* scheme and hit the Run button, Messages will bring up the *IconStore* app and displays the Hello World label. In case if the *IconStore* extension is brought up, you can click the app ico in the app drawer to activate it.

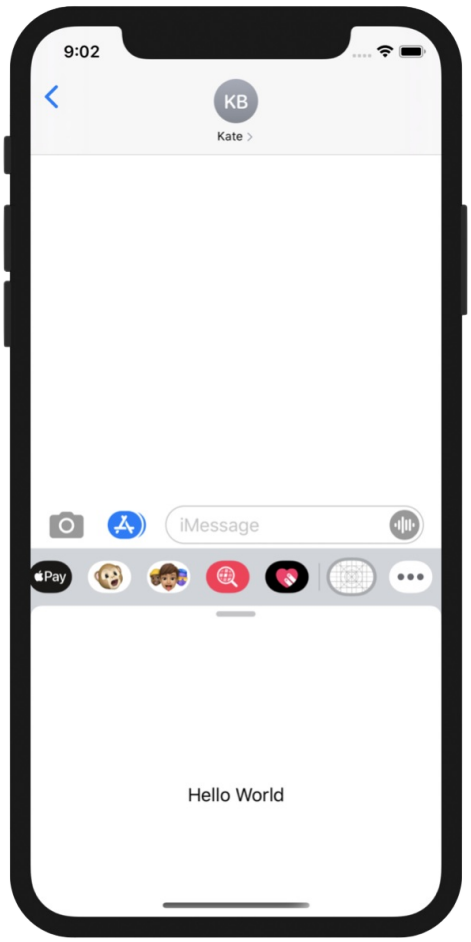


Figure 35.8. Running the message extension

As you may have already realized, developing a message extension (or iMessage app) is very similar to developing an iOS app. It has its own storyboard, asset catalog, and `.swift` files. Therefore, to build the iMessage app, we will design a new UI using the new storyboard and provide the implementation of *MessagesExtension*.

Sharing Code Using an Embedded Framework

Before we implement the messages extension, let's first check out the existing code. Open the `IconCollectionViewController.swift` file. You should see a variable named `iconSet`, which is an array of `Icon` objects. It stores all icon items for display in the app.

```
private var iconSet: [Icon] = [ Icon(name: "Candle icon", imageName: "candle", des
cription: "Halloween icons designed by Tania Raskalova.", price: 3.99, isFeatured:
false),
                                Icon(name: "Cat icon", imageName: "cat", descripti
on: "Halloween icon designed by Tania Raskalova.", price: 2.99, isFeatured: true),
                                Icon(name: "dribbble", imageName: "dribbble",


...

                            ]
```

In the app extension, we also need the icon data to display the icons in the Messages browser. Obviously, you can copy and paste the data set (i.e. `iconSet`) into a new file of the app extension. But this is not a good practice. We should avoid duplicating code.

Instead, as the code is shared between the iOS app and the iMessage app, we will create a framework that embeds the shared code. Here is what we are going to do:

1. Create an embedded framework called `IconDataKit`. This framework can be used by the CollectionViewDemo app and the IconStore iMessage app.
2. In the framework, we will create a file named `IconData.swift`. We will define the `iconSet` array in the file, and initialize it with the icon items.
3. Since the `Icon` class is also used by both apps, we will also move the `Icon.swift` file to the framework.
4. Remove the original `iconSet` variable from `IconCollectionViewController` and use the new `iconSet` provided by the framework

Create the IconDataKit framework

To create a framework, select *CollectionViewDemo* in the project navigator. Then go up to the Xcode menu, select *Editor* > *Add Target*.... Under iOS, choose *Framework* and hit `Next` to proceed.

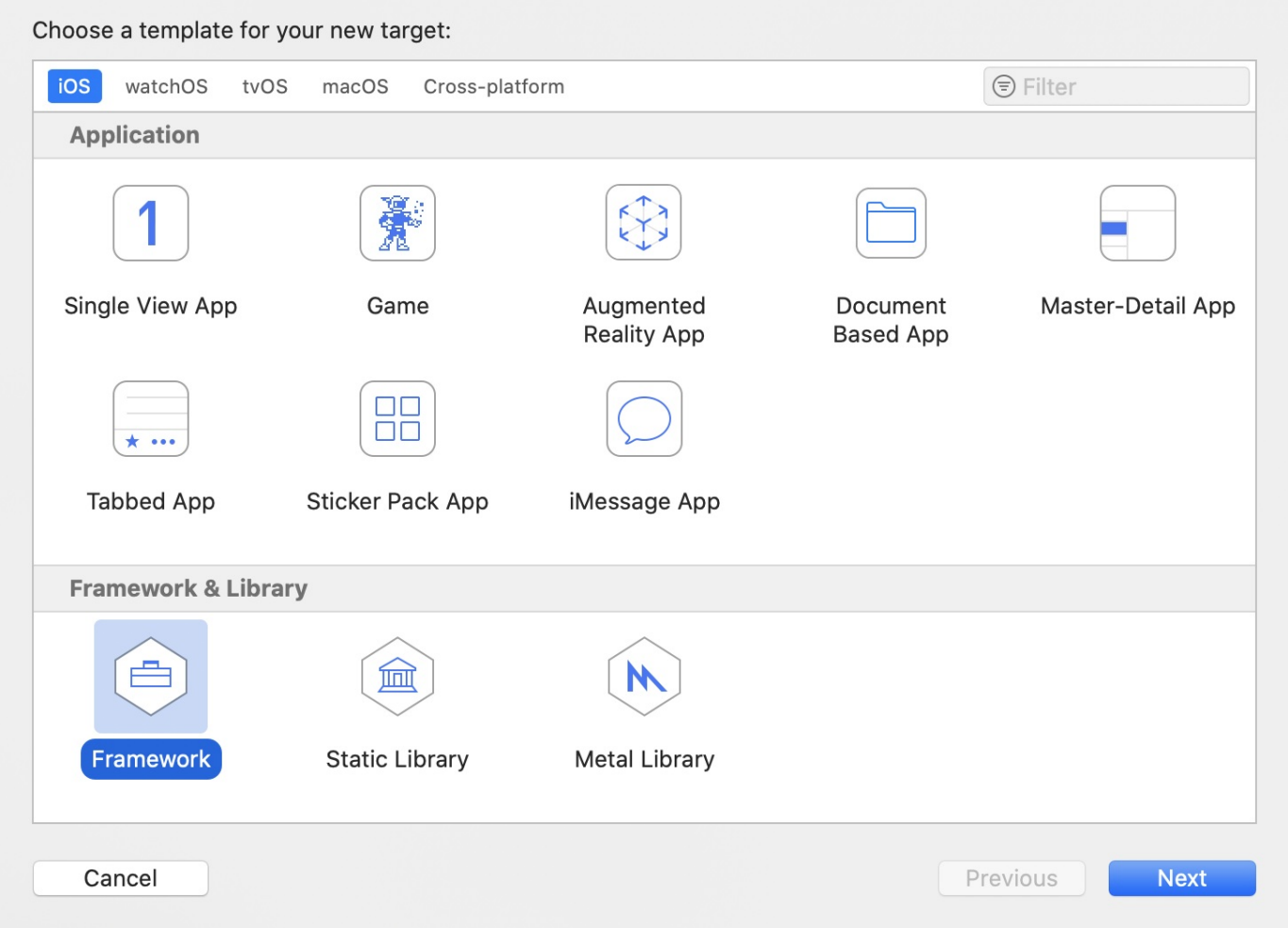


Figure 34.9. Choosing the Cocoa Touch Framework template

In the next screen, name the product `IconDataKit` and confirm. Xcode will create a new folder called `IconDataKit`.

Before writing code for the framework, select *CollectionViewDemo* in the project navigator and choose the *IconDataKit* target. Under the Deployment Info section, enable *Allow app extension API only*. We have to turn on this option as the framework is going to be used by an app extension. Optionally, you can set the target iOS version (e.g. 11.0 or up).

Note: When creating a framework to share code between the containing app (i.e. iOS app) and the message extension, you have to make sure that the embedded framework does not contain APIs unavailable to app extensions. Otherwise, when you submit your app to the App Store, their review team will reject your app extension.

Defining the `iconSet` array

Now, right-click the `IconDataKit` folder in the project navigator and select `New File…`. Choose the *Swift file* template to create a simple `.swift` file, and name it `IconData`.

Open the `IconData.swift` file once it is created, and update its content like this:

```
import Foundation

public struct IconData {
    public static var iconSet: [Icon] = [ Icon(name: "Candle icon", imageName: "ca
ndle", description: "Halloween icons designed by Tania Raskalova.", price: 3.99, i
sFeatured: false),
                                          Icon(name: "Cat icon", imageName: "cat",
 description: "Halloween icon designed by Tania Raskalova.", price: 2.99, isFeatur
ed: true),
                                          Icon(name: "dribbble", imageName: "dribb
ble", description: "Halloween icon designed by Tania Raskalova.", price: 1.99, isF
eatured: false),
                                          Icon(name: "Ghost icon", imageName: "gho
st", description: "Halloween icon designed by Tania Raskalova.", price: 4.99, isFe
atured: false),
                                          Icon(name: "Hat icon", imageName: "hat",
 description: "Halloween icon designed by Tania Raskalova.", price: 2.99, isFeatur
ed: false),
                                          Icon(name: "Owl icon", imageName: "owl",
 description: "Halloween icon designed by Tania Raskalova.", price: 5.99, isFeatur
ed: true),
                                          Icon(name: "Pot icon", imageName: "pot",
 description: "Halloween icon designed by Tania Raskalova.", price: 1.99, isFeatur
ed: false),
```

```
                                        Icon(name: "Pumkin icon", imageName: "pu
mkin", description: "Halloween icon designed by Tania Raskalova.", price: 0.99, is
Featured: false),
                                        Icon(name: "RIP icon", imageName: "rip",
 description: "Halloween icon designed by Tania Raskalova.", price: 7.99, isFeatur
ed: false),
                                        Icon(name: "Skull icon", imageName: "sku
ll", description: "Halloween icon designed by Tania Raskalova.", price: 8.99, isFe
atured: false),
                                        Icon(name: "Sky icon", imageName: "sky",
 description: "Halloween icon designed by Tania Raskalova.", price: 0.99, isFeatur
ed: false),
                                        Icon(name: "Toxic icon", imageName: "tox
ic", description: "Halloween icon designed by Tania Raskalova.", price: 2.99, isFe
atured: false),
                                        Icon(name: "Book icon", imageName: "ic_b
ook", description: "Colorful icon designed by Marin Begović.", price: 2.99, isFeat
ured: false),
                                        Icon(name: "Backpack icon", imageName: "
ic_backpack", description: "Colorful icon designed by Marin Begović.", price: 3.99
, isFeatured: false),
                                        Icon(name: "Camera icon", imageName: "ic
_camera", description: "Colorful camera icon designed by Marin Begović.", price: 4
.99, isFeatured: false),
                                        Icon(name: "Coffee icon", imageName: "ic
_coffee", description: "Colorful icon designed by Marin Begović.", price: 3.99, is
Featured: true),
                                        Icon(name: "Glasses icon", imageName: "i
c_glasses", description: "Colorful icon designed by Marin Begović.", price: 3.99,
isFeatured: false),
                                        Icon(name: "Icecream icon", imageName: "
ic_ice_cream", description: "Colorful icon designed by Marin Begović.", price: 4.99
, isFeatured: false),
                                        Icon(name: "Smoking pipe icon", imageNam
e: "ic_smoking_pipe", description: "Colorful icon designed by Marin Begović.", pri
ce: 6.99, isFeatured: false),
                                        Icon(name: "Vespa icon", imageName: "ic_
vespa", description: "Colorful icon designed by Marin Begović.", price: 9.99, isFe
atured: false)]
}
```

Both the `IconData` structure and the `iconSet` variable are defined with the access level `public`, so that other modules can access them. If you want to learn more about access levels in Swift, you can check out the official documentation (https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AccessControl.html).

Moving the Icon Class from CollectionViewDemo to the Framework

Next, we will migrate the Icon class (i.e. `Icon.swift`) from *CollectionViewDemo* to the *IconDataKit* framework. To do that, you just need to drag the `Icon.swift` file under *CollectionViewDemo* to *IconDataKit*. But make sure you change the target membership from *CollectionViewDemo* to *IconDataKit*. Otherwise, you will experience an error when building the framework.

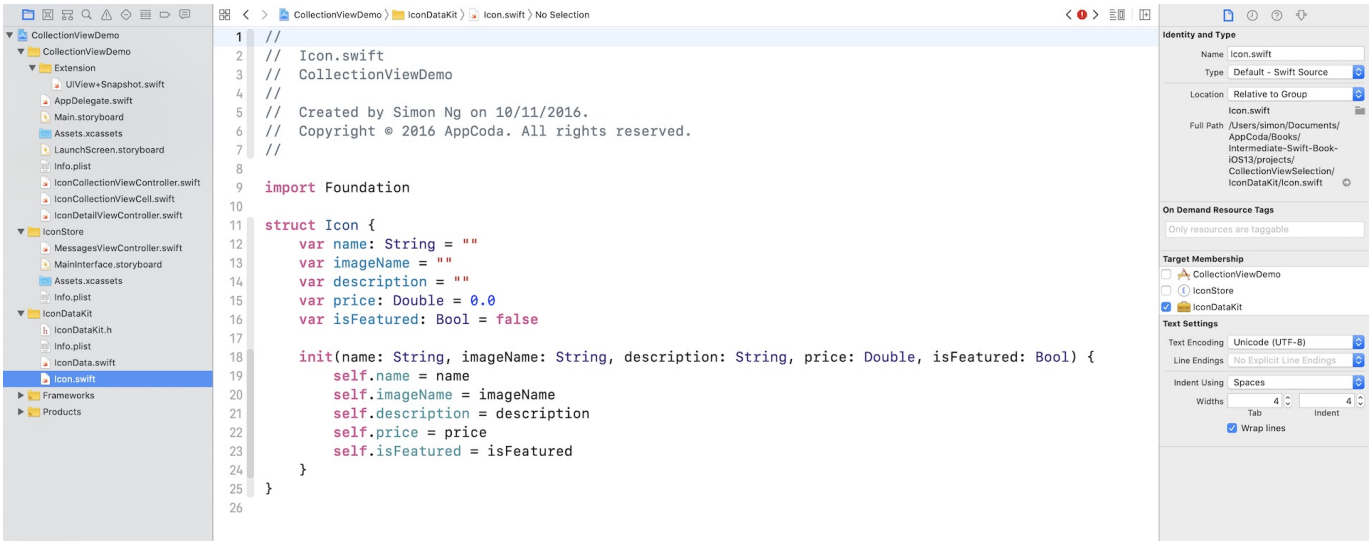


Figure 34.10. Migrating the Icon.swift to the IconDataKit framework

Similar to what we did earlier, we need to modify the code a bit to change the access level to *public* like this:

```
public struct Icon {
    public var name: String = ""
    public var imageName = ""
    public var description = ""
    public var price: Double = 0.0
    public var isFeatured: Bool = false

    public init(name: String, imageName: String, description: String, price: Double
, isFeatured: Bool) {
        self.name = name
        self.imageName = imageName
        self.description = description
        self.price = price
        self.isFeatured = isFeatured
    }
}
```

Replacing the Value of the iconSet Variable

Now that the icon data is migrated to the `IconDataKit` framework, it is time to replace the original `iconSet` variable in the `IconCollectionViewController` class with the one in `IconData`.

Open `IconCollectionViewController.swift` and add the following import statement:

```
import IconDataKit
```

Replace the `iconSet` variable like this:

```
private var iconSet: [Icon] = IconData.iconSet
```

Now we refer to the set of icon data defined in IconDataKit. Lastly, open `IconDetailViewController.swift`, which also refers to the `Icon` class. Insert the following statement at the very beginning to import the `IconDataKit` framework:

```
import IconDataKit
```

That's it. We have now migrated the common data to a framework. However, if you run the app (i.e. CollectionViewDemo) on an iOS 13 device or simulator, you will receive the following error, indicating that the module `IconDataKit` is not compatible with our app.

```
Compiling for iOS 11.0, but module 'IconDataKit' has a minimum deployment target o
f iOS 13.2: /Users/simon/Library/Developer/Xcode/DerivedData/Collectio...
```

The deployment version of the `CollectionViewDemo` target is set to 11.0. In order to use the `IconDataKit` module, go to the `IconDataKit` target and change the target version in the Deployment Info section from `13.2` to `11.0`.

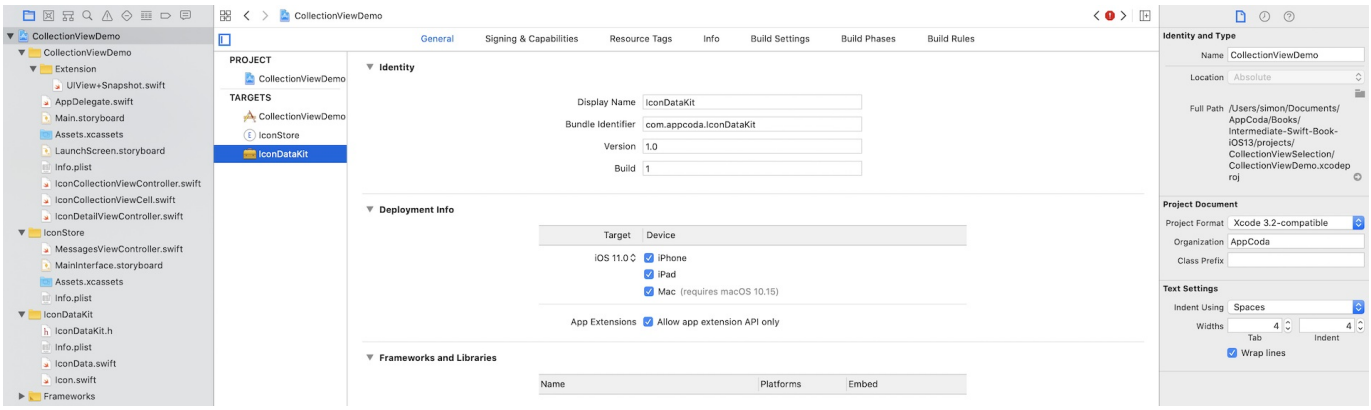


Figure 35.11. Update the deployement target of IconDataKit

Now run the app again, it should work. The app should look the same as it is. However, the underlying implementation of the icon data is totally different. And, the framework is ready to be used by both *CollectionViewDemo* and *IconStore*.

Designing the UI of the iMessage App

Okay, it's time to move onto the implementation of the iMessage app. We will begin with the user interface. If you forget the look & feel of the iMessage app we are going to build, refer to figure 35.4. The iMessage app displays a list of icons with description and price in a table view. When the user taps any of the icons, it will bring up a modal view controller to display the icon details.

Now open `MainInterface.storyboard` under *IconStore*. Let's see how to design the iMessage app UI.

For any iMessage app, `MSMessagesAppViewController` is the principal view controller. This is the view controller it is presented to users when the iMessage app is launched. The storyboard already comes with a default view controller, which is a child class of `MSMessagesAppViewController`. We're going to design this controller and turn it into a table view for displaying a list of icons.

First, delete the default "Hello World" label. Then drag a table view object from the Object library to the view controller. Resize it to fit the whole view. In the Attributes inspector, change the *Prototype Cells* option from `0` to `1` to add a prototype cell. Next, change the height of the cell to `103` points. Make sure you select the table view cell and go to the Attributes inspector. Set the cell's identifier to `Cell`.

To ensure the table view fits all screen sizes, select the table view and click the `Add New Constraints` button. Set the spacing values to zero and add the constraints.

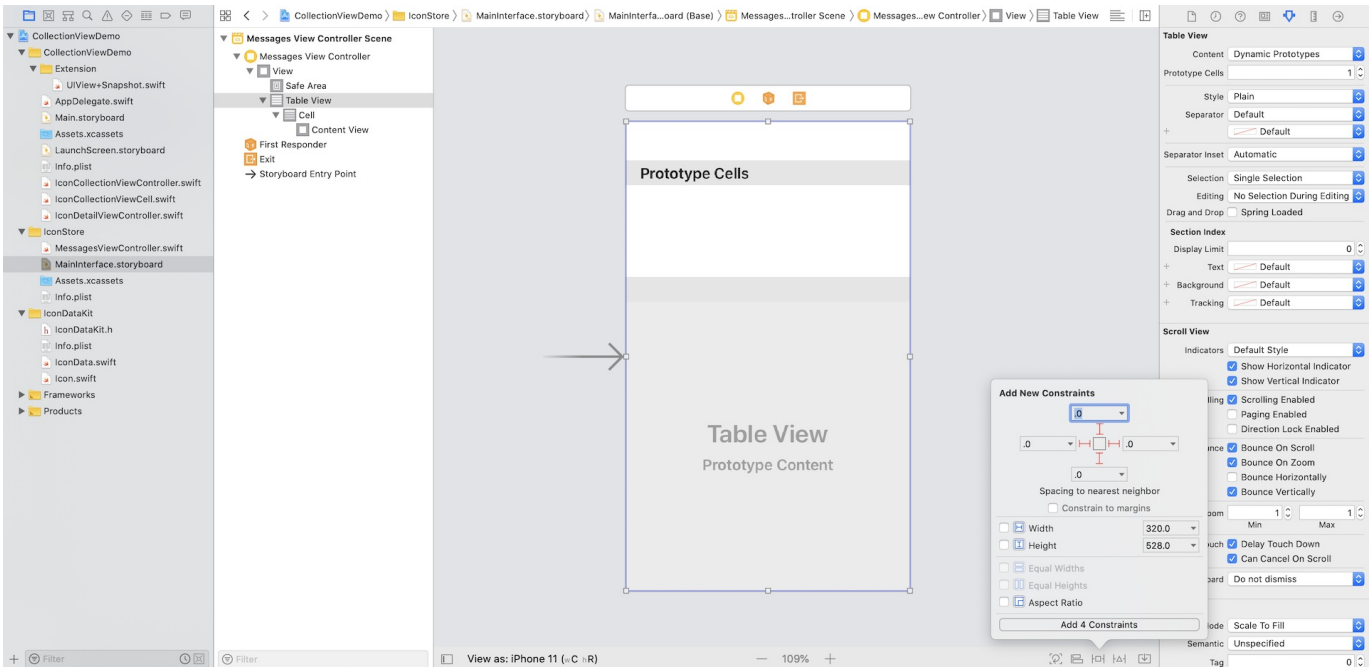


Figure 35.12. Designing the UI of the Messages View Controller

So far, your UI should look like figure 35.12. Now we are going to design the prototype cell.

First, drag an image view to the cell. Change its size to 72 points by 86 points. In the Attributes inspector, set the content mode to `Aspect Fit`.

Next, add a label to the cell and change the title to `Name`. Set the font size to `23` points and type to `Avenir Next`.

Drag another label to the cell and set the title to `Description`. Change the font color to `Dark Gray`, and set the font size to `14` points.

Now drag another label to the cell and set the title to `Price`. Change the font size to `23` points and set the font to `Avenir Next`. Also, set the Alignment option to right-aligned.

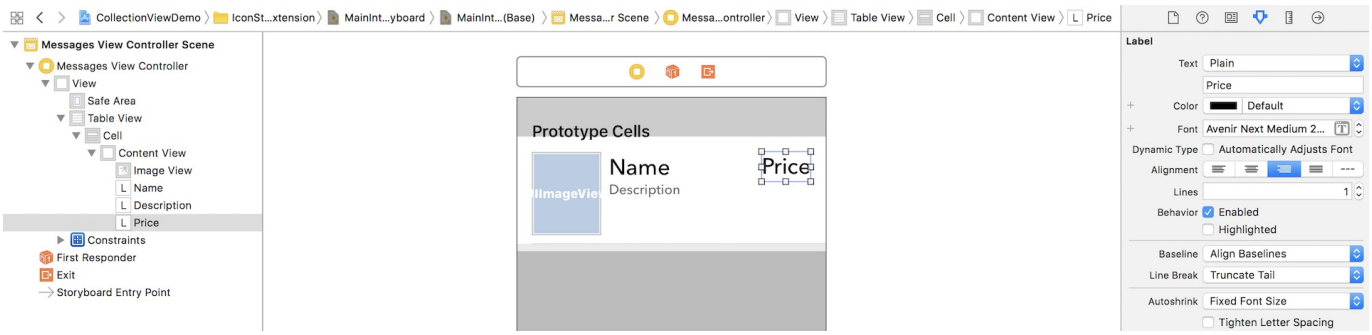


Figure 35.13. Designing the UI of the Messages View Controller

Once finished, your cell UI should be similar to that shown in figure 35.13.

In order to ensure the UI elements fit all types of screens, we will use stack views and add some layout constraints for the labels and image view.

First, hold the command key, and select both *Name* and *Description* labels. Click the `Embed in` button to embed them in a stack view. Then select both the stack view and the *Price* label. Again, click the `Embed in` button to embed both items in a stack view. In the Attributes inspector, set the spacing option to `20` points.

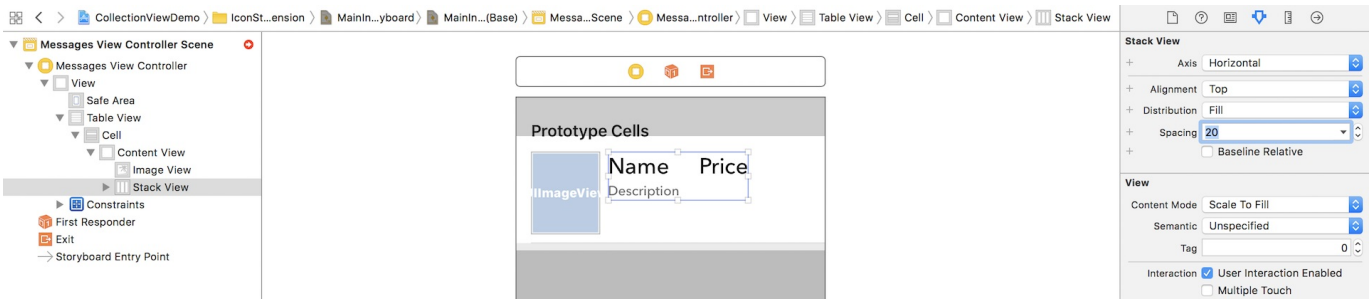


Figure 35.14. Embedding labels in stack views

Next, select the stack view we just created and the image view. Click the `Embed in` button to embed both UI elements in a stack view. In the Attributes inspector, set the spacing to `10` points.

Once again, select the stack view, that embeds all the labels and image view. Click the `Add New Constraints` button and add 4 spacing constraints for the stack view. Refer to figure 35.15 for the spacing values.

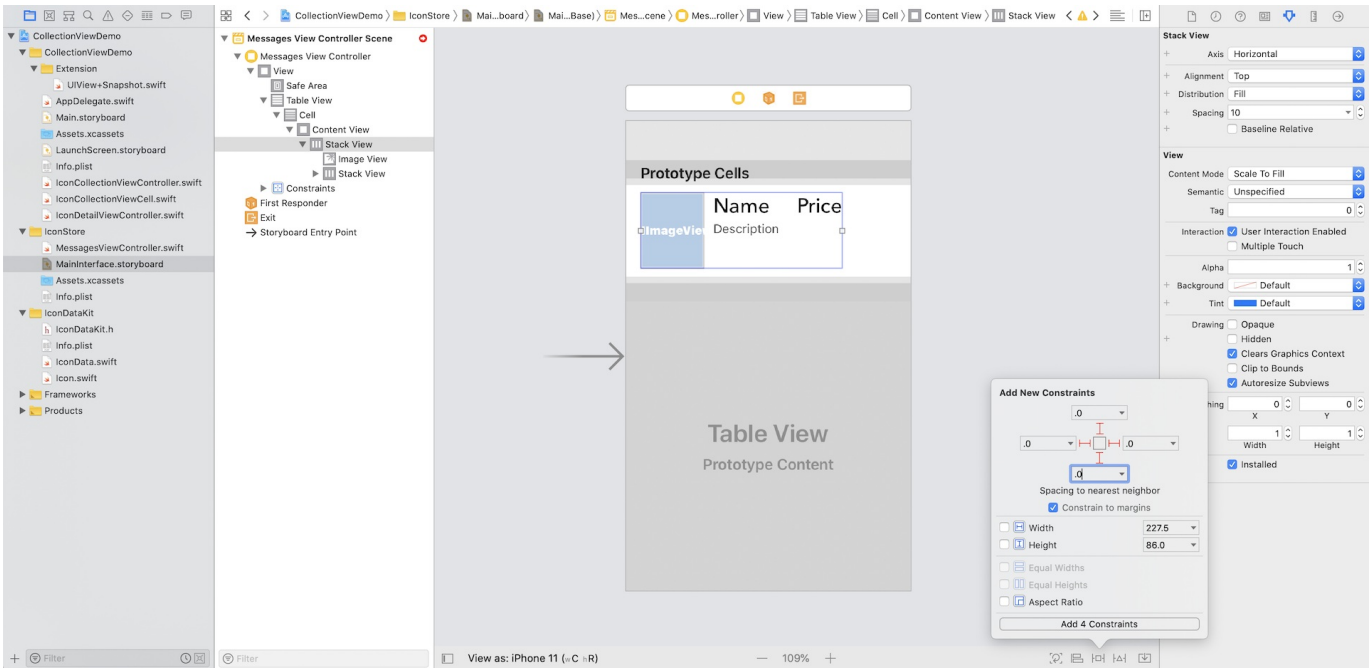


Figure 35.15. Embedding labels in stack views

If you experience any layout issues, just hit the `Update Frames` in the layout bar to fix the issues.

Lastly, we want to fix the size of the image view. Select the image view, and then click the `Add New Constraints` button. Check both *width* and *height* checkboxes, and add the constraints.

Now that you've completed the design of the Messages View Controller, let's move onto the coding part.

Implementing MessagesViewController

The Messages View Controller in the storyboard is associated with the `MessagesViewController.swift`. In order to display the icons in the table, we have to implement two things:

- Create a new class for the custom table view cell
- Update the `MessagesViewController` class to implement both `UITableViewDataSource` and `UITableViewDelegate` protocols

> **Note:** I assume that you understand how to work with UITableView and populate data in it. If not, please refer to the beginner book for details.

So first, right click the *IconStore* folder in the project navigator. Choose `New File…` and select Cocoa Touch Class. Name the class `IconTableViewCell`.

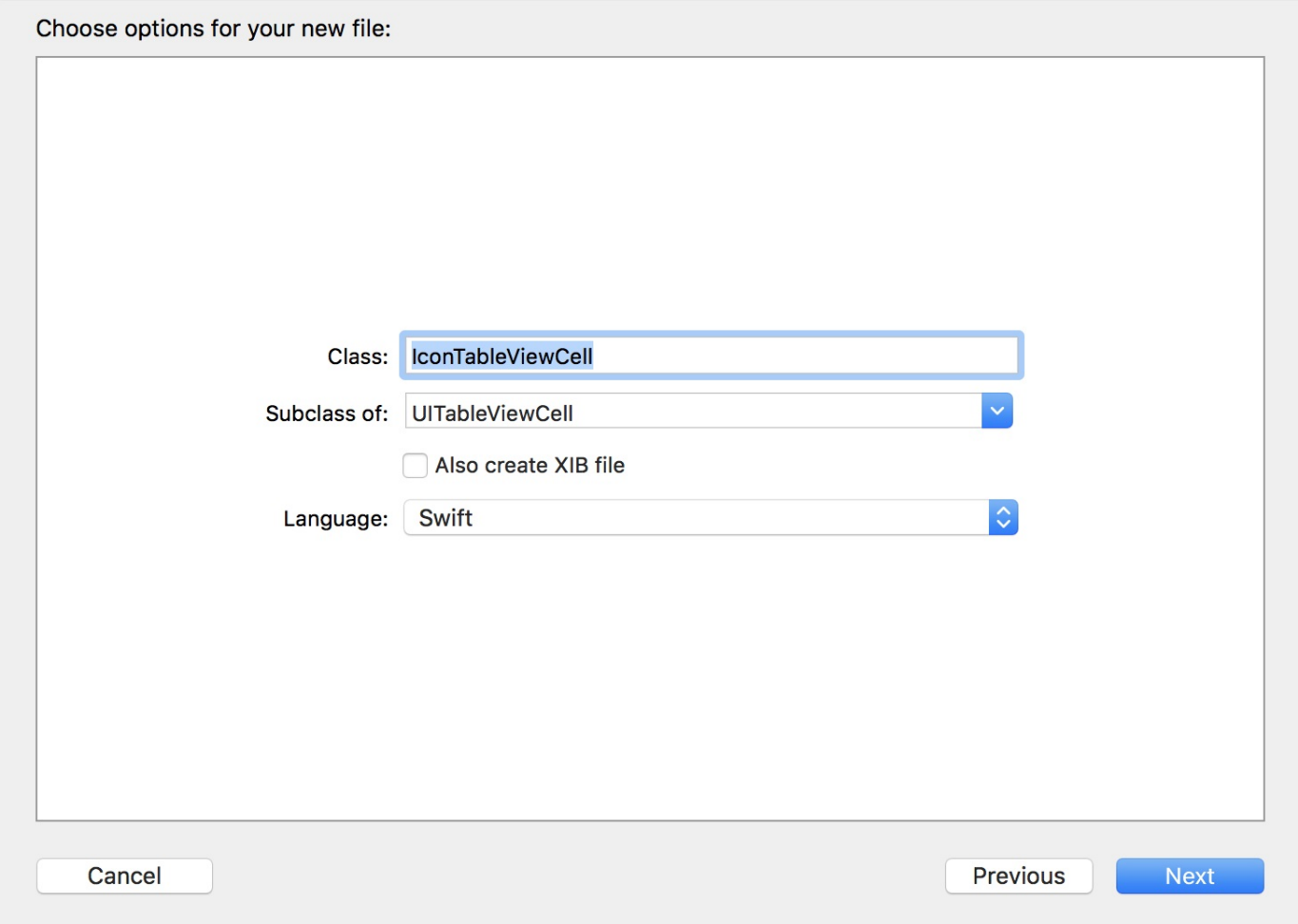


Figure 35.16. Embedding labels in stack views

Once the file is created, add the following outlet variables in the `IconTableViewCell` class:

```
@IBOutlet var iconImageView: UIImageView!
@IBOutlet var nameLabel: UILabel!
@IBOutlet var descriptionLabel: UILabel! {
    didSet {
        descriptionLabel.numberOfLines = 0
    }
}
@IBOutlet var priceLabel: UILabel!
```

Now go back to `MainInterface.storyboard`, and select the prototype cell. In the Identity inspector, set the custom class to `IconTableViewCell`. Then connect the labels/image view with the corresponding outlet variable.

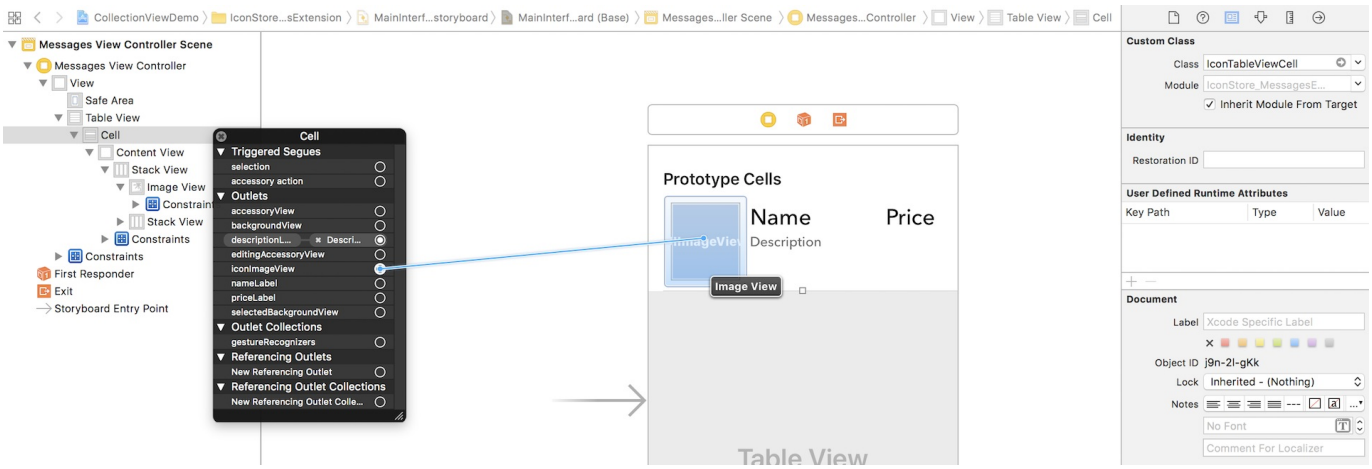


Figure 35.17. Establish a connection between the outlet variables and the labels/images view

The next step is to implement both `UITableViewDataSource` and `UITableViewDelegate` protocols.

First, in the `MessagesViewController.swift` file, add an import statement to import the `IconDataKit` framework. We need to do this because we are going to load the icon data from the framework.

```
import IconDataKit
```

Next, define an outlet variable for the table view in the class:

```
@IBOutlet var tableView: UITableView!
```

You need to switch back to `MainInterface.storyboard` to connect the table view with the `tableView` outlet.

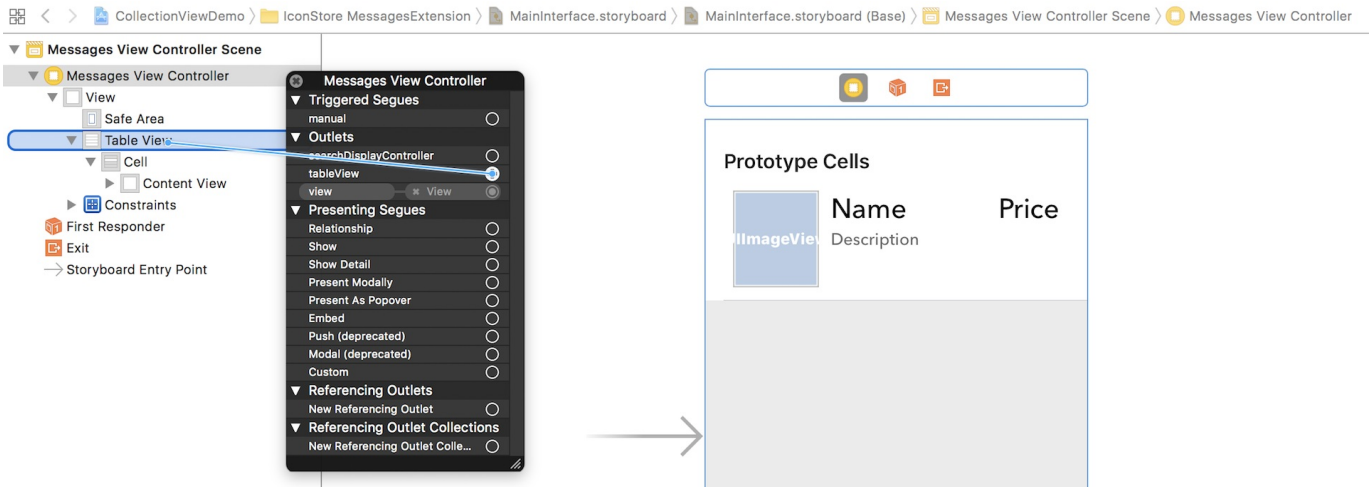


Figure 35.18. Establish a connection between the outlet variable and the table view

Similar to `IconCollectionViewController`, we need to define a variable to hold the icon set (or the icon data). In `MessagesViewController`, define a new variable named `iconSet` to store the icon data from `IconData`:

```
private var iconSet = IconData.iconSet
```

To populate the icon data in the table view, we will implement three methods as required by the `UITableViewDataSource` protocol. Create an extension to implement the protocol:

```
extension MessagesViewController: UITableViewDataSource {

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

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

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

        let icon = iconSet[indexPath.row]
        cell.nameLabel.text = icon.name
        cell.descriptionLabel.text = icon.description
        cell.priceLabel.text = "$\(icon.price)"
        cell.iconImageView.image = UIImage(named: icon.imageName)

        return cell
    }

}
```

Lastly, update the `viewDidLoad()` method to set the data source of the table view:

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

    tableView.dataSource = self
}
```

Now it is ready to test the iMessage app and see if it works. Make sure you select the *MessagesExtension* scheme and choose whatever iOS simulator (e.g. iPhone 11) as you like. Hit the Run button and load the message extension in the Messages app.

If everything works as expected, your iMessage app should display a list of icons in Messages. But you will notice that all icon images are missing.

Currently, the icon images are put in the asset catalog of the *CollectionViewDemo* app. If you select the asset catalog, you should find that its target membership is set to *CollectionViewDemo*. To allow the message extension to access the asset, check *IconStore* under target membership.

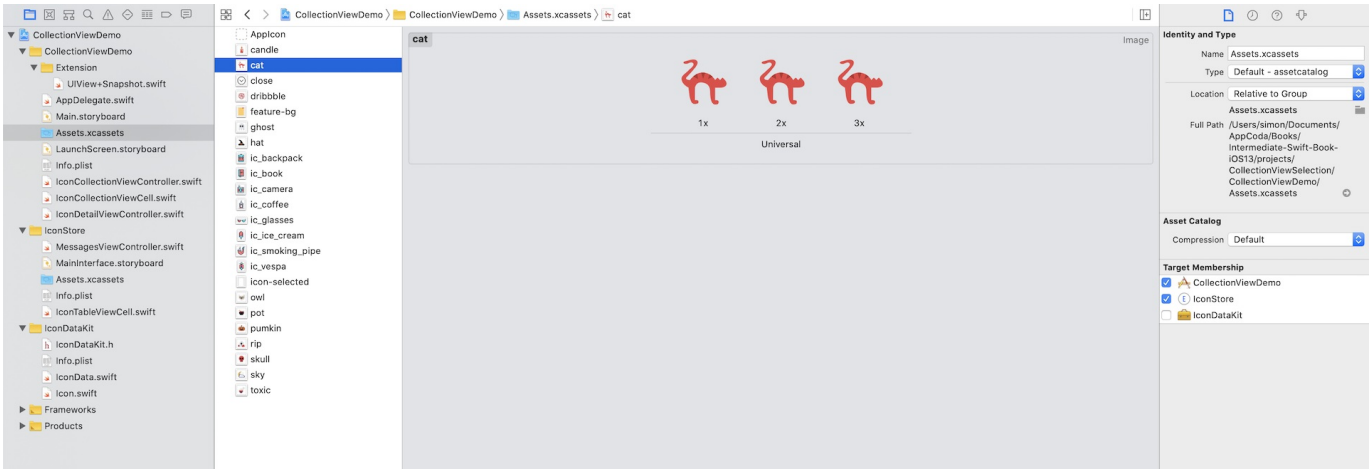


Figure 35.19. Enable IconStore MessagesExtension in Target Membership

Run the iMessage app again. It should be able to load the icon images. You can click the expand button at the lower right corner to expand the view to reveal more icons.

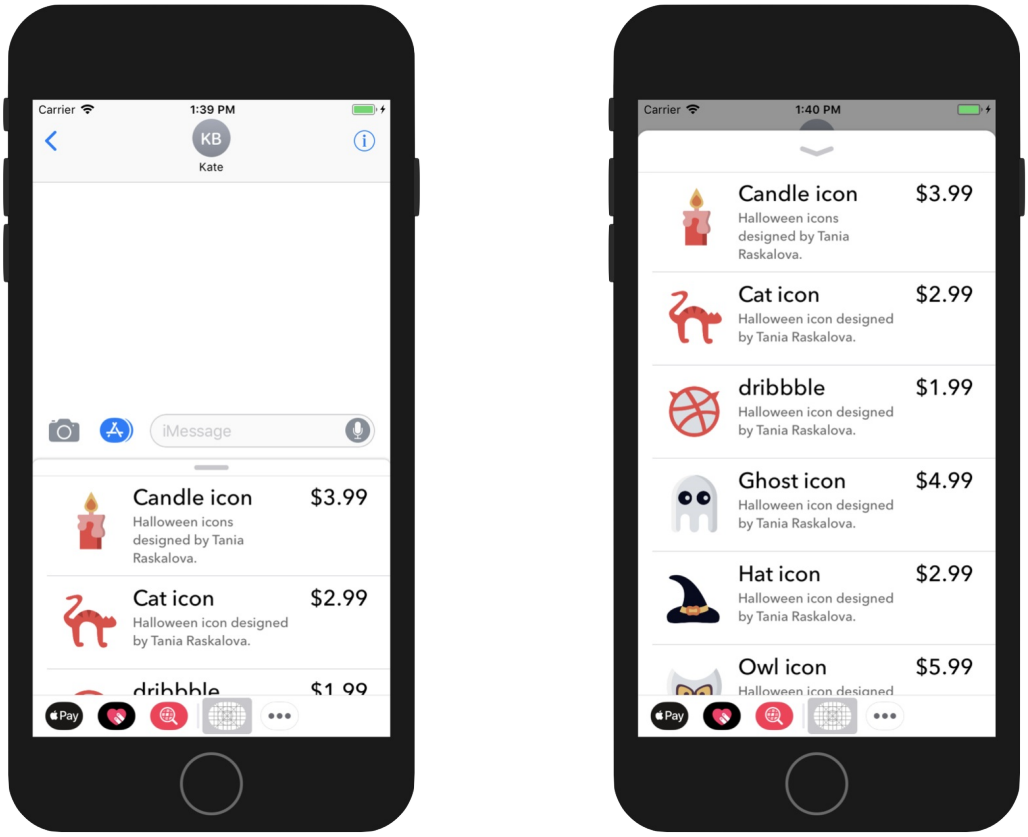


Figure 35.20. iMessage app in compact mode (left) and expanded mode (right)

Adding a Message to the Conversation

Did you tap an icon and expect it to show up in the message field? Meanwhile, it won't. Unlike the sticker app we built in the previous chapter, you have to handle the item selection on your own for a custom iMessage app.

If you understand the `UITableViewDelegate` protocol, you should be very familiar with the implementation of table view selection. It comes down to this method of the protocol:

```
optional func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexP
ath)
```

In the `MessagesViewController` class, insert a line of code to declare the `selectedIcon` variable:

```
private var selectedIcon: Icon?
```

This variable is used to hold the selected icon for later use. For the `tableView(_:didSelectRowAt:)` method, we will implement it using an extension:

```
extension MessagesViewController: UITableViewDelegate {

    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath)
{

        requestPresentationStyle(.compact)
        tableView.deselectRow(at: indexPath, animated: true)

        let icon = iconSet[indexPath.row]

        if let conversation = activeConversation {
            let messageLayout = MSMessageTemplateLayout()
            messageLayout.caption = icon.name
            messageLayout.subcaption = "$\(icon.price)"
            messageLayout.image = UIImage(named: icon.imageName)

            let message = MSMessage()
            message.layout = messageLayout

            if var components = URLComponents(string: "http://www.appcoda.com") {
                components.queryItems = icon.queryItems
                message.url = components.url
            }

            conversation.insert(message, completionHandler: { (error) in
                if let error = error {
                    print(error)
                }
            })
        }
    }
}
```

Let me walk you through the code line by line.

As you know, iMessage apps can be in two states: *compact* and *expanded*. The message field only appears when the app is in compact mode. Therefore, the first line of code (`requestPresentationStyle(.compact)`) ensures the iMessage app returns to compact mode.

The second line is pretty trivial. We simply call the `deselectRow` method of the table view to deselect the row.

The next two lines are to retrieve the current selected icon.

The rest of the code is the core of the method. `MSMessagesAppViewController` has a property called `actionConversation` , which holds the conversation that the user is currently viewing in the Messages app. To add a message to the existing conversion, you will need to implement a couple of things:

1. Create an `MSMessage` object - it is the object that will be inserted in the conversation. To create an `MSMessage` object, you are required to set both its `url` and `layout` properties. The `URL` property is the model of the message. In other words, it contains the message's data in form of URL. Here is an example:

   ```
   http://www.appcoda.com/?name=Cat%20icon&imageName=cat&description=Halloween%20icon
   %20designed%20by%20Tania%20Raskalova.&price=2.99
   ```

   The information of the cat icon is encoded into a URL string. Each property of an icon is converted into a URL parameter. At the receiving end, it can pick up the URL and easily get back the message content by parsing the URL parameters.

   Not only is the URL designed for data passing, it is intended to link to a particular web page to display the custom message content for devices who do not support the messaging extension. Say, you view a message sent from an iMessage app using the built-in Messages app on macOS. You will be redirected to the URL and use Safari to view the message content.

   The `layout` property defines the look & feel of the message. The Messages framework comes with an API called `MSMessageTemplateLayout` that lets developers easily create a message bubble. The message template includes the Message

extension's icon, an image (video/audio file) and a number of text elements such as title and subtitle. Figure 35.21 shows the message template layout.

Figure 35.21. Message Template Layout

2. Once the `MSMessage` object is created, you can add it to the active conversation, which is an instance of `MSConversation`. To do that, you can call its `insert(_:completionHandler:)` method like this:

```
conversation.insert(message, completionHandler: { (error) in
    if let error = error {
        print(error)
    }
})
```

Now let's take a look at the code snippet again. To insert a message into the active conversation, we first create the `MSMessageTemplateLayout` object. We set the caption to the icon's name, subcaption to the icon's price, and the image to the icon's image.

And then we create the `MSMessage` object and set its `layout` property to the layout object just created.

As discussed earlier, we have to set the `url` property of the `MSMessage` object to the URL version of the message content.

The question is:

How can we encode and transform the content of the `icon` object into a URL string like this?

```
http://www.appcoda.com/?name=Cat%20icon&imageName=cat&description=Halloween%20icon
%20designed%20by%20Tania%20Raskalova.&price=2.99
```

The iOS SDK has a `URLComponents` structure. You use it to easily access, set, or modify a URL's component parts. In general, you create a `URLComponents` structure with a base URL, and set its `queryItems` property, which is actually an array of `URLQueryItem`. So, you can create the URL string like this:

```
if var components = URLComponents(string: "http://www.appcoda.com") {
    let name = URLQueryItem(name: "name", value: icon.name)
    let imageName = URLQueryItem(name: "imageName", value: icon.imageName)
    let description = URLQueryItem(name: "description", value: icon.description)
    let price = URLQueryItem(name: "price", value: "\(icon.price)")
    components.queryItems = [name, imageName, description, price]
    message.url = components.url
}
```

But in the code snippet, we simplify the code to this:

```
if var components = URLComponents(string: "http://www.appcoda.com") {
    components.queryItems = icon.queryItems
    message.url = components.url
}
```

I want to centralize the encoding and decoding of the message content in the `Icon` class. Therefore I added a couple of extensions in the class. Insert the following code in `Icon.swift`:

```
public extension Icon {

    enum QueryItemKey: String {
        case name = "name"
        case imageName = "imageName"
```

```
        case description = "description"
        case price = "price"
    }

    var queryItems: [URLQueryItem] {
        var items = [URLQueryItem]()

        items.append(URLQueryItem(name: QueryItemKey.name.rawValue, value: name))
        items.append(URLQueryItem(name: QueryItemKey.imageName.rawValue, value: im
ageName))
        items.append(URLQueryItem(name: QueryItemKey.description.rawValue, value:
description))
        items.append(URLQueryItem(name: QueryItemKey.price.rawValue, value: String
(price)))

        return items
    }

    init(queryItems: [URLQueryItem]) {
        for queryItem in queryItems {
            guard let value = queryItem.value else { continue }

            if queryItem.name == QueryItemKey.name.rawValue {
                self.name = value
            }

            if queryItem.name == QueryItemKey.imageName.rawValue {
                self.imageName = value
            }

            if queryItem.name == QueryItemKey.description.rawValue {
                self.description = value
            }

            if queryItem.name == QueryItemKey.price.rawValue {
                self.price = Double(value) ?? 0.0
            }
        }
    }
}

public extension Icon {
```

```
    init?(message: MSMessage?) {
        guard let messageURL = message?.url else { return nil }

        guard let urlComponents = URLComponents(url: messageURL, resolvingAgainstB
aseURL: false),
            let queryItems = urlComponents.queryItems else {
                return nil
        }

        self.init(queryItems: queryItems)
    }
}
```

Also, insert an additional import statement at the very beginning:

```
import Messages
```

In the first extension, we use enum to represent the available URL parameters of the message content. The `queryItems` property is computed on the fly to initialize the `URLQueryItem` pairs.

The extension also provides an init method that accepts an array of `URLQueryItem`, and set its values back to the properties of an `Icon` object.

The second extension is designed for the receiving side, which will be used later. It takes in an `MSMessage` object and converts the content to an `Icon` object.

By using extensions, we add more functionalities to the `Icon` class and centralize all the conversion logic in a common place. It will definitely make the code cleaner and easier to maintain. And this is why we can simply use a single line of code to compute the query items:

```
components.queryItems = icon.queryItems
```

Finally, don't forget to insert this line of code in the `viewDidLoad` method of `MessagesViewController`:

```
tableView.delegate = self
```

That's it. Let's rebuild and test the iMessage app. If Xcode shows you any errors, you will probably need to compile the `IconDataKit` framework again. You will just need to select the `IconDataKit` scheme and hit the Play button to rebuild it. Then you choose the `MessagesExtension` scheme to launch the app in simulator. Now if you pick an icon, it will be displayed in the message field, and send it over to another user.

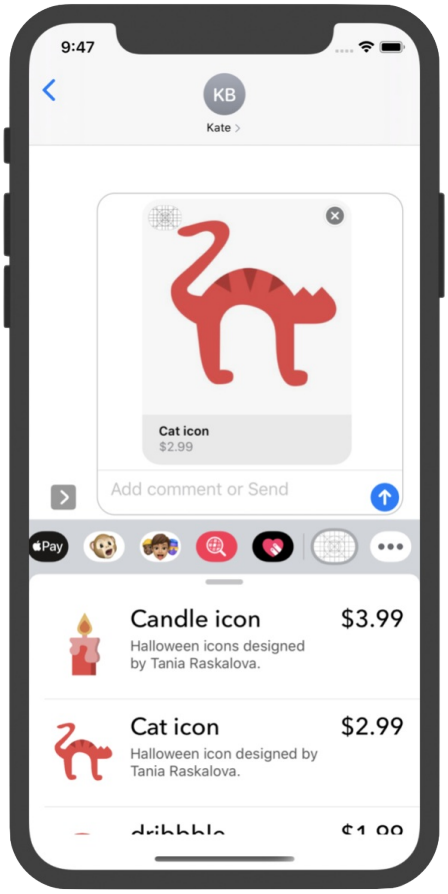


Figure 35.22. Message Template Layout

Displaying the Icon Details

Tapping the message now brings up the iMessage app in expanded mode. This is not we expect. Instead, we want to display the details of the chosen icon. To make this happen, there are a few modifications/enhancements we have to make:

1. Design the detail screen in `MainInterface.storyboard`.
2. Create a new class named `IconDetailViewController` for the detail screen.
3. Modify the `MessagesViewController` class such that it brings up `IconDetailViewController` whenever a message is selected.

Designing the Detail View Controller

Let's begin with the first change and design the detail view controller. Open `MainInterface.storyboard`. In Object library, drag a view controller into the storyboard, and add the following UI elements:

- Drag an image view to the view controller. In the Size inspector, set `X` to `0`, `Y` to `90`, `Width` to `375`, and `Height` to `170`. In the Attributes inspector, set the content mode to `Aspect Fit`.
- Add a *Name* label to the view controller. Choose the font to whatever style you like. I use *Avenir Next* and set the font size to `20` points. Next, add another label named *Description* and put it below the *Name* label. Change its font size to `17` points. Then add another label named *Price*. Make it a bit large than the other two labels (say, set the font size to 50 points). For all the label, change the alignment option to center under the Attributes inspector.
- Lastly, add a button to the view controller and name it `BUY`. In Attributes inspector, set the background color to *yellow* and text color to *white*. Change the width to `175` and height to `47`. To make the button round corner, add a runtime attribute `layer.cornerRadius` in the Identity inspector, and set its value to `5`.

Your detail view controller should be very similar to that shown in figure 35.23.

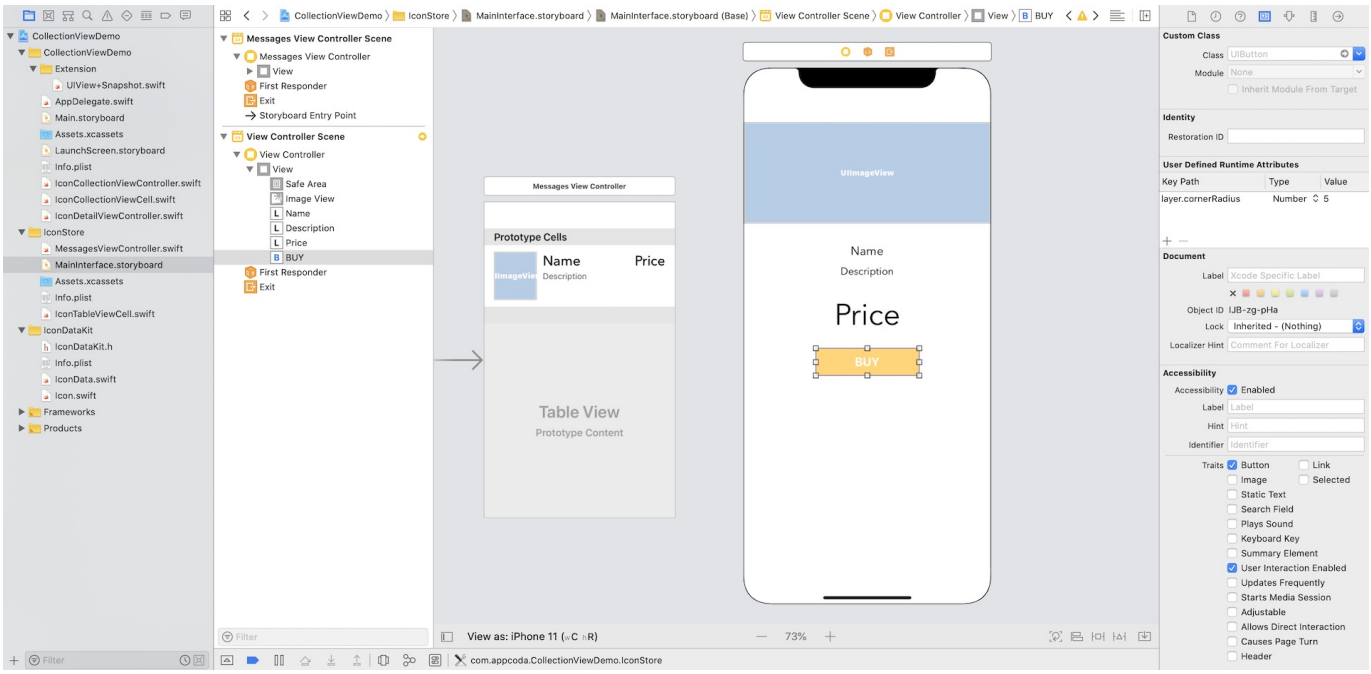


Figure 35.23. The UI design of the detail view controller

As always, we need to add some layout constraints so that the view can fit all devices.

First, select the *Name*, *Description* and *Price* labels. Click the *Embed* button in the layout bar to embed them in a stack view.

Next, select the *Buy* button. Click the `Add New Constraints` button to add a couple of size constraints. Check both *Width* and *Height* option to add two constraints.

Then select the stack view just created and the *Buy* button. Again, click the Embed button to embed them in another stack view. Select the new stack view. In the Attributes inspector, set the *spacing* option to `20` points to add some spaces between the labels and the Buy button. Also, change the alignment option to *Center*.

Once again, select the new stack view and the image view. Click the *Embed* button to embed both views in a new stack view.

Now make sure you select the new stack view, and click the `Add New Constraints` button to add the spacing constraints for the top, left and right sides. You can refer to figure 35.24 for details.

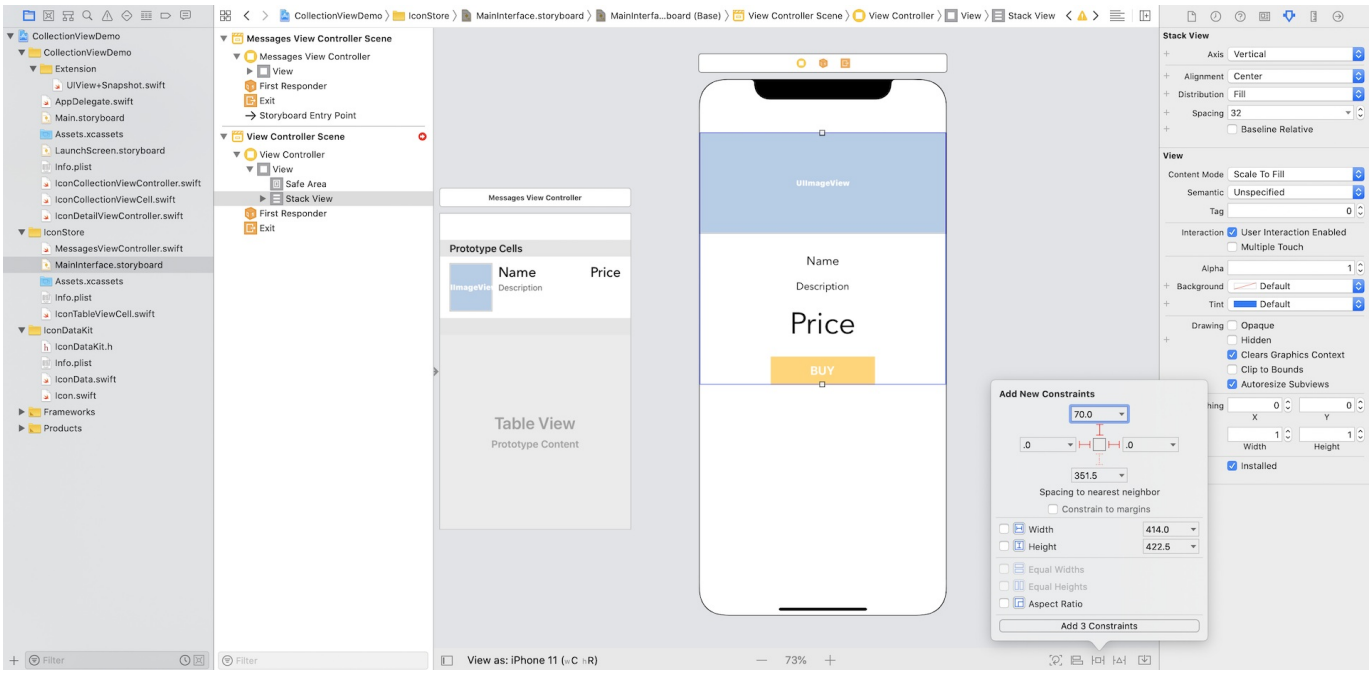


Figure 35.24. The UI design of the detail view controller

Finally, select the image view and add a height constraint to resolve the ambiguity. Click the `Add New Constraints` button and check the height checkbox to add the height constraint.

That's it for the design.

The detail view controller will appear when a user taps one of the table cells. We will connect both view controllers using a segue. Press and hold the *control* key, drag from the *Messages View Controller* to the detail view controller. In the popover menu, choose *Present Modally* as the segue type.

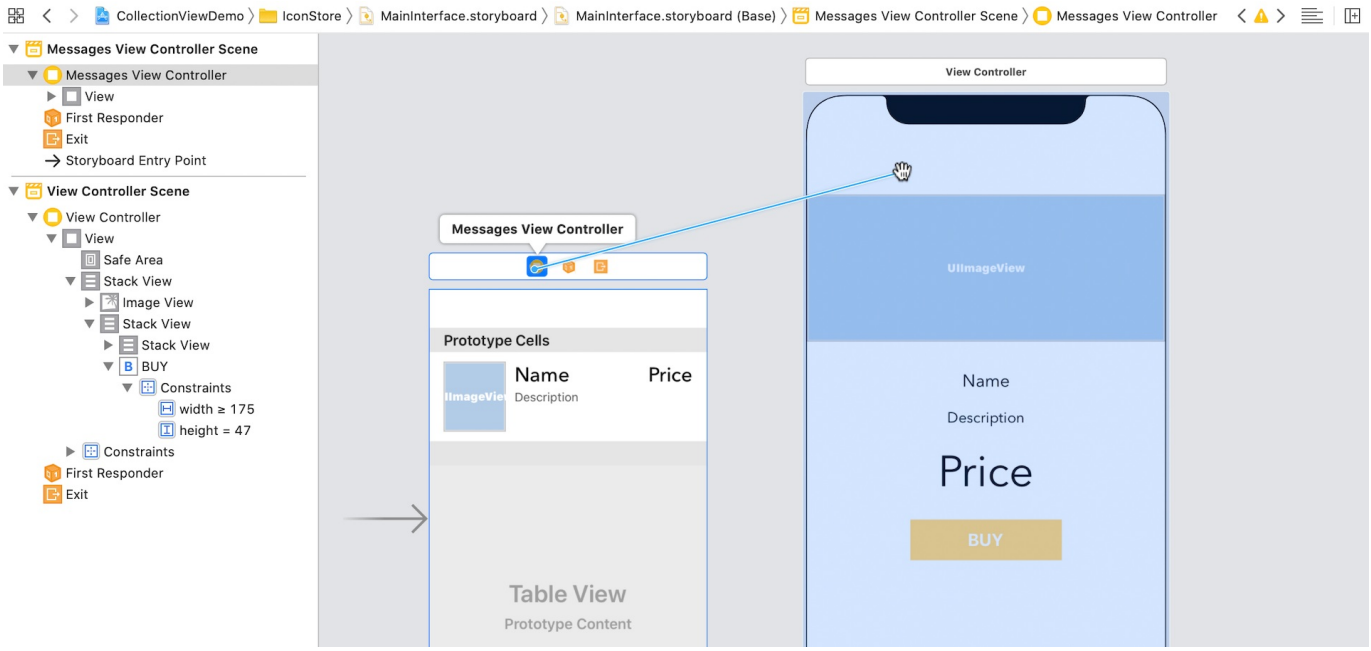


Figure 35.25. Connect both view controllers using a segue

We will need to refer to this segue in our code. So, select the segue and go to the Attributes inspector to give it an identifier. Name the identifier *IconDetail*.

Creating a New Class for the Detail View Controller

Similar to the custom cell, we will create a custom class for the detail view controller. Right click the *IconStore* folder and select `New File…`. Choose the Cocoa Touch Class template and name the class `IconDetailViewController`. Make sure it is extended from `UIViewController`.

In the `IconDetailViewController.swift` file created, insert the following line of code to import the `IconDataKit` framework:

```
import IconDataKit
```

In order to update the content of the UI elements in the detail view controller, declare the following outlets in the class and add an `icon` variable:

```
@IBOutlet var nameLabel: UILabel! {
    didSet {
        nameLabel.text = icon?.name
    }
}
@IBOutlet var descriptionLabel: UILabel! {
    didSet {
        descriptionLabel.text = icon?.description
    }
}
@IBOutlet var iconImageView: UIImageView! {
    didSet {
        iconImageView.image = UIImage(named: icon?.imageName ?? "")
    }
}
@IBOutlet var priceLabel: UILabel! {
    didSet {
        if let icon = icon {
            priceLabel.text = "$\(icon.price)"
        }
    }
}

var icon: Icon?
```

The `icon` variable will store the selected icon (as passed from the Messages View Controller) to display in the detail view. You can initialize the value of the labels and image in the `viewDidLoad()` method. But I prefer to use `didSet` for outlet initialization. It is more readable and keeps the code more organized.

As usual, head back to `MainInterface.storyboard` and set the custom class of the detail view controller to `IconDetailViewController`. And then right click Icon Detail View Controller in document outline and connect the outlets with the appropriate label/image view.

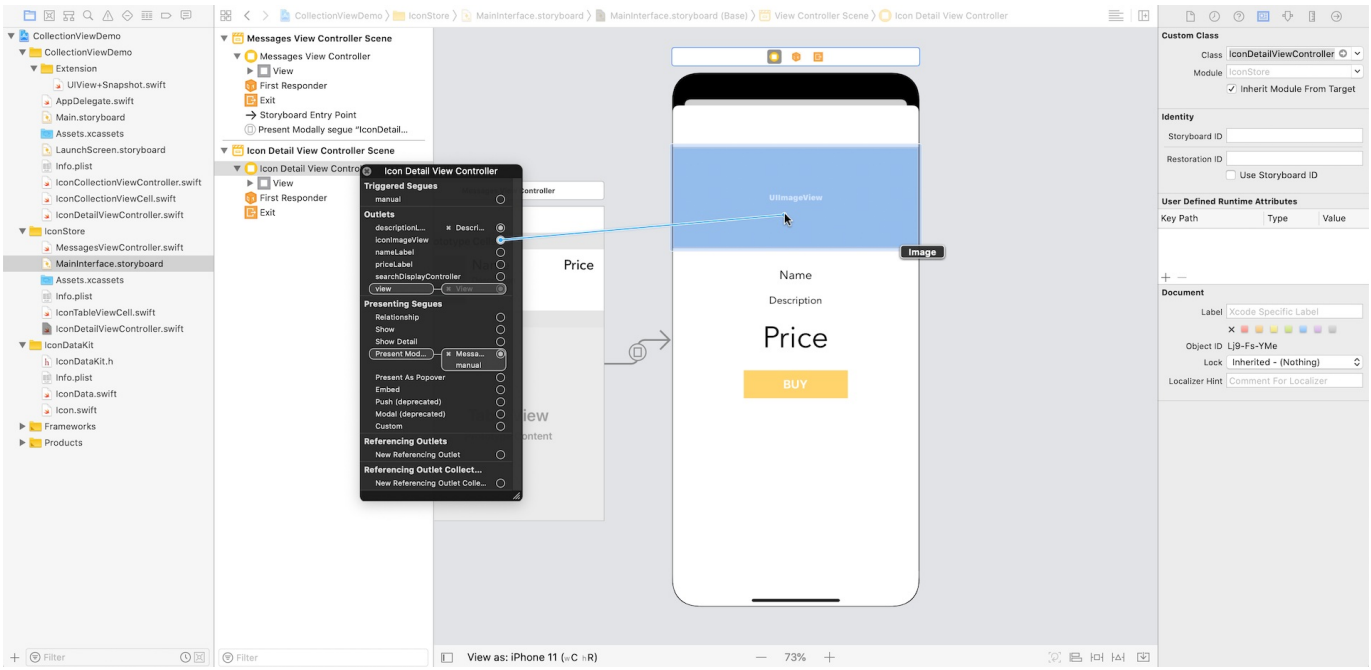


Figure 35.26. Establish a connection between the outlets and UI elements

Managing Message Selections and Extension States

How can we trigger the detail view controller when a user taps a message in the Messages app?

You will first have to understand how the `MSMessagesAppViewController` class works.

The `MSMessagesAppViewController` class has some built-in methods to track messages such as when the message is selected by a user, and when a user deletes a message from the input field. It also comes with methods that are invoked when the message extension transits from one state (e.g. inactive) to another (e.g. active).

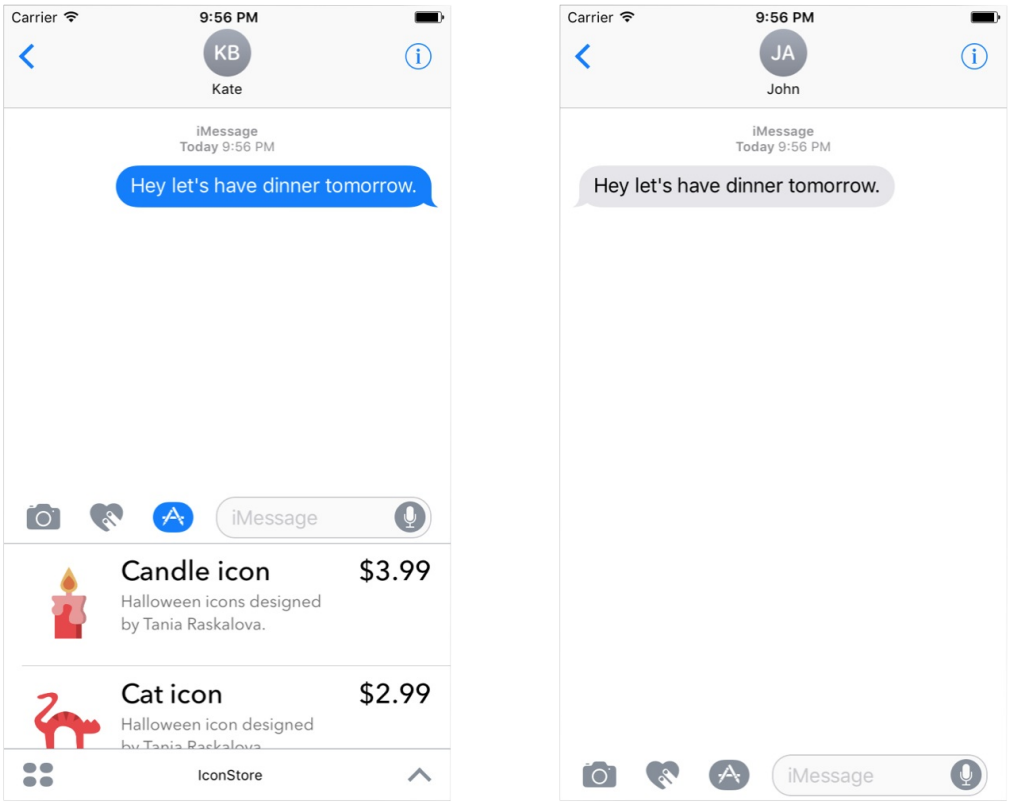


Figure 35.27. Message extension in the active state (left) and inactive state (right)

If a user selects one of messages in the conversation, while the extension is active, the `didSelect` method will be called. Both the `message` parameter and the conversation object's `selectedMessage` property contain the message selected by the user.

```
func didSelect(_ message: MSMessage, conversation: MSConversation)
```

It is quite obvious that we need to override this method with our own implementation so as to bring up the icon detail view controller. Let's first create a helper method like this in the `MessagesViewController.swift` file:

```
func presentIconDetails(message: MSMessage) {
    selectedIcon = Icon(message: message)
    performSegue(withIdentifier: "IconDetail", sender: self)
}
```

The method does a couple of things:

1. Create an icon object from the selected message.
2. Call the `performSegue` method with the specific identifier to present the detail view.

In order to pass the selected icon from the Messages View Controller to the Icon Detail View Controller, add the `prepare(for:sender:)` method like this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if let identifier = segue.identifier, identifier == "IconDetail" {
        let destinationController = segue.destination as! IconDetailViewController
        destinationController.icon = selectedIcon
    }
}
```

Now create the `didSelect` method like this:

```
override func didSelect(_ message: MSMessage, conversation: MSConversation) {
    guard let selectedMessage = conversation.selectedMessage else {
            return
    }

    presentIconDetails(message: selectedMessage)
}
```

When a message is selected, we call the `presentIconDetails` method to bring up the detail view and display the selected icon. You may test the message extension right now. Pick an icon and send it over to a recipient. But it is very likely you'll experience a couple of issues:

- On the sender side, you can reveal the details of the icon when you select the message in the conversation. However, when you close the detail view, it still appears in the message browser.
- On the receiving side, you can't reveal the icon details when you select a message.

For the first issue, we didn't dismiss the icon detail view controller. This is why you still see the icon detail view when the iMessage app returns to its compact mode.

The second issue is more complicated. Probably you expect the `didSelect` method is called when the message is selected by the recipient. The fact is that the method is only called while the message extension is in active mode. This is why you can't bring up the detail view controller on the receiving side.

The `MSMessagesAppViewController` class has several methods to manage the extension's state such as:

- `willBecomeActive(with:)` - invoked before the extension becomes active.
- `didBecomeActive(with:)` - invoked after the extension becomes active.
- `willResignActive(with:)` - Invoked before the message resigns its active status.
- `didResignActive(with: MSConversation)` - Invoked after the message resigns its active status.

And it has a number of methods that manage the presentation styles:

- `willTransition(to:)` - invoked before the extension transitions to a new presentation style. Say, the extension changes from compact mode to expanded mode.
- `didTransition(to:)` - invoked after the extension transitions to a new presentation style.

To resolve the first issue, we will implement the `willTransition(to:)` method and dismiss the modal view controller when the message extension returns to compact mode.

```
override func willTransition(to presentationStyle: MSMessagesAppPresentationStyle)
 {
    // Called before the extension transitions to a new presentation style.

    // Use this method to prepare for the change in presentation style.
    if presentationStyle == .compact {
        dismiss(animated: false, completion: nil)
        return
    }

}
```

For the second issue, the `willBecomeAction` method is the one we are interested in. When the message extension is activated by the user, the method will be first called. So we implement the method like this to present the icon detail view controller with the selected message:

```
override func willBecomeActive(with conversation: MSConversation) {
    // Called when the extension is about to move from the inactive to active stat
e.
    // This will happen when the extension is about to present UI.

    // Use this method to configure the extension and restore previously stored st
ate.
    guard presentationStyle == .expanded else {
        dismiss(animated: false, completion: nil)
        return
    }

    if let selectedMessage = conversation.selectedMessage {
        presentIconDetails(message: selectedMessage)
    }
}
```

Now it is ready to test the iMessage app again. You should be able to reveal the icon details when tapping a message in the conversation, even for the recipient.

Summary

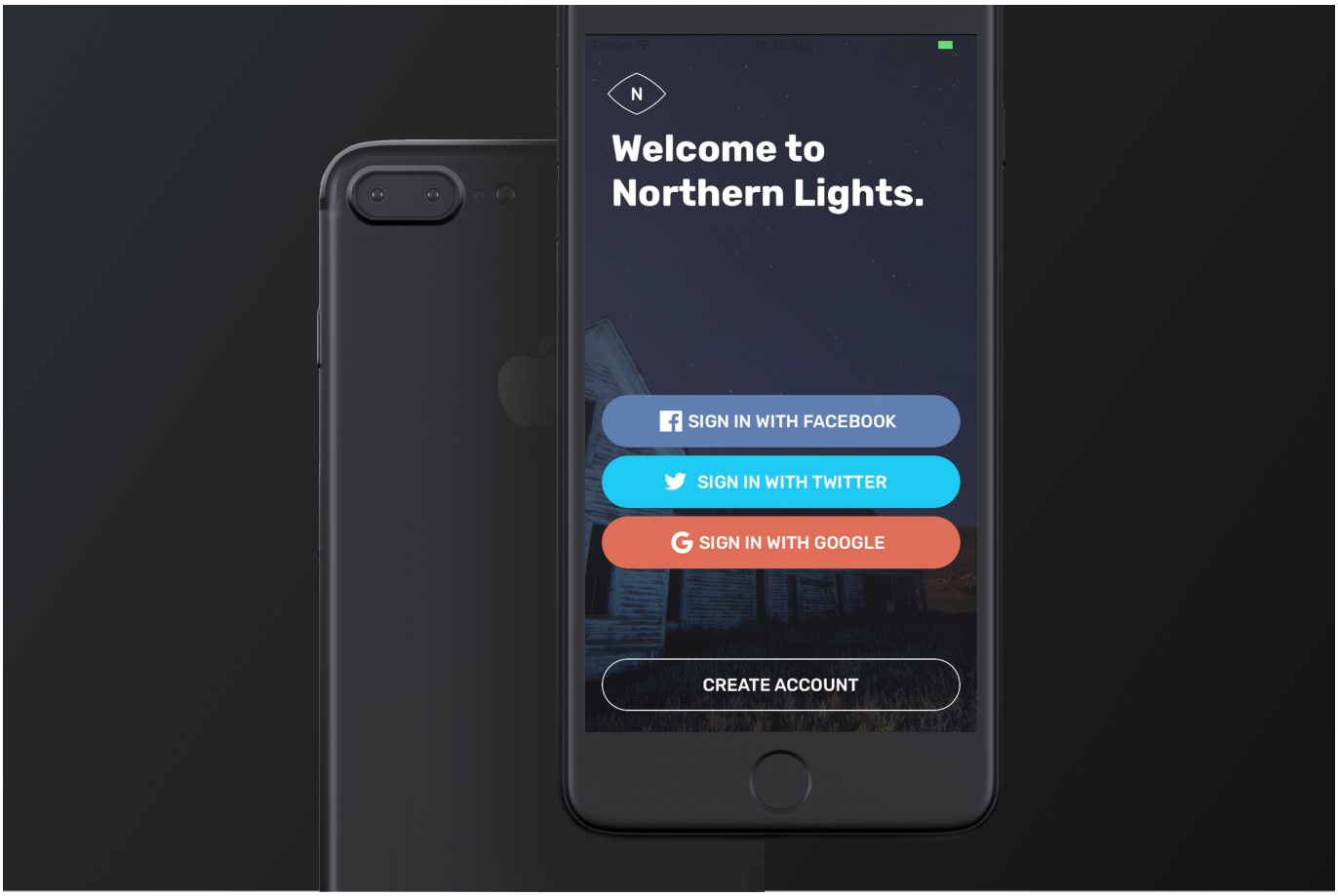
In this chapter, I have walked you through an introduction of iMessage apps. You now know how to create app extensions for the Messages app using the Message framework.

The launch of the new Message App Store opens up a lot of opportunities for iOS developers. As compared with the App Store, which has over 2 million apps, the Message App Store only has fewer apps when it first launches. It is still a good time to develop an iMessage app to reach more users. And, as mentioned at the beginning of the chapter, you can let your users help promote your app. Consider one sends an icon to a group of

users, and some of the users do not have your app installed, it is very likely some recipients will install the app in order to view the message details. So take some time to explore the Message framework and build an iMessage app!

For reference, you can download the complete project from http://www.appcoda.com/resources/swift5/iMessageApp.zip.

Chapter 36
Building Custom UI Components Using IBDesignable and IBInspectable



Some developers prefer not to use Interface Builder to design the app UI. Everything should be written in code, even for the UIs. Personally, I prefer to mix both storyboards and code together to layout the app.

But when it comes to teaching beginners how to build apps, Interface Builder is a no-brainer. Designing app UIs using Interface Builder is pretty intuitive, even for people without much iOS programming experience. One of the best features is that you can customize a UI component (e.g. button) without writing a line of code. For example, you

can change the background color or font size in the Attributes inspector. You can easily turn a default button into something more visually appealing by customizing the attributes.

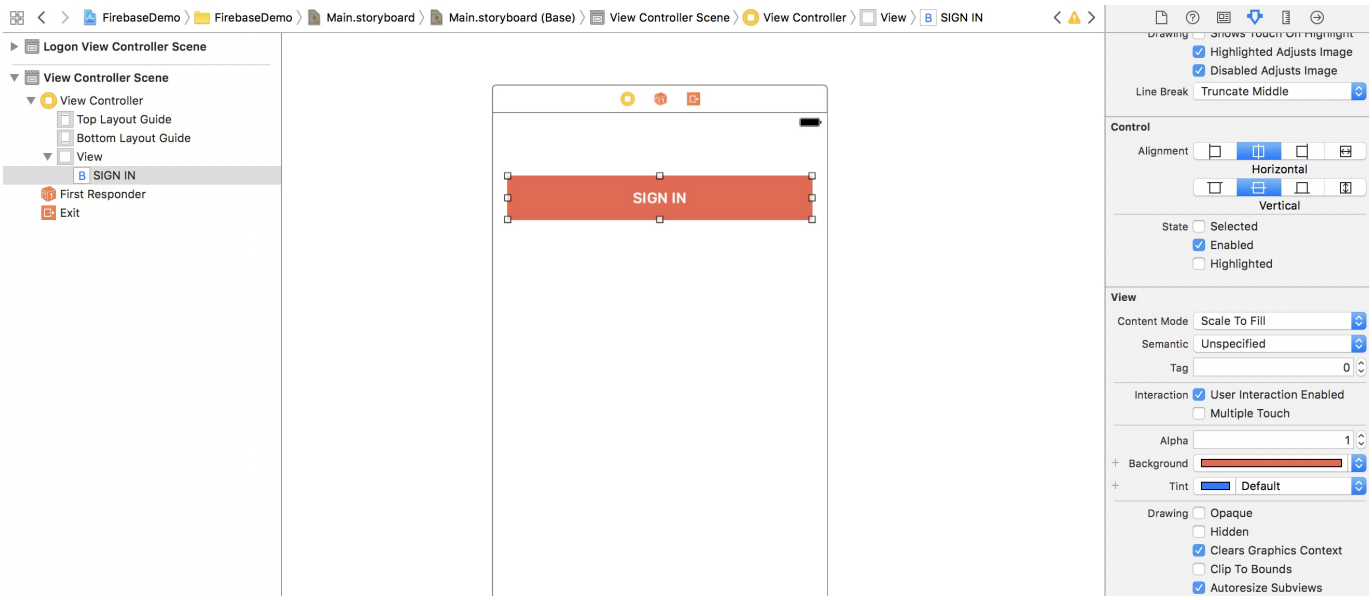


Figure 36.1. Designing a button in Interface Builder

That said, Interface Builder has its own limitation - not all attributes of a UI object are available for configuration. Let me ask you, can you create a button like this by using Interface Builder?

Figure 36.2. A more fancy button

To create a custom button like that, you still need to write code, or even develop your own class. This shouldn't be a big issue. But wouldn't it be great if you can design that button right in Interface Builder and view the result in real time?

IBInspectable and *IBDesignable* are the two keywords that make such thing possible. And, in this chapter, I will give you an introduction to both attributes and show you how to make use of them to create custom UI components.

Understanding IBInspectable and IBDesignable

In brief, *IBInspectable* allows you to add extra options in the Attributes inspector of Interface Builder, By indicating a class property of a `UIView` as *IBInspectable*, the property is then exposed to the Attributes inspector as an option. And, if you indicate a `UIView` class as *IBDesignable*, Interface Builder renders the custom view in real time. This means you can see how the custom view looks like as you edit the options.

To better understand *IBInspectable* and *IBDesignable*, let me give you an example.

Figure 36.3. A rounded corner button

You may be very familiar with the implementation of a rounded corner button. In case you have no idea about it, you can modify the layer's property to achieve that. Every view object is backed by a `CALayer`. To round the corners of a button, you set the `cornerRadius` property of the layer programmatically like this:

```
button.layer.cornerRadius = 5.0
button.layer.masksToBounds = true
```

A positive value of corner radius would cause the layer to draw rounded corners on its background. An alternative way to achieve the same result is to set the user defined runtime attributes in the Identity inspector.

Figure 36.4. Setting the runtime attributes of a button

User defined runtime attributes is already a powerful feature of Interface Builder that lets you configure the properties of a view. However, it is still not very intuitive. You have to remember each property of a view or look up the documentation for the required property.

IBInspectable was introduced in Xcode 6 to make view customization even better. It doesn't mean you do not need to write code. You still have to do it. But you are given the power to expose the properties to the Attributes inspector. To create a rounded corner button, you may declare a class like this and mark the `cornerRadius` property using `@IBInspectable`:

```
class RoundedCornerButton: UIButton {
    @IBInspectable var cornerRadius: CGFloat = 0.0 {
        didSet {
            layer.cornerRadius = cornerRadius
            layer.masksToBounds = cornerRadius > 0
        }
    }
}
```

Now, when you create a `RoundedCornerButton` object in the storyboard, Interface Builder adds an extra option named *Corner Radius* in the Attributes inspector, and make its value configurable.

Figure 36.5. The corner radius option now appears in the Attributes inspector

If you take a closer look at the name of the option, Xcode automatically converts the property name from `cornerRadius` to Corner Radius. It is a minor feature but this makes every option more readable.

The `cornerRadius` property has a type `CGFloat`. Interface Builder displays the *Corner Radius* option as a numeric stepper. Not all properties can be added in the Attributes inspector, according to Apple's documentation, *IBInspector* supports the following types:

- Int
- CGFloat
- Double
- String
- Bool
- CGPoint
- CGSize
- CGRect
- UIColor
- UIImage

If you declare a property as *IBInspectable* but out of the supported type, Interface Builder will not generate the option in the Attributes inspector.

While the keyword `@IBInspectable` allows developers to expose any of the view properties to Interface Builder, you cannot see the result on the fly. Every time you modify the value of corner radius, you will need to run the project before you can see how the button looks on screen.

IBDesignable further takes view customizations to another level. You can now mark a `UIView` class with the keyword `@IBDesignable` so as to let Interface Builder know that the custom view can be rendered in real time.

```
@IBDesignable class RoundedCornerButton: UIButton {
    @IBInspectable var cornerRadius: CGFloat = 0.0 {
        didSet {
            layer.cornerRadius = cornerRadius
            layer.masksToBounds = cornerRadius > 0
        }
    }
}
```

Using the same example as shown above (but with the keyword `@IBDesignable`), Interface Builder now renders the button on the fly for any property changes.

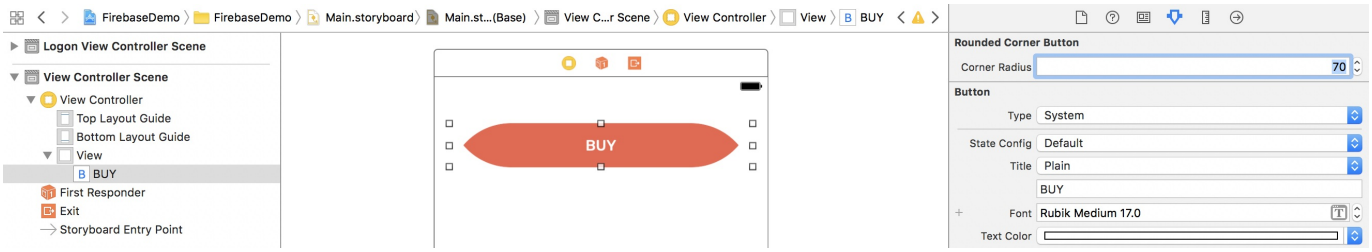


Figure 36.6. Interface Builder renders the button in the canvas

Creating a Fancy Button

Now that you should have some ideas about *IBInspectable* and *IBDesignable*, let's see how we can apply it in our daily work. The figure below displays a standard system button, and the fancy buttons which we are going to build.

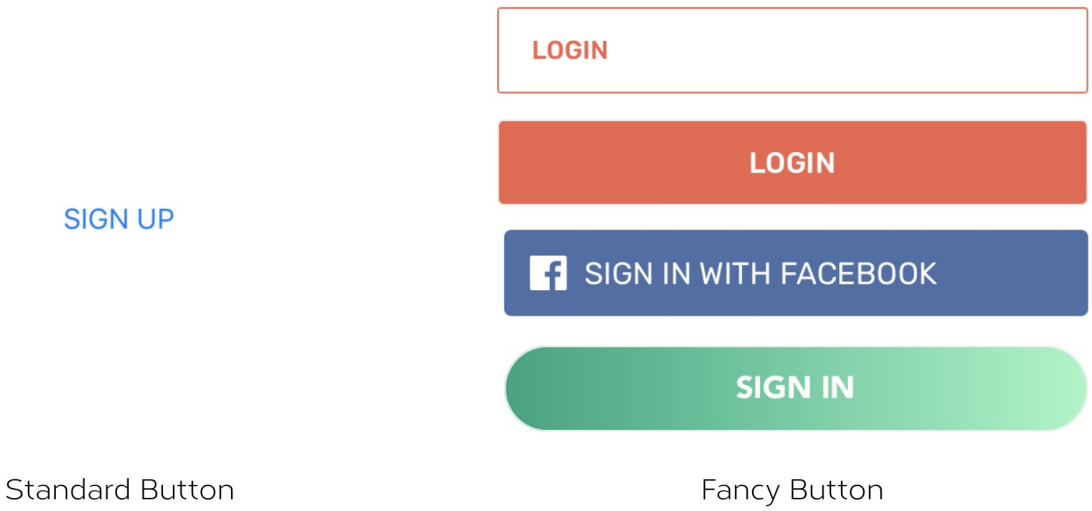


Figure 36.7. Standard button (left), Fancy button (right)

Since iOS 7, stock buttons are pretty much like a label but tappable. We plan to create a fancy button that is customizable through the Attributes inspector, and you can view the changes right in Interface Builder. This fancy button supports the following customizations:

- Corner radius
- Border width
- Border color
- Title padding for left, right, top and bottom sides
- Image padding for left, right, top and bottom sides
- Left/right image alignment
- Gradient color

Let's get started. First, create a new project using the Single View Application template and name it *FancyButton*.

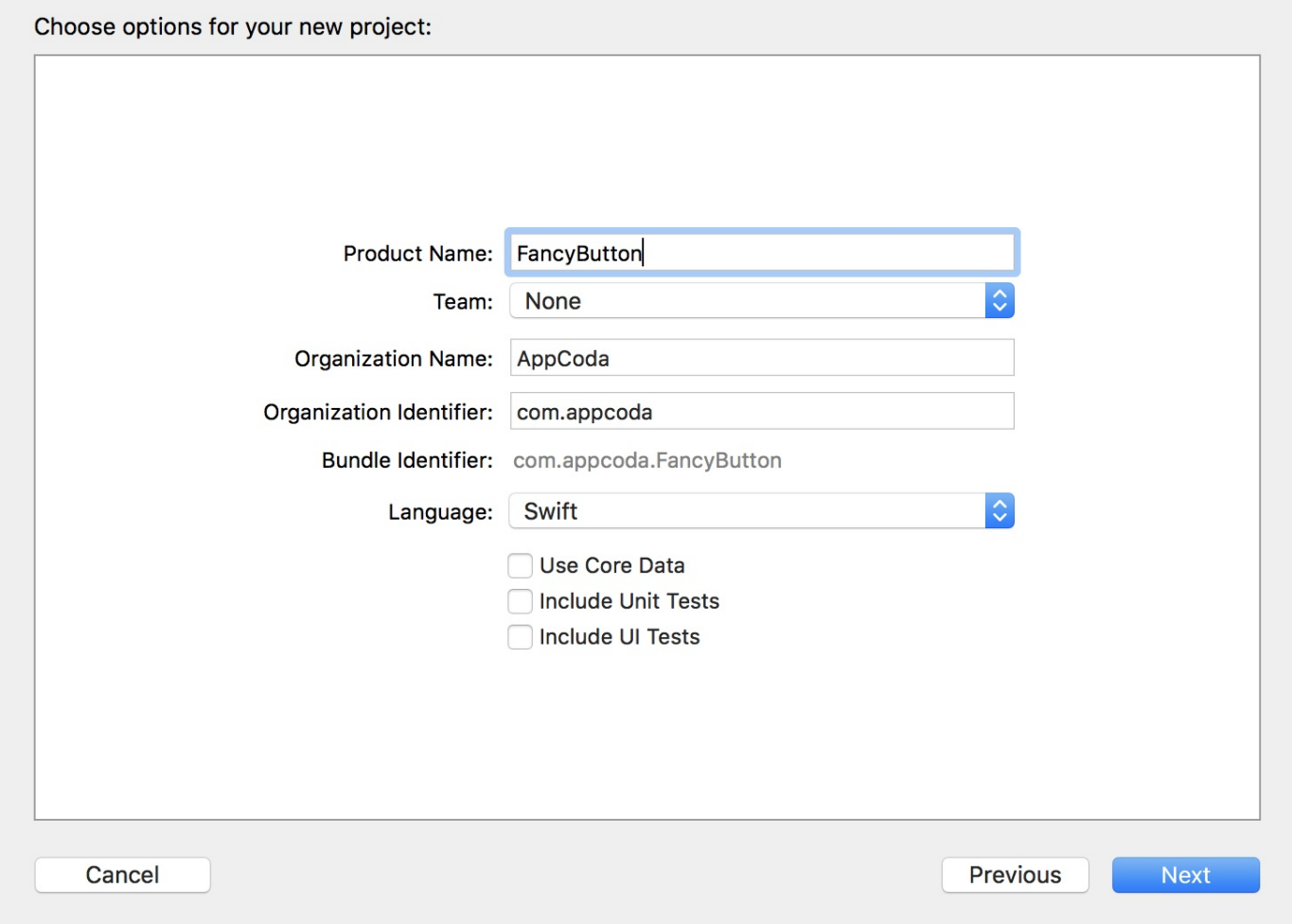


Figure 36.8. Create a new project and name it FancyButton

After creating the project, download this image pack, and add all the icons to the asset catalog.

Okay, we have the project configured. It is time to create the fancy button. We will create a custom class for this button. So right click *FancyButton* in the project navigator and select *New File...*. Choose the *Cocoa Touch Class* template. Name the new class *FancyButton* and set its subsclass to `UIButton`.

Corner Radius, Border Width and Border Color

Let's start with corner radius, border width and border color. Update the `FancyButton` class like this:

```
import UIKit

@IBDesignable
class FancyButton: UIButton {

    @IBInspectable var cornerRadius: CGFloat = 0.0 {
        didSet {
            layer.cornerRadius = cornerRadius
            layer.masksToBounds = cornerRadius > 0
        }
    }

    @IBInspectable var borderWidth: CGFloat = 0.0 {
        didSet {
            layer.borderWidth = borderWidth
        }
    }

    @IBInspectable var borderColor: UIColor = .black {
        didSet {
            layer.borderColor = borderColor.cgColor
        }
    }

}
```

We tell Interface Builder that `FancyButton` should be rendered in real time by adding the `@IBDesignable` keyword. And, we declare three properties (cornerRadius, borderWidth and borderColor) and make them *IBInspectable*.

Now open `Main.storyboard` to switch to Interface Builder. We will add a button to test out the `FancyButton` class. Drag a button object from the Object library to view controller, and change its title to **SIGN IN** (or whatever title you like). In the Identity inspector, change the custom class from default to `FancyButton`.

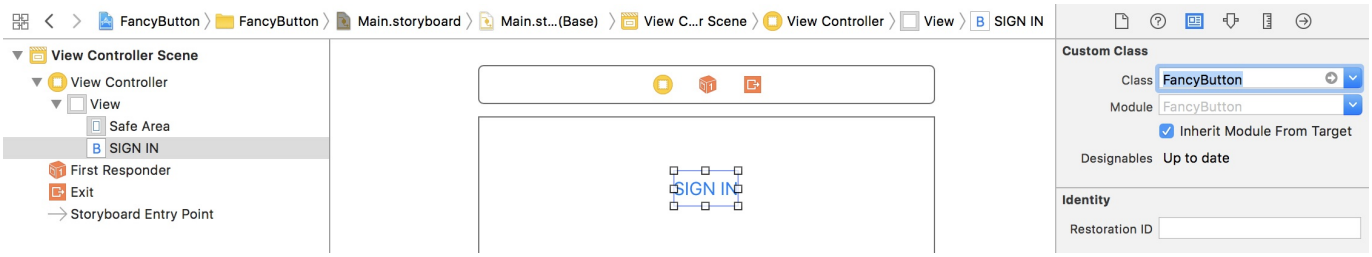


Figure 36.9. Set the custom class to FancyButton

It's time to see the magic happen! Go to the Attributes inspector, and you'll see a new section named *Fancy Button* with three options (including Corner Radius, Border Width and Border Color).

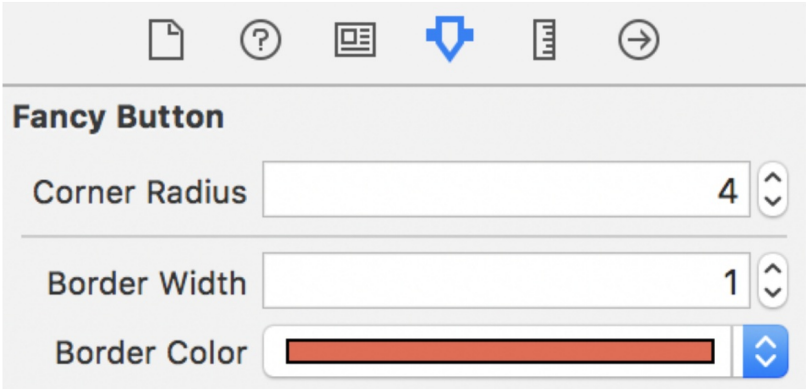


Figure 36.10. The Fancy button's properties appears in the Attributes inspector

You can now easily create a button like that shown in the figure below. If you want to create the same button. Resize it to 343 by 50 points. Set the corner radius to `4`, border width to `1`, and border color to `red`. You can try out other combinations to modify the look & feel of the button in real time.

Figure 36.11. Button with borders

Title and Image Padding

Now let's try to change the horizontal alignment of the control from *Centre* to *Left*, and see how it looks.

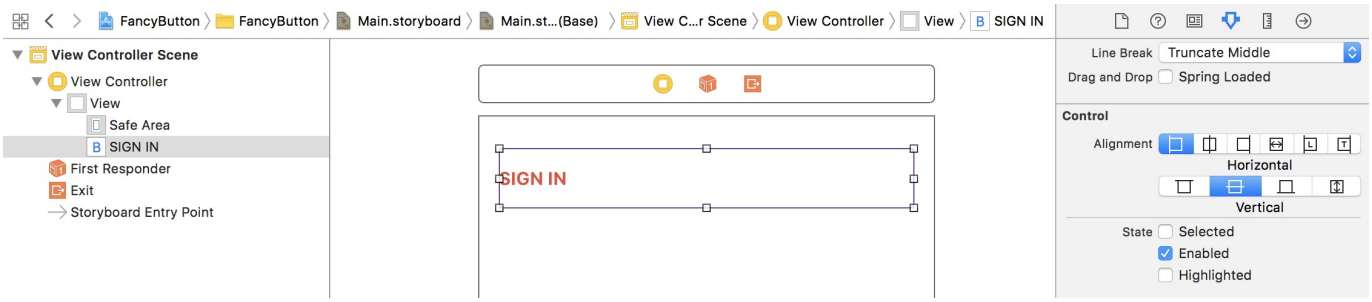


Figure 36.12. Changing the horizontal alignment from center to left

As you can see in the figure above, there is no space between the title label and the left edge. How do you add paddings to the title label? The `UIButton` class comes with a property named `titleEdgeInsets` for repositioning the title label. You can specify different values for each of the four insets (top, left, bottom, right). A positive value will move the title closer to the center of the button. Now modify the `FancyButton` class and add the following *IBInspectable* properties:

```
@IBInspectable var titleLeftPadding: CGFloat = 0.0 {
    didSet {
        titleEdgeInsets.left = titleLeftPadding
    }
}

@IBInspectable var titleRightPadding: CGFloat = 0.0 {
    didSet {
        titleEdgeInsets.right = titleRightPadding
    }
}

@IBInspectable var titleTopPadding: CGFloat = 0.0 {
    didSet {
        titleEdgeInsets.top = titleTopPadding
    }
}

@IBInspectable var titleBottomPadding: CGFloat = 0.0 {
    didSet {
        titleEdgeInsets.bottom = titleBottomPadding
    }
}
```

We add four new properties to allow any developers using `FancyButton` to configure the title label's padding in Interface Builder. You can now switch to `Main.storyboard` and test it out.

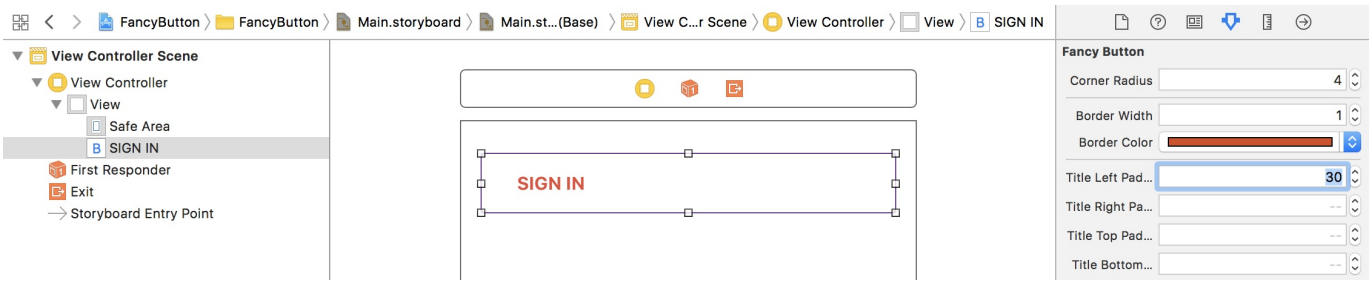


Figure 36.13. Configure the title label's padding

Buttons with Images

`UIButton` allows you to replace a title label with an image. You can set the title to *blank* and change the image option to `facebook` (which is the image you imported earlier). By varying the corner radius and border options, you can easily create buttons like this:

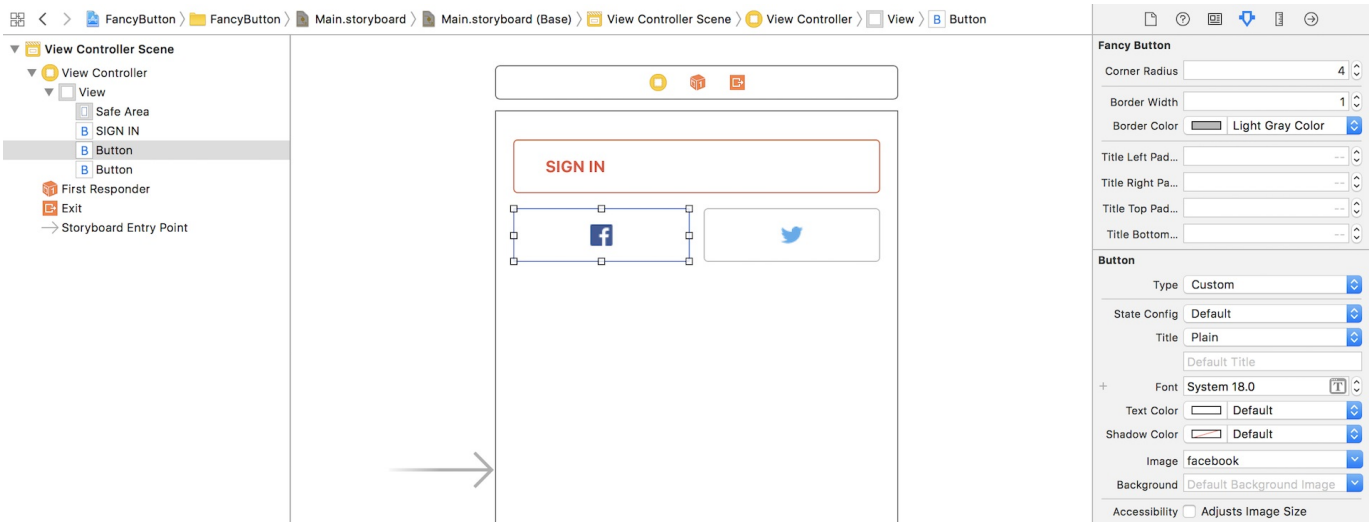


Figure 36.14. Buttons with image

With a configurable button component, you can create different button designs by applying different values. Let's say, you want to create a circular button with borders and an image. You can set the corner radius to half of the button's width, and set the border width to a positive value (say, 5). Figure 36.15 shows the sample buttons.

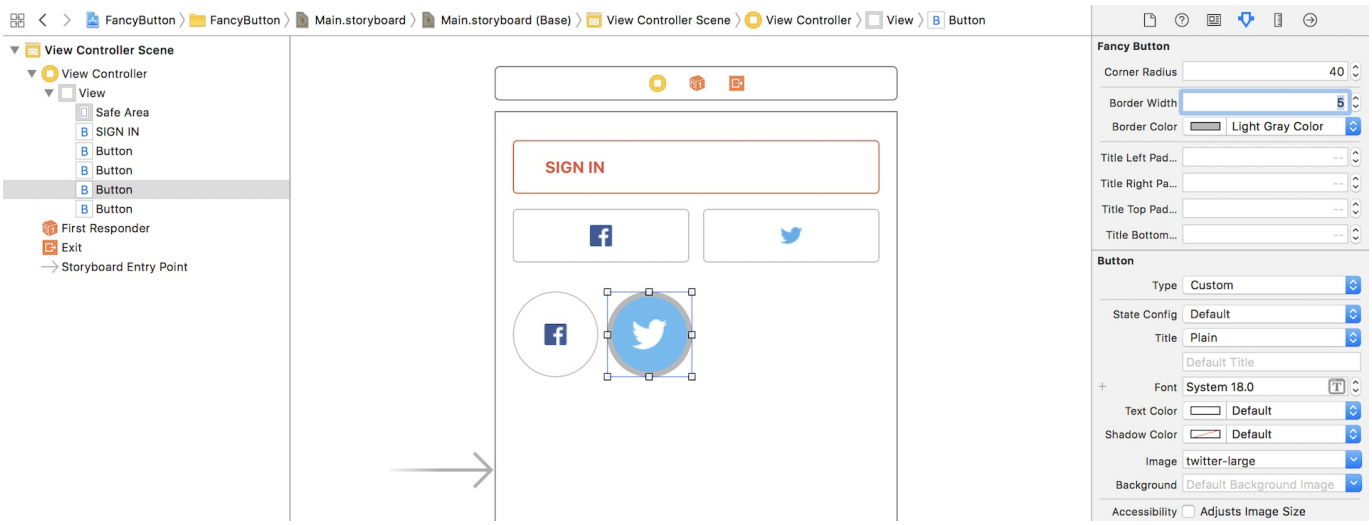


Figure 36.15. Buttons with image

Image Padding

In some cases, you want to include both title and images in the button. Let's say, you want to create a *Sign in with Facebook* button and the Facebook icon. You can set the button's title to *SIGN IN WITH FACEBOOK* and image to `Facebook`. The image is automatically placed to the left of the title.

As a side note, the facebook icon is in blue. If you want to change its color, you will need to change the button's type from *Custom* to *System*. The image will then be treated as a template image, and you can alter its color by changing the *Tint* option.

By default, there is no space between the Facebook image and the left edge of the button. Also, there is no space between the image and the title label. You can set the title's left padding to `20` to add a space but how can you add a padding for the Facebook image?

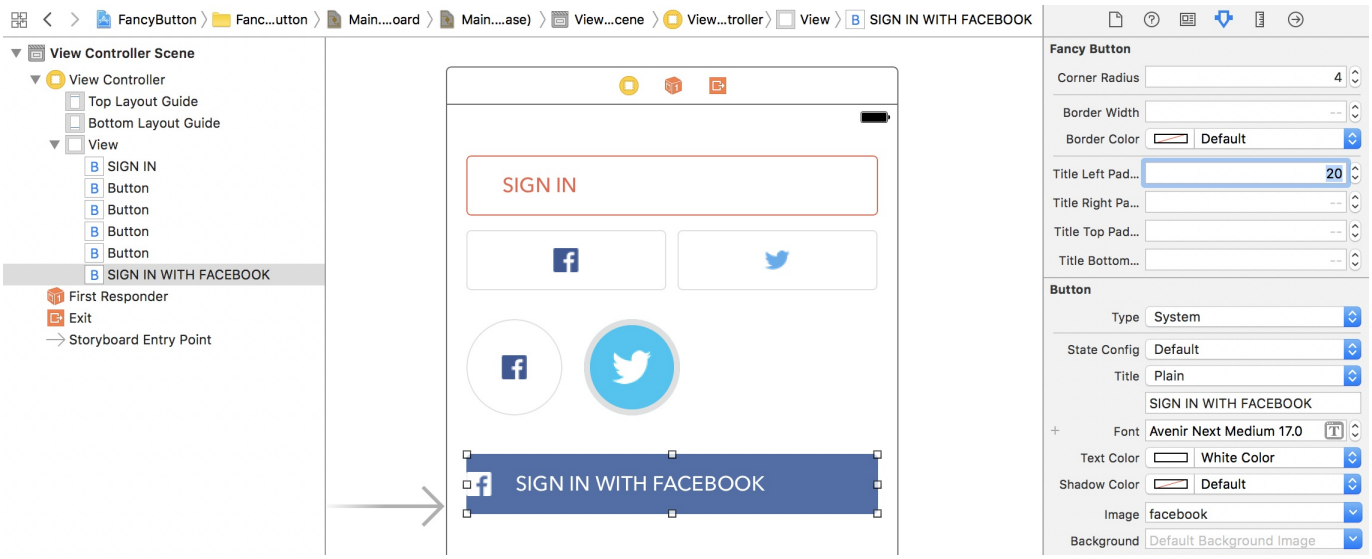


Figure 36.16. Buttons with both title and image

Similar to `titleEdgeInsets`, `UIButton` has another property named `imageEdgeInsets` for you to add padding around the image. Now open `FancyButton.swift` and insert the following `IBInspectable` property to the class:

```
@IBInspectable var imageLeftPadding: CGFloat = 0.0 {
    didSet {
        imageEdgeInsets.left = imageLeftPadding
    }
}

@IBInspectable var imageRightPadding: CGFloat = 0.0 {
    didSet {
        imageEdgeInsets.right = imageRightPadding
    }
}

@IBInspectable var imageTopPadding: CGFloat = 0.0 {
    didSet {
        imageEdgeInsets.top = imageTopPadding
    }
}

@IBInspectable var imageBottomPadding: CGFloat = 0.0 {
    didSet {
        imageEdgeInsets.bottom = imageBottomPadding
    }
}
```

After the changes, go back to the Interface Builder. You can now add a space between the image and the edge of the button's view by setting the value of *Image Left Padding*.

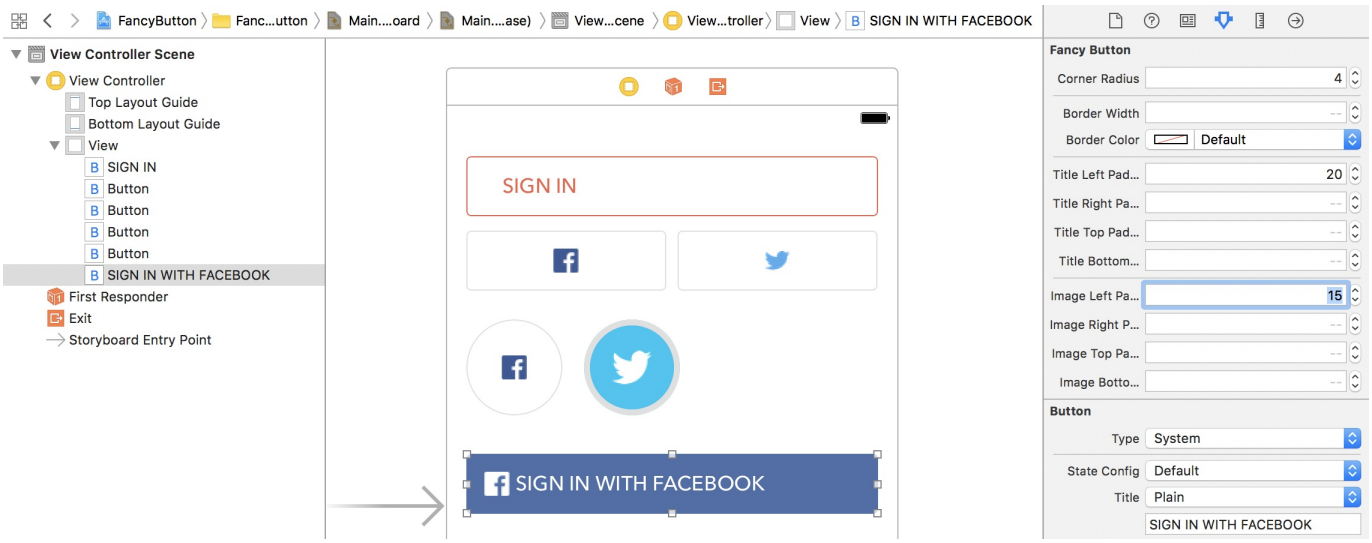


Figure 36.17. Adding padding for the image

Aligning the Image to the Right of the Title

By default, the image is aligned to the left of the button's title. What if you want to align the image to the right of the title? How can you do that?

There are multiple ways to do that. For me, I make use of the `imageEdgeInsets.left` property to achieve that.

Figure 36.18. Aligning the image view to the right of the button

Take a look at the above figure. To move the image view of a button to the right edge of the button, you can set the value of `imageEdgeInsets.left` to the following:

```
imageEdgeInsets.left = self.bounds.width - imageView.bounds.width
```

However, the above calculation doesn't include the right padding of the image view.

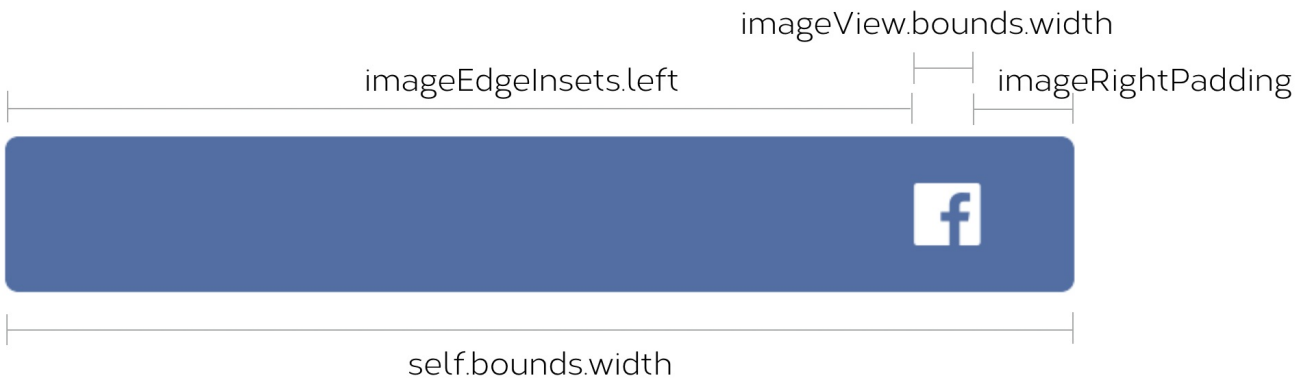


Figure 36.19. Right aligned image with padding

If we want to align the button's image like that shown in the figure, we have to change the formula like this:

```
imageEdgeInsets.left = self.bounds.width - imageView.bounds.width - imageRightPadd
ing
```

Now let's dive into the implementation. Insert the following code in the `FancyButton` class:

```
@IBInspectable var enableImageRightAligned: Bool = false

override func layoutSubviews() {
    super.layoutSubviews()

    if enableImageRightAligned,
        let imageView = imageView {
        imageEdgeInsets.left = self.bounds.width - imageView.bounds.width - imageR
ightPadding
    }
}
```

We add a property called `enableImageRightAligned` to indicate if the image should be right aligned. Later when you access the Attributes inspector, you will see an ON/OFF switch for you to choose.

Since we calculate the left padding (i.e. imageEdgeInsets.left) base on the the button's width (i.e. self.bounds.width) , we need to override the `layoutSubviews()` method and update the property there.

After applying the code changes, switch back to the storyboard and create another button using `FancyButton` . Now you can create a button like this by setting *Enable Image Right Aligned* to `ON` , and *Image Right Padding* to `20` .

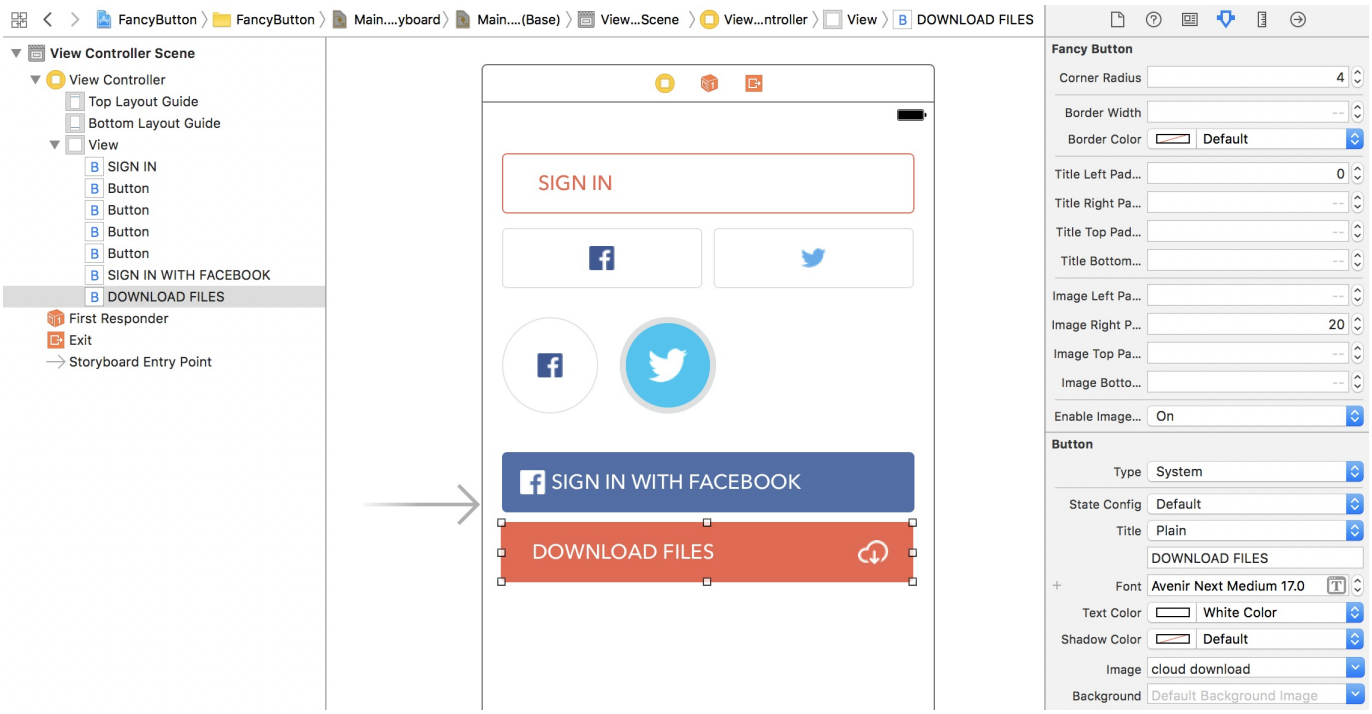


Figure 36.20. Creating a button with right aligned image

Color Gradient

A button can't be said as fancy if it doesn't support color gradient, right? So the last thing we will implement is to create an *IBInspectable* option for the `FancyButton` class.

So how can you create a gradient effect quickly and painlessly?

The iOS SDK has a class named `CAGradientLayer` that draws a color gradient over its background color. It is a subclass of `CALayer`, and allows developers to generate color gradients with a few lines of code like this:

```
let gradientLayer = CAGradientLayer()
gradientLayer.frame = self.bounds
gradientLayer.colors = [UIColor.blue, UIColor.red]
self.layer.insertSublayer(gradientLayer, at: 0)
```

A `CAGradientLayer` object has various properties for configuring the gradient effects. However, you basically need to provide two colors for the API to create the color gradient. In the above code, we set the first color to *blue* and the second color to *red*. If you put the code snippet in the `layoutSubviews()` method, you will see the result like this:

Figure 36.21. A sample gradient button

By default, as you can see, the direction of the gradient is from the top to the bottom. If you want to change the gradient direction to horizontal (say, from left to right), you can modify the `startPoint` and `endPoint` property like this:

```
gradientLayer.startPoint = CGPoint(x: 0.0, y: 0.5)
gradientLayer.endPoint = CGPoint(x: 1.0, y: 0.5)
```

Now that I have walked you through the basics of color gradient, let's modify the `FancyButton` class to support color gradient. Insert three new properties in the `FancyButton` class, and update the `layoutSubview()` method like below:

```
@IBInspectable var enableGradientBackground: Bool = false

@IBInspectable var gradientColor1: UIColor = UIColor.black

@IBInspectable var gradientColor2: UIColor = UIColor.white

override func layoutSubviews() {
    super.layoutSubviews()

    if enableImageRightAligned,
        let imageView = imageView {
        imageEdgeInsets.left = self.bounds.width - imageView.bounds.width - imageR
ightPadding
    }

    if enableGradientBackground {
        let gradientLayer = CAGradientLayer()
        gradientLayer.frame = self.bounds
        gradientLayer.colors = [gradientColor1.cgColor, gradientColor2.cgColor]
        gradientLayer.startPoint = CGPoint(x: 0.0, y: 0.5)
        gradientLayer.endPoint = CGPoint(x: 1.0, y: 0.5)
        self.layer.insertSublayer(gradientLayer, at: 0)
    }
}
```

The `enableGradientBackground` property indicates whether if you want to apply a gradient effect on the button. The other two properties let you define the colors of the gradient.

If the button is enabled with gradient, we create the `CAGradientLayer` object and apply the gradient effect using the given colors.

Now you're ready to test it in Interface Builder. You can enable the gradient option in Attributes inspector, set the two color options. However, there is one thing to note. Interface Builder is not capable to render the gradient effect in real time.

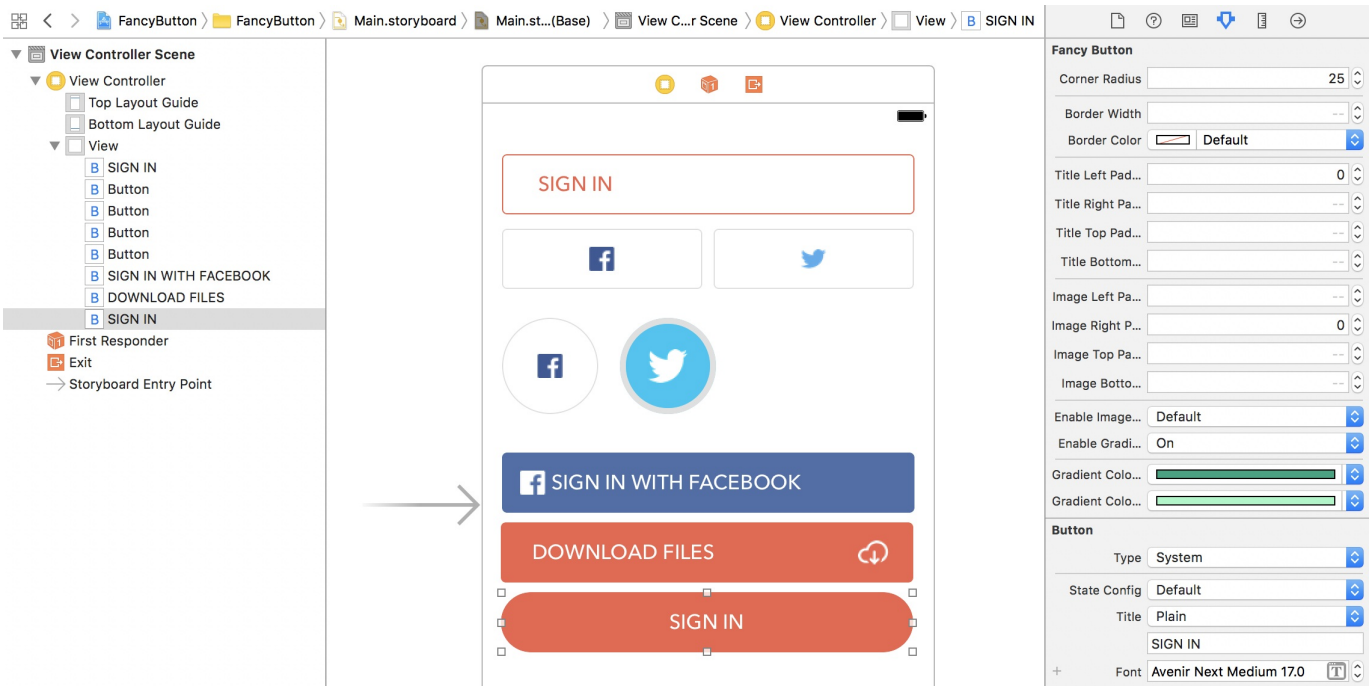


Figure 36.22. A sample gradient button

To see the gradient effect, you have to run the app in the simulator. Figure 36.23 shows the resulting gradient effect.

Figure 36.23. A sample gradient button

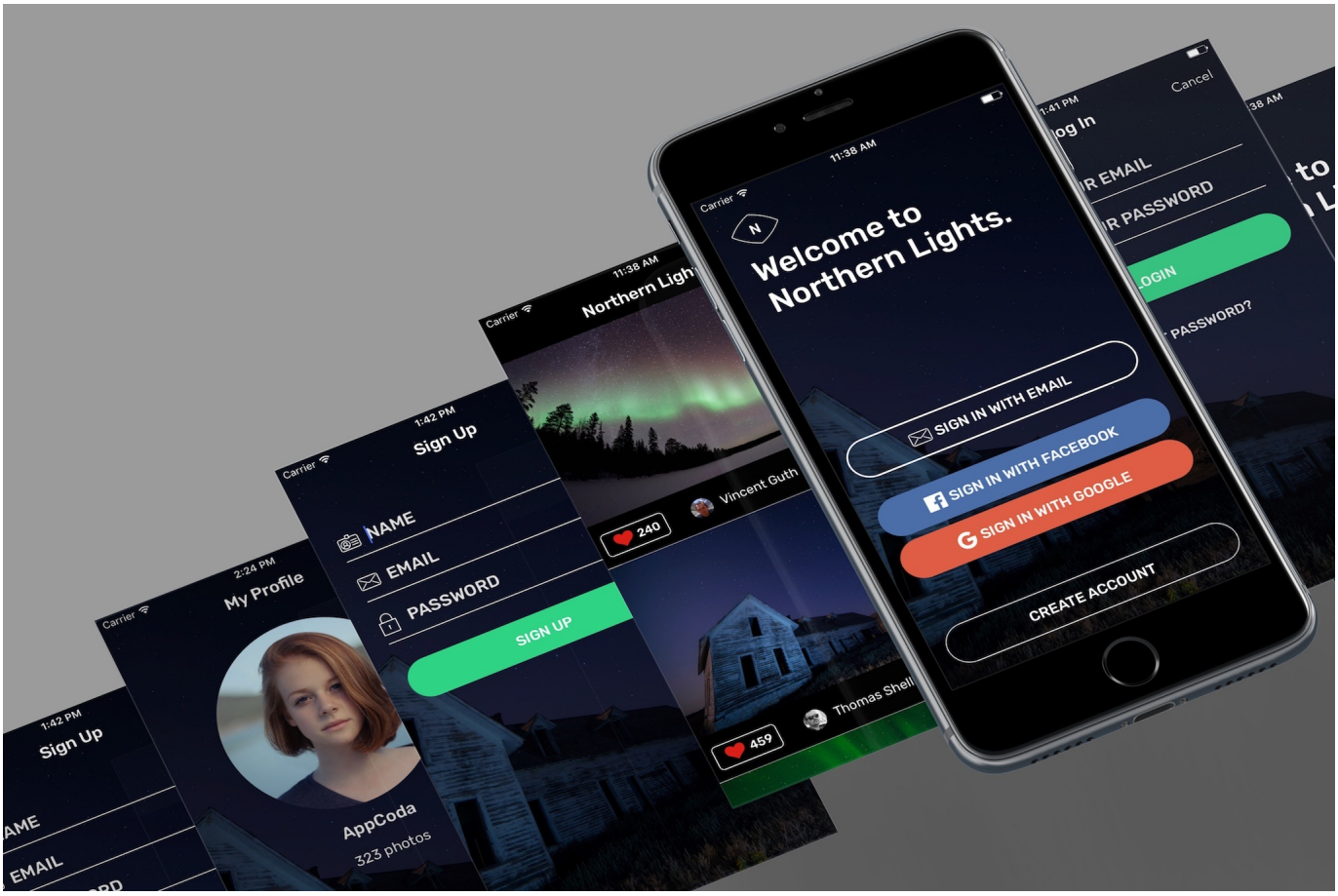
Summary

Isn't the Fancy Button cool? You now have a `FancyButton` class that can be reused in any Xcode projects. Or if you work in a team, you may share the class with other developers. They can start using it to build a fancy button right in storyboards and see the changes in real time.

IBInspectable and *IBDesignable* can be applied most view objects. As an exercise, try to create another customizable object and let developers configure its properties in Interface Builder.

For reference, you can download the demo project from http://www.appcoda.com/resources/swift5/IBDesignableDemo.zip.

Chapter 37
Using Firebase for User Authentication



Since Facebook announced the demise of Parse Cloud, a lot of app developers were looking for Parse alternatives. Among all available options, Firebase is one of the most popular choices for app developers to replace their apps' backend.

> **Note:** If you want to stick with Parse, you can refer to chapter 30 in which we show you how to keep using Parse through a third party provider.

One reason of its popularity is that Firebase is hosted and run by Google, which means the servers are powerful and reliable, so you do not have to worry about the stability of your app's backend. On top of that, Firebase supports nearly all kinds of platforms including iOS, Android, and web. It is very likely you are going to build apps for iOS as you're reading this book. However, in case you want to expand your apps to other platforms, Firebase is ready to support your projects.

Firebase is also used by some very big tech companies like PicCollage, Shazam, Wattpad, Skyscanner and other big start-ups so you can see how popular Firebase is.

As its name suggests, Firebase starts out as a cloud backend for mobile. In mid-2016, Google took Firebase further to become a unified app platform. Not only can you use it as a real-time database or for user authentication, it now can act as your analytic, messaging and notifications solutions for mobile apps.

In this chapter, I will focus on how to use Firebase for user authentication. Later, we will explore other features of Firebase.

```
Prerequisite: You will need to understand CocoaPods before you
can install Firebase in your Xcode project. If you have no idea
about what CocoaPods is, please read chapter 33 first.
```

The Demo App

As usual, the demo app is simple. But it doesn't mean it has to be ugly. You can first download the starter project (http://www.appcoda.com/resources/swift5/FirebaseLoginStarter.zip) to take a look.

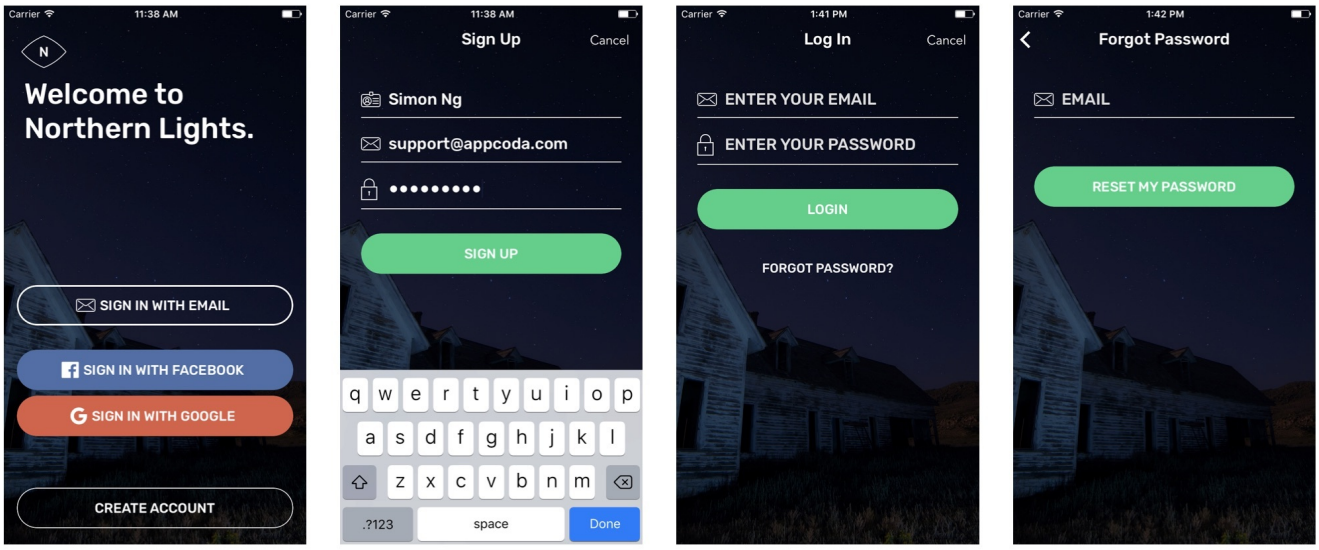


Figure 37.1. Sample screens of the demo app

To demonstrate how we utilize Firebase to implement the following user account-related functions:

- Sign up
- Login
- Logout
- Reset password
- Email verification

I have designed a few screens for this demo app. You can open the `Main.storyboard` file to take a look.

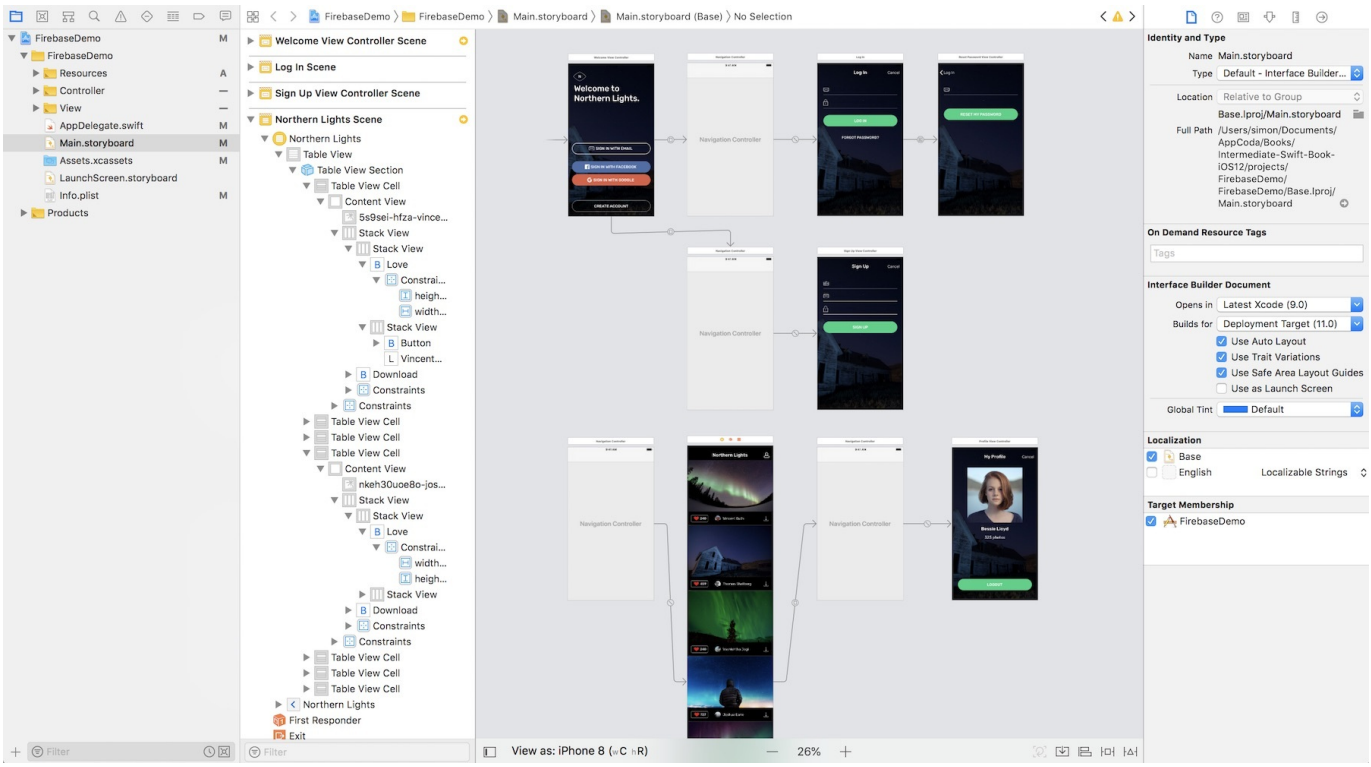


Figure 37.2. Storyboard

You are free to build the project and have a quick tour. When the app is first launched, it shows the welcome screen (i.e. Welcome View Controller) with login and sign up buttons. I have already linked the app views with segues. Tapping the `Create Account` button will bring up the Sign Up screen, while tapping the `Sign in with Email` button will show you the Login screen. If a user forgets the password, we also provide a `Forgot Password` function for him/her to reset the password.

I have built the home screen (i.e. Northern Lights View Controller) that shows a list of northern lights photos. You can't access this screen right now as we haven't implemented the Login function. But once we build them, the app will display the home screen after a user completes a login or sign up. In the home screen, the user can also bring up the profile screen by tapping the top-right icon.

Now that you should have some ideas about the demo app, let's get started to implement user authentication with Firebase. But before moving on, first change the bundle ID of the starter project. Select `FireBaseDemo` project in the project navigator, and then select

`FirebaseDemo` target. In the General tab, you can find the bundle identifier field under Identity section. It is now set to `com.appcoda.FirebaseDemo`. Change it other value (say, .FirebaseDemo). This value should be unique so that you can continue with the Firebase configuration.

Adding Firebase to the Xcode Project

Before you are allowed to use Firebase as your app's backend, you will have to go to https://firebase.google.com/ and register your app project.

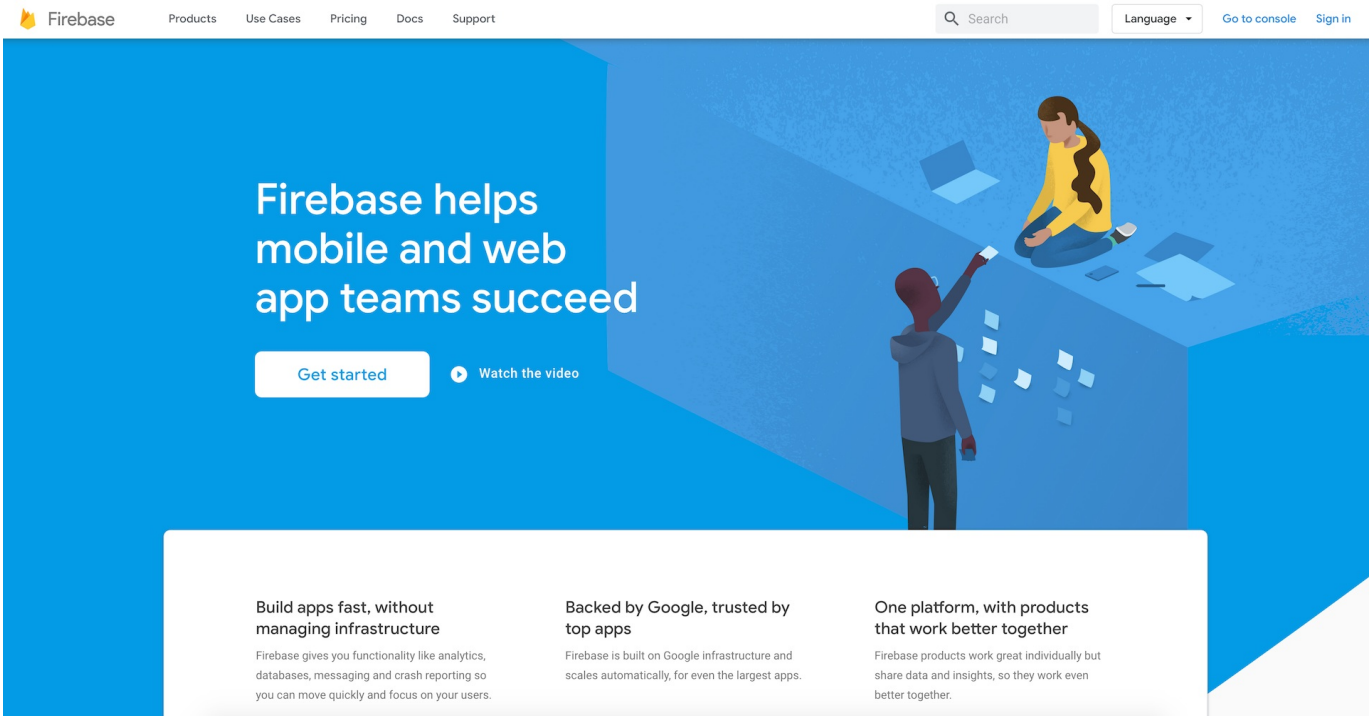


Figure 37.3. Firebase home page

As mentioned earlier, Firebase is a product of Google. You can sign into Firebase using your Google account. Once logged in, click `Go to Console` and then select `Create a Project`. It will take you to a screen where you name your project. Name your project whatever you want (e.g. Firebase Demo) and select your country/region. Optionally, you can choose to enable Google Analytics, though it's not required for this demo. Hit `Create Project` to continue.

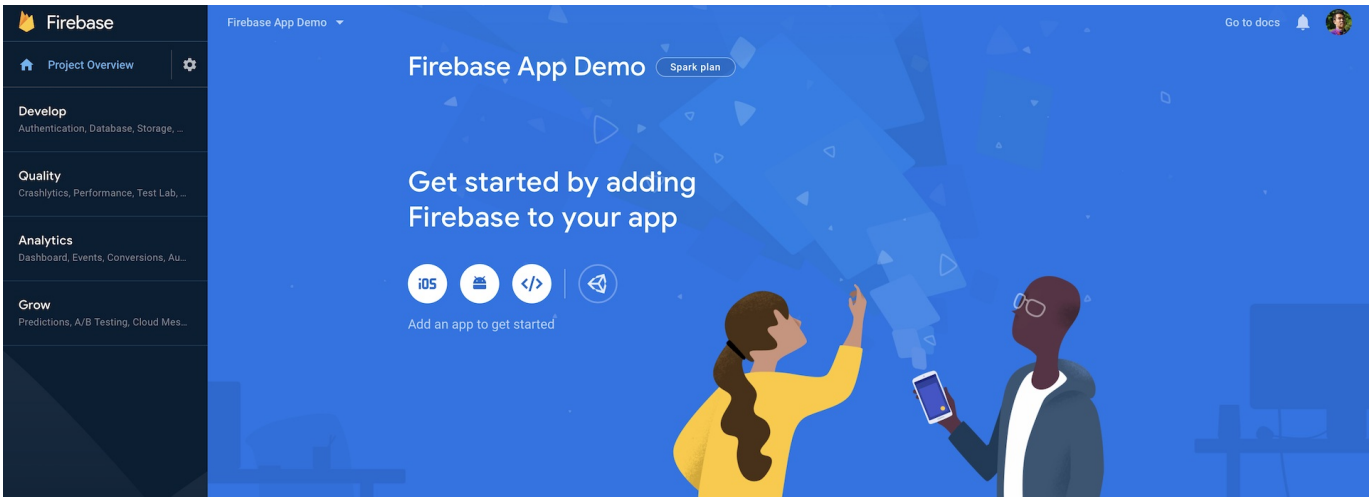


Figure 37.4. Firebase Dashboard

Once Firebase created the new project for you, it will take you to the dashboard of your project. This is where you can access all the features of Firebase such as database, notifications, and AdMob. In the overview, you should see three options for adding an app. Click the *iOS* icon. You'll then be prompted for filling in the bundle ID and app nickname. Here I use `com.appcoda.FirebaseAppIntDemo` as the bundle ID, but yours should be different from mine. Make sure this ID matches the one you set in the starter project earlier. For app nickname, you can fill in whatever you prefer. Like the nickname field, the App Store ID field is optional. If your app is already live on the App Store, you can add its ID.

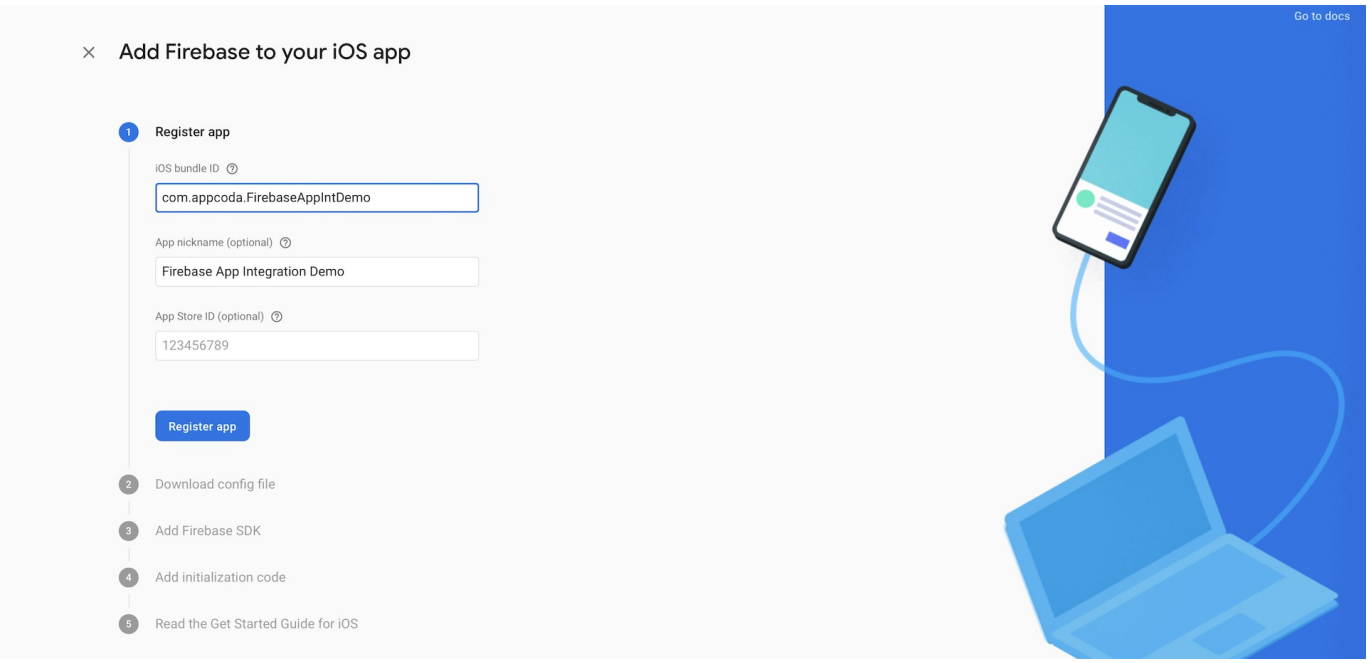


Figure 37.5. Filling in the bundle ID and App Nickname

When you finish, click the `Register App` button to proceed and Firebase will generate a file named `GoogleService-Info.plist` for you. Hit the `Download GoogleService-Info.plist` button to download it to your Mac computer.

This plist file is specifically generated for your own project. If you look into the file, you will find different identifiers for accessing the Firebase services such as AdMob and storage.

Now follow the on-screen instruction to add the `GoogleService-Info.plist` file (which can be found in your download folder by default) to your Xcode project.

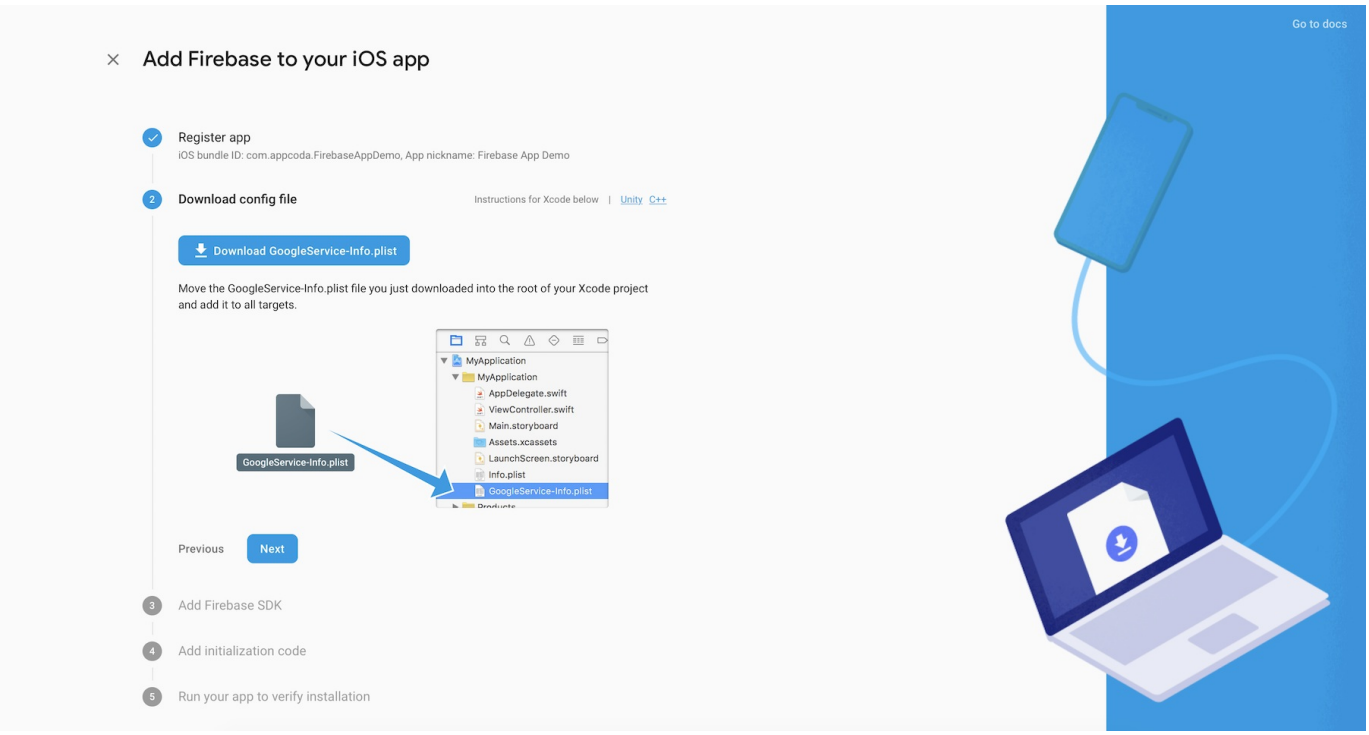


Figure 37.6. Adding GoogleService-Info.plist to your project

Next, press *Next* to proceed to the next step. The best way to install the Firebase library is through CocoaPods. This is why I mentioned before that you should have some understandings about CocoaPods before using Firebase. Now close your Xcode project and open Terminal.

Change to the directory of your Xcode project, and key in the following command to create a Podfile:

```
pod init
```

Once the file is created, open it and edit it like this:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'FirebaseDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for FirebaseDemo
  pod 'Firebase/Core'
  pod 'Firebase/Auth'

end
```

Here we specify to use the `Core` and `Auth` libraries of Firebase. This is all you need to implement user authentication and user profile configuration in this app project.

Now save the file and key in the following command in Terminal:

```
pod install
```

CocoaPods will start to install the dependencies and pods into the project.

When the pods are installed, you will find a new workspace file named `FirebaseDemo.xcworkspace`. Make sure you open the workspace file instead of the `FirebaseDemo.xcodeproj` file. Once opened, you should find the workspace installed with the Firebase libraries.

Implementing Sign Up, Login and Logout Using Firebase

Now that we have configured our project with the Firebase libraries, it is time to write some code. We will first implement the Sign Up and Login features for the demo app.

Initializing Firebase

To use Firebase, the very first thing is to call the `configure()` of `FirebaseApp`, the entry point of Firebase SDKs. By making this call, it reads the `GoogleService-Info.plist` file you added before and configures your app for using the Firebase backend.

We will call this API when the app is first launched. So select `AppDelegate.swift` in the project navigator. At the beginning of the file, insert the following line of code to import the Firebase API:

```
import Firebase
```

Next, update the `application(_:didFinishLaunchingWithOptions:)` method like this:

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

    // Set up the style and color of the common UI elements
    customizeUIStyle()

    // Configure Firebase
    FirebaseApp.configure()

    return true
}
```

Here we insert a line of code to call `FirebaseApp.configure()` to initialize and configure Firebase. This line of code helps you connect Firebase when your app starts up.

Now run the app. And then switch back to the Firebase dashboard to finish the app configuration. Click *Continue to console* to proceed.

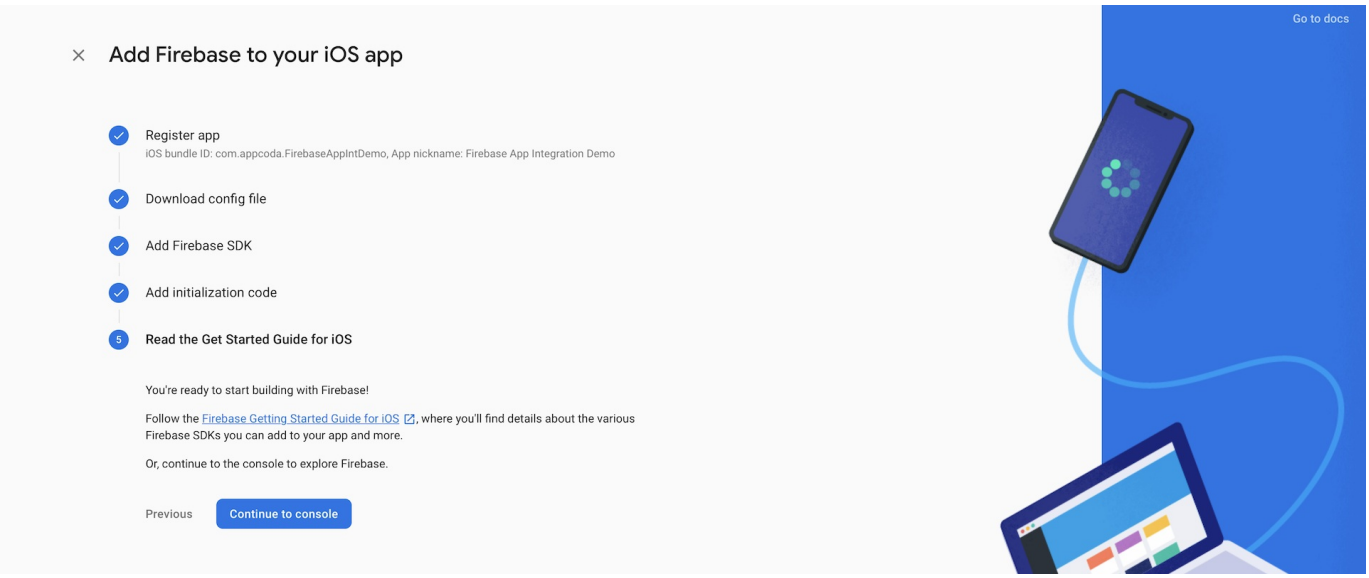


Figure 37.7. If your app initializes Firebase successfully, you should see the success message in tthe last step.

Sign Up Implementation

Now we're ready to do implement the Sign Up feature using Firebase. Firebase supports multiple authentication methods such as email/password, Facebook, and Twitter. In this demo, we will use the *email/password* approach.

To do that, go back to the Firebase dashboard. In the side menu, select *Develop* > *Authentication* and then choose `Set up Sign-in Method`. By default, all authentication methods are disabled. Now click *Email/Password* and turn the switch to ON. Save it and you'll see its status changes to *Enabled*.

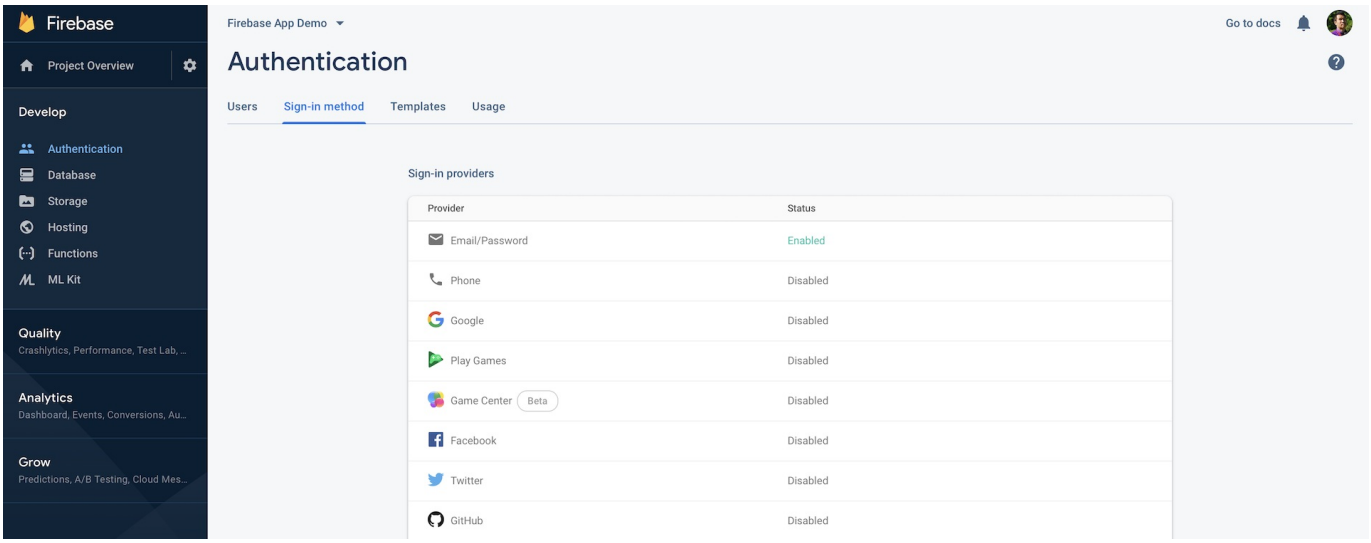


Figure 37.8. Configuring the Sign-in Method

Once this is enabled, you're now ready to implement the sign up and authentication feature.

Go back to Xcode and select `SignUpViewController.swift`. This is the view controller file for the Sign Up View Controller in storyboard. For the starter project, we haven't implemented any action method for the Sign Up button. This is what we're going to do now.

Similar to `AppDelegate.swift`, we will need to first add an import statement at the beginning of the file in order to use the Firebase APIs:

```
import Firebase
```

Next, we will add an action method called `registerAccount` in the `SignUpViewController` class. Insert the following code for the method:

```
@IBAction func registerAccount(sender: UIButton) {

    // Validate the input
    guard let name = nameTextField.text, name != "",
        let emailAddress = emailTextField.text, emailAddress != "",
```

```
        let password = passwordTextField.text, password != "" else {

            let alertController = UIAlertController(title: "Registration Error", m
essage: "Please make sure you provide your name, email address and password to com
plete the registration.", preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            present(alertController, animated: true, completion: nil)

            return
    }

    // Register the user account on Firebase
    Auth.auth().createUser(withEmail: emailAddress, password: password, completion
: { (user, error) in

        if let error = error {
            let alertController = UIAlertController(title: "Registration Error", m
essage: error.localizedDescription, preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            self.present(alertController, animated: true, completion: nil)

            return
        }

        // Save the name of the user
        if let changeRequest = Auth.auth().currentUser?.createProfileChangeRequest
() {
            changeRequest.displayName = name
            changeRequest.commitChanges(completion: { (error) in
                if let error = error {
                    print("Failed to change the display name: \(error.localizedDes
cription)")
                }
            })
        }

        // Dismiss keyboard
        self.view.endEditing(true)
```

```
        // Present the main view
        if let viewController = self.storyboard?.instantiateViewController(withIde
ntifier: "MainView") {
            UIApplication.shared.keyWindow?.rootViewController = viewController
            self.dismiss(animated: true, completion: nil)
        }
    })
}
```

Let me go through the above code line by line. If you've built the starter project and run the app before, you know the Sign Up screen has three fields: *name*, *email*, and *password*.

When this method is called, we first perform some simple validations. Here we just want to make sure the user fills in all the fields before we send the information to Firebase for account registration.

I prefer to use `guard` instead of `if` for input validations. When the conditions are not met (here, some fields are blank), it executes the code in the `else` block to display an error alert. If the user fills in all the required fields, it continues to execute the code in the method. In this scenario, `guard` makes our code more readable and clean.

Once we get the user information, we call `Auth.auth().createUser` with the user's email address and password. `Auth` is the class for managing user authentication. We first invoke its class method `auth()` to get the default auth object of our app. To register the user on Firebase, all you need is to call the `createUser` method with the email address and password. Firebase will then create an account for the user using the email address as the user ID.

The `createUser` method has a completion handler to tell you whether the registration is successful or not. You provide your own handler (or a closure) to verify the registration status and perform further processing. In our implementation, we first verify if there is any errors by checking the `error` object. In case the user registration fails, we display an alert message with the error returned. Some of the possible errors are:

- The email address is badly formatted.

- The email address already exists.

If there is no error (i.e. `error` object is nil), it means the registration is a success. Firebase will automatically sign in the account.

Apparently, the `createUser` method doesn't save the user's name for you. It only needs the user's email address and password to create the account for authentication. To set the user's name for the account, we can set the `displayName` property of the `User` object. When the sign up is successful, Firebase returns you the `User` object (here, it is the `user` variable) of the current user. This built-in user object has a couple of properties for storing profile information including display name and photo URL.

In the code above, we set the display name to the user's name. In order to update the user profile, we first call `createProfileChangeRequest()` to create an object for changing the profile data. Then we set its `displayName` properties and invoke `commitChanges(completion:)` to commit and upload the changes to Firebase.

The last part of the method is to dismiss the sign up view controller and replace it with the home screen (i.e. MainView or Northlights View). In the starter project, I have already set the navigation controller of the Northlights view with a storyboard ID named *MainView*. So in the code above, we instantiate the controller by calling `instantiateViewController` with the identifier and set it as the root view controller. Then we dismiss the Sign Up view controller. Now when the user completes the sign up, he/she will be able to access the main view.

Okay, we still miss one thing. We haven't connected the Sign Up button with the `registerAccount` action method yet. Go to `Main.storyboard` and locate the Sign Up View Controller. Control drag from the *Sign Up* button to *Sign Up View Controller* in the outline view (or the dock). In the popover menu, choose `registerAccountWithSender:` to connect the method.

Before moving to the next section, you can build the project and test the Sign Up function. After launching the app, tap `Create Account`, fill in the account information and tap `Sign Up`. You should be able to create an account on Firebase and access the home screen.

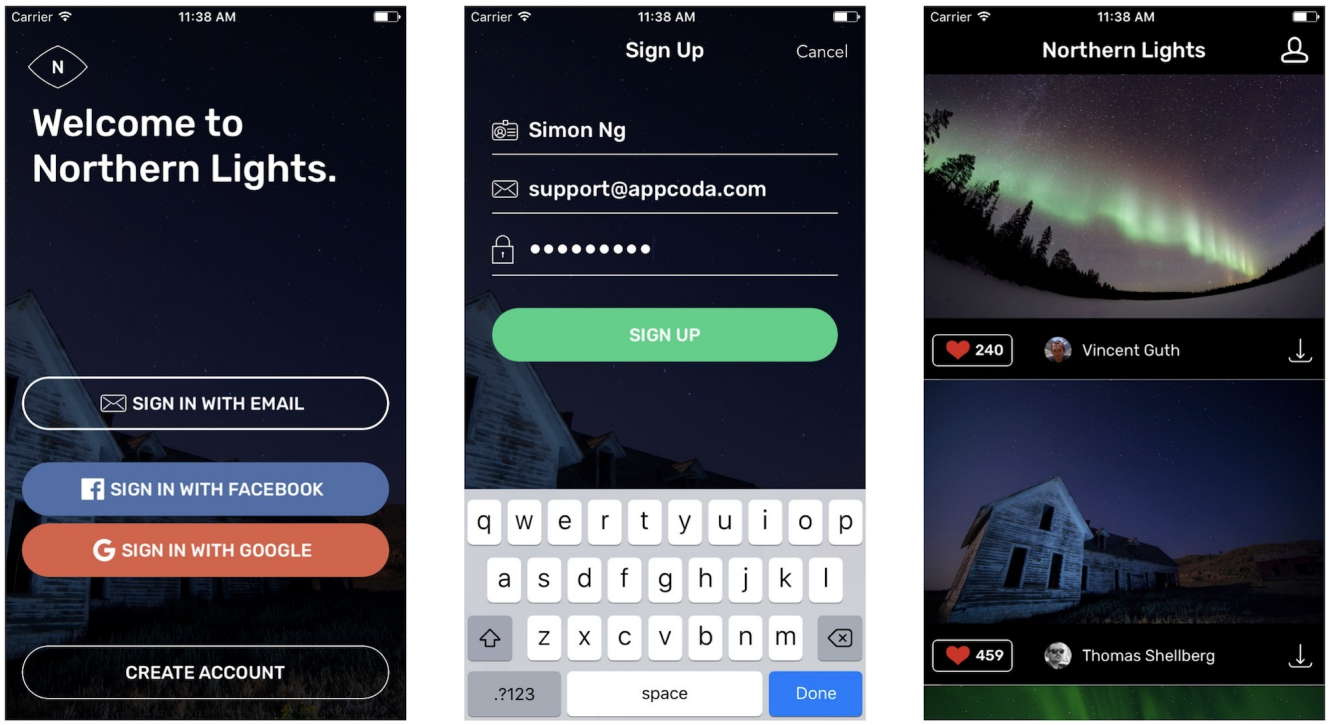


Figure 37.9. Signing up an account

If you go back to the Firebase console, you will find the user ID in the Users tab under the Authentication section.

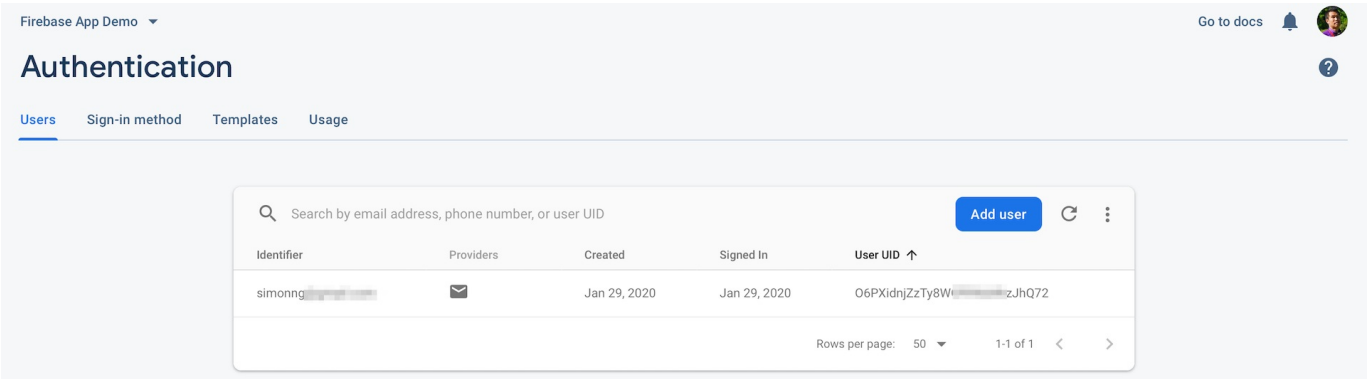


Figure 37.10. A sample user record in Firebase database

Login Implementation

Now that we have completed the implementation of the Sign Up feature, I hope you already have some ideas about how Firebase works. Let's continue to build the login function.

The implementation is very similar to Sign Up. With the Firebase SDK, you can implement the Login function with just a simple API call. In the project navigator, select `LoginViewController.swift`, which is the class that associates with the Login View Controller in the storyboard. If you use the starter project, the outlets are all connected with its corresponding text fields.

Again you need to import Firebase before using the APIs. So add the following line of code at the very beginning of the file:

```
import Firebase
```

Next, create a new action method called `login` in the class like this:

```
@IBAction func login(sender: UIButton) {

    // Validate the input
    guard let emailAddress = emailTextField.text, emailAddress != "",
        let password = passwordTextField.text, password != "" else {

            let alertController = UIAlertController(title: "Login Error", message:
"Both fields must not be blank.", preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            present(alertController, animated: true, completion: nil)

            return
    }

    // Perform login by calling Firebase APIs
    Auth.auth().signIn(withEmail: emailAddress, password: password, completion: {
(user, error) in

        if let error = error {
```

```
            let alertController = UIAlertController(title: "Login Error", message:
 error.localizedDescription, preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            self.present(alertController, animated: true, completion: nil)

            return
        }

        // Dismiss keyboard
        self.view.endEditing(true)

        // Present the main view
        if let viewController = self.storyboard?.instantiateViewController(withIde
ntifier: "MainView") {
            UIApplication.shared.keyWindow?.rootViewController = viewController
            self.dismiss(animated: true, completion: nil)
        }

    })
}
```

As you can see, the code is quite similar to the `registerAccount` method we discussed earlier. At the very beginning of the method, we validate the user input to ensure that both fields are not blank.

Assuming the user has filled his/her email address and password, we call the `signIn` method of the Firebase API to perform the login. The method accepts three parameters including the email address (i.e. the login ID) and password. The third parameter is the completion handler. When the sign in completes, Firebase returns us the result of the operation through the completion handler.

Similar to what we have done before, we check if there is an error. If everything is perfect, we dismiss the on-screen keyboard, and bring the user to the home screen (i.e. MainView).

Lastly, before you test the app, switch to `Main.storyboard` and locate the Login View Controller. You will have to connect the action method with the *Log In* button. Control drag from the *Log In* button to the view controller in the dock or document outline. Select `loginWithSender:` from the popover menu.

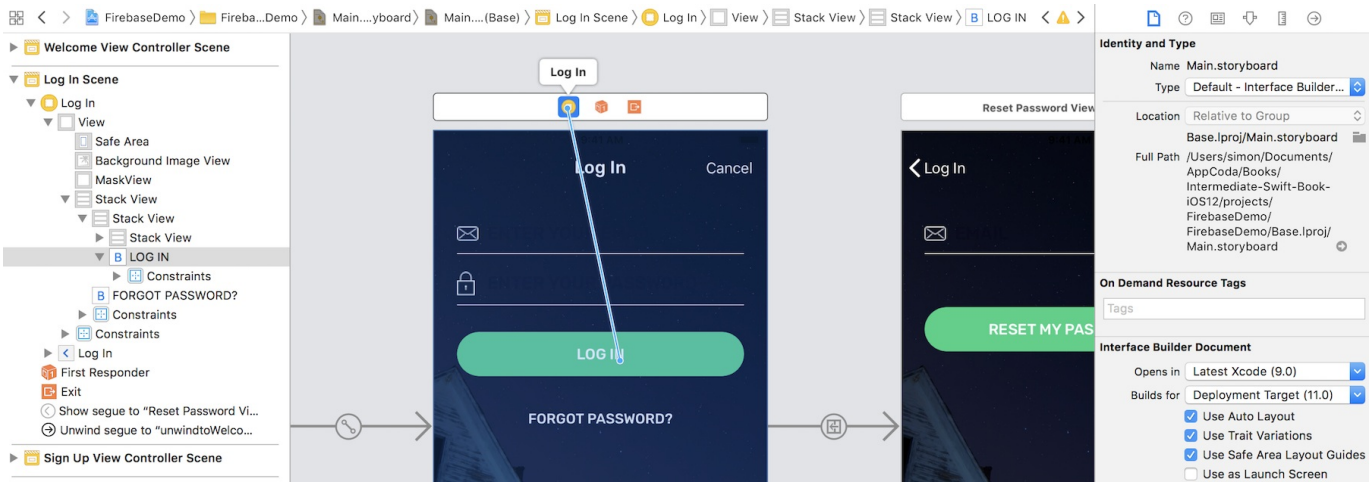


Figure 37.11. Connecting the Log In button with the action method

Now you're ready to test the login function. Use the same account that you signed up earlier to test the login. The app should bring you to the home screen if everything is correct. Or it will display you some errors.

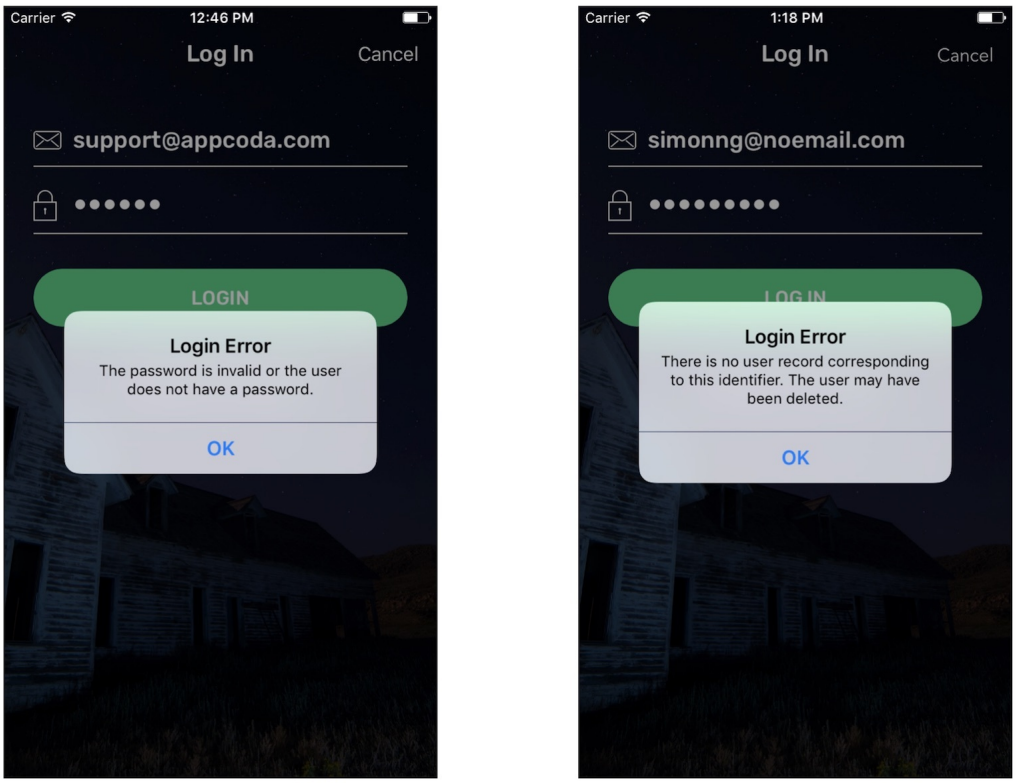


Figure 37.12. Sample login errors returned by Firebase

Logout Implementation

If you fully understand how to implement sign up and login, it should not be difficult for you to implement the logout function. All you need is refer to the Firebase documentation and see which API is suitable for logout.

Anyway, let's go back to Xcode and continue to implement the feature.

The logout button can be found in the profile view controller. Therefore, select `ProfileViewController.swift` and add an import statement to use the Firebase APIs:

```
import Firebase
```

Next, create a new action method called `logout` in the class:

```
@IBAction func logout(sender: UIButton) {
    do {
        try Auth.auth().signOut()

    } catch {
        let alertController = UIAlertController(title: "Logout Error", message: er
ror.localizedDescription, preferredStyle: .alert)
        let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: nil)
        alertController.addAction(okayAction)
        present(alertController, animated: true, completion: nil)

        return
    }

    // Present the welcome view
    if let viewController = self.storyboard?.instantiateViewController(withIdentif
ier: "WelcomeView") {
        UIApplication.shared.keyWindow?.rootViewController = viewController
        self.dismiss(animated: true, completion: nil)
    }

}
```

To log out a user, you just need to call the `signOut()` method of `Auth`. The method throws an error if the logout is unsuccessful. We simply present an alert prompt to display the error message. And, if the logout performs properly, we dismiss the current view and bring the user back to the welcome screen, which is the original screen when the app is first launched.

Again, remember to go to the storyboard and connect the action method with the *Logout* button. Locate the Profile View Controller. Control drag from the *Logout* button to the Profile View Controller in the document outline view (or the dock). Choose `logoutWithSender:` when prompted to connect the button with the action method.

For demo purpose, the profile screen now shows a static profile photo and a sample name. As the user has provided his/her name during sign up, you may wonder whether we can retrieve the name from Firebase and display it in the profile view.

With the Firebase SDK, it turns out that it is pretty straightforward to do that. If you are not forgetful, you should remember we have set the display name of the user. We can retrieve that information and set it to the label.

To do that, update the `viewDidLoad()` method like this:

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

    self.title = "My Profile"

    if let currentUser = Auth.auth().currentUser {
        nameLabel.text = currentUser.displayName
    }
}
```

We can retrieve the current user object by accessing the `currentUser` property of the authentication object. Then we just assign the display name to the name label.

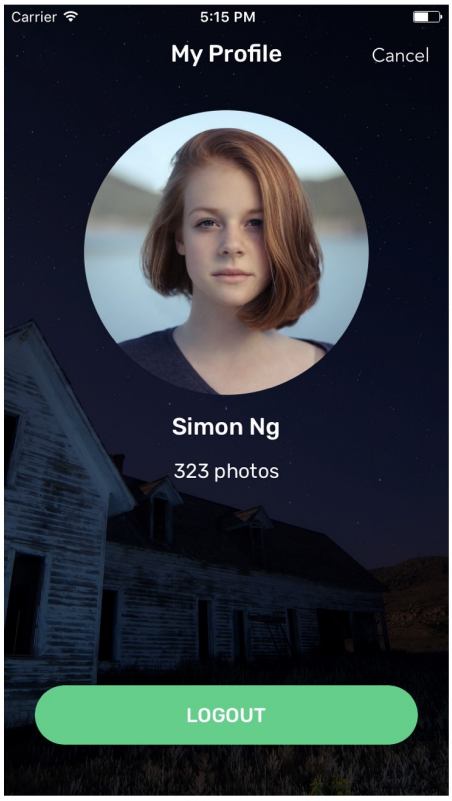


Figure 37.13. The user profile screen

Resetting the User's Password

What happens if the user forgot his/her login password? We are going to implement the password reset function for users to reset the password. Again it is a simple API call.

In Xcode, open the `ResetPasswordViewController.swift` file. This is the class for the Reset Password view controller in the storyboard. First, insert the import statement for accessing the Firebase APIs:

```
import Firebase
```

Next, we will create an action method named `resetPassword`. Insert the following code in the class:

```
@IBAction func resetPassword(sender: UIButton) {

    // Validate the input
    guard let emailAddress = emailTextField.text,
        emailAddress != "" else {

            let alertController = UIAlertController(title: "Input Error", message:
"Please provide your email address for password reset.", preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            present(alertController, animated: true, completion: nil)

            return
    }

    // Send password reset email
    Auth.auth().sendPasswordReset(withEmail: emailAddress, completion: { (error) in


        let title = (error == nil) ? "Password Reset Follow-up" : "Password Reset
Error"
        let message = (error == nil) ? "We have just sent you a password reset ema
il. Please check your inbox and follow the instructions to reset your password." :
 error?.localizedDescription

        let alertController = UIAlertController(title: title, message: message, pr
eferredStyle: .alert)
        let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: { (ac
tion) in

            if error == nil {

                // Dismiss keyboard
                self.view.endEditing(true)

                // Return to the login screen
                if let navController = self.navigationController {
                    navController.popViewController(animated: true)
                }
            }
```

```
            })
            alertController.addAction(okayAction)

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

    }
```

Similar to what we have implemented for other methods, we first validate the user's input at the very beginning. We just want to make sure the user provides the email address, which is the account ID for password reset.

Once the validation is done, we call the `sendPasswordReset` method with the user's email address. If the given email address can be found in Firebase, the system will send a password reset email to the specified email address. The user can then follow the instructions to reset the password.

In the code above, we display an alert prompt showing either an error or a success message after the `sendPasswordReset` method call. If it is a success, we ask the user to check the inbox, and then the app automatically navigates back to the login screen.

Before testing the app, make sure you go back to `Main.storyboard`. Locate the Reset Password View Controller. Control drag from the *Reset Password* button to the Reset Password View Controller in the outline view (or the dock). Choose `resetPasswordWithSender:` to connect the action method.

Now build the app and test it. Go to the Reset Password screen and fill in your email address. You will receive a password reset email after hitting the Reset Password button. Just follow the instructions and you can reset the password.

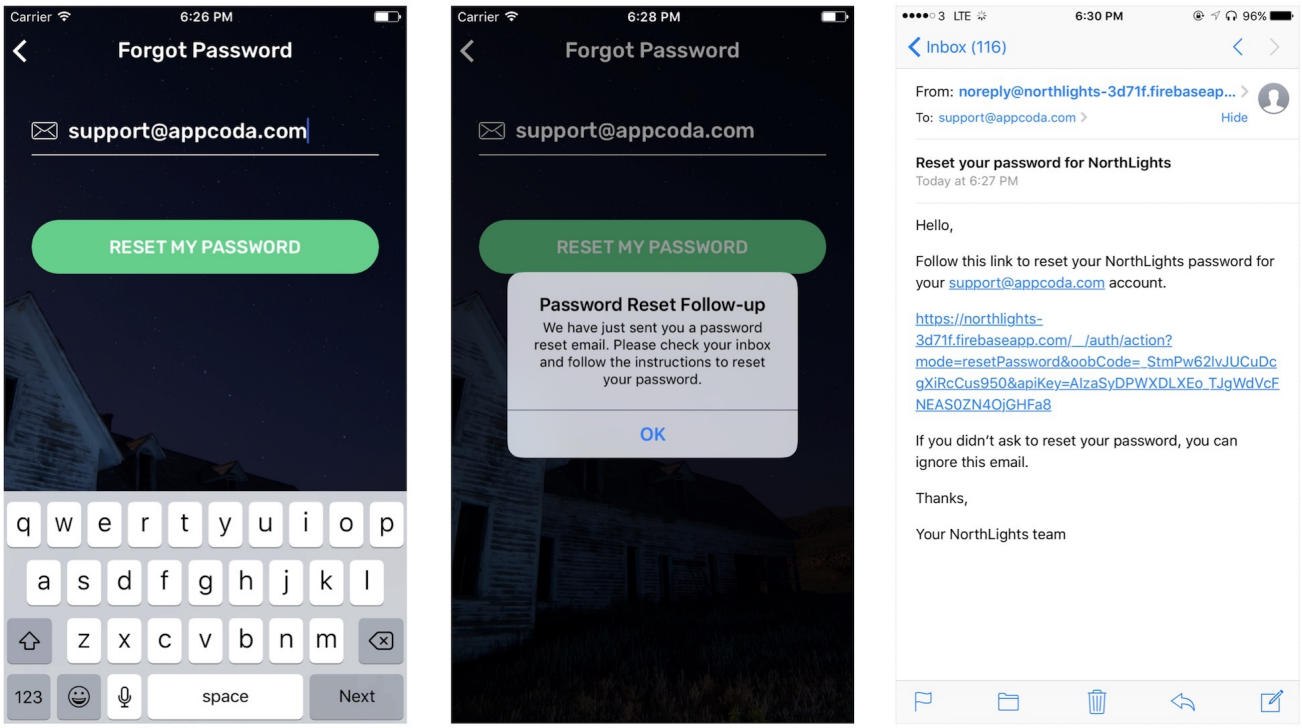


Figure 37.14. Resetting a password

Firebase allows you to customize the content and the *from* email address of the password reset email. You can go up to the Firebase console, and choose *Authentication* > Email Templates to customize the email.

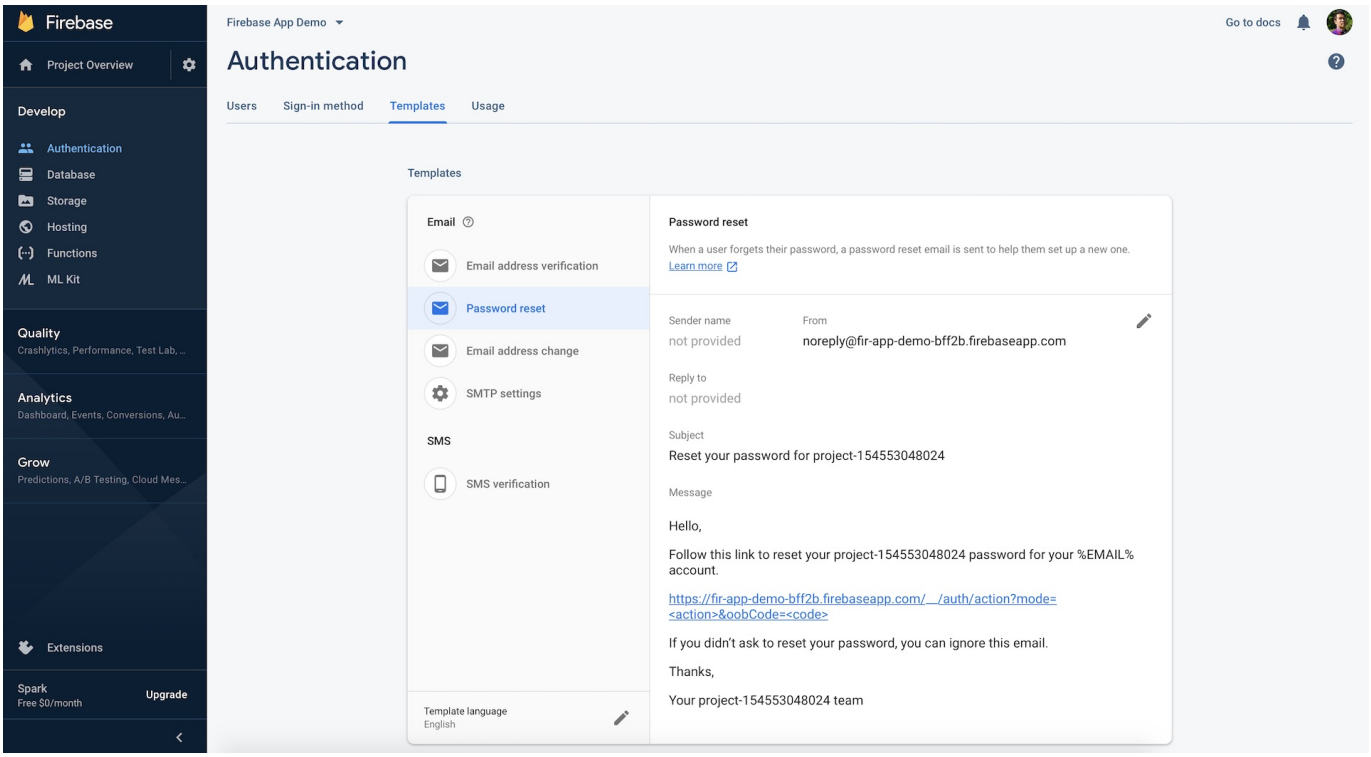


Figure 37.15. Customizing the password reset email

Working with Email Verification

As long as the email address provided by users conforms to the format of an email address, the app accepts it for user registration. What happens if the user provides a fake email address? How can you prevent spam accounts?

One of the popular ways to reduce the number of spam accounts is to implement email verification. After the user signs up using an email address, we send an email with a verification link to that email address. The user can only complete the sign up process by clicking the verification link.

Can we implement this type of verification in our app using Firebase? The answer is absolutely "Yes". Let's see how we can modify the app to support the feature.

If you go to the Firebase console and check out the *Authentication* function. You will find an email template about email address verification under the *Email Templates* tab. Firebase has the email verification built-in, but you have to write some code to enable the

feature.

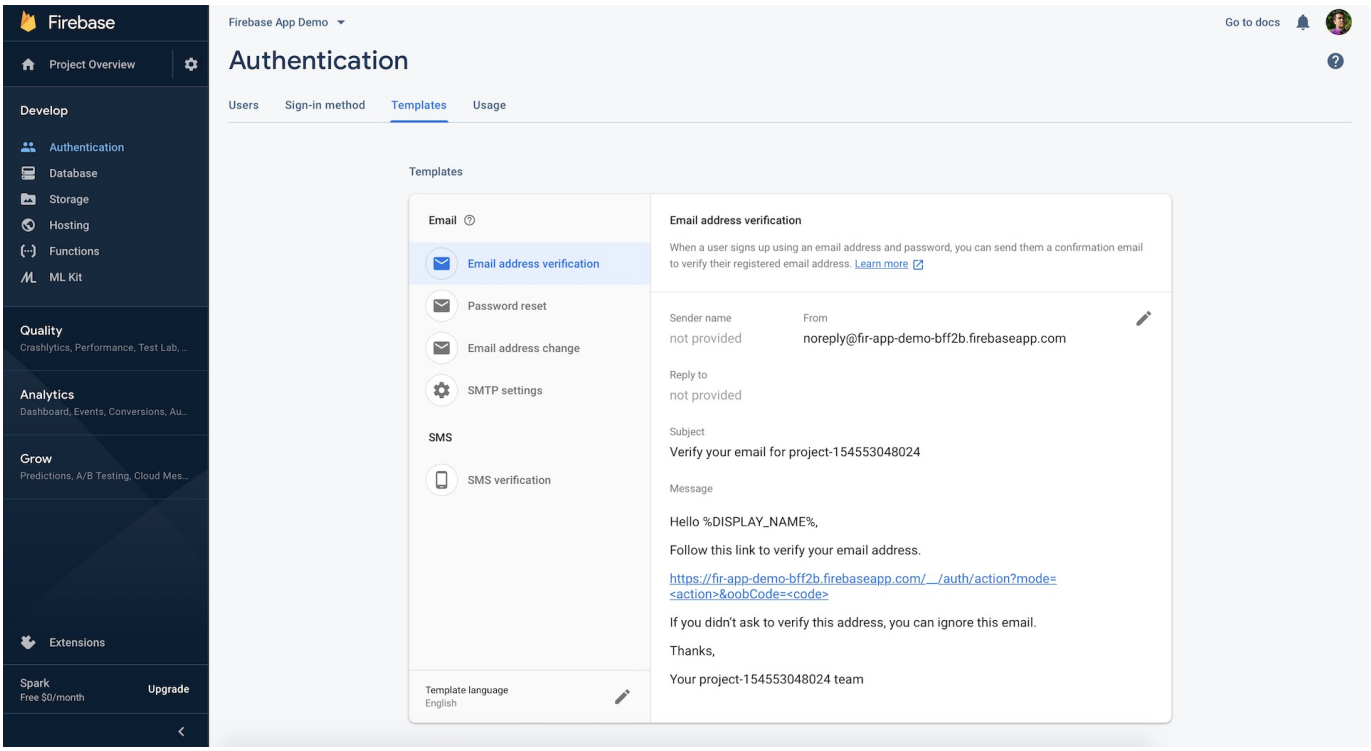


Figure 37.16. Email template for email address verification

Every time when you want/need to know more about an API, the best way is to refer to the official documentation. If you haven't checked out the API documentation, take a look at the description of `User` here:

https://firebase.google.com/docs/reference/ios/firebaseauth/api/reference/Classes/FIRUser

The `User` class specifies that it has a property named `isEmailVerified` that indicates whether the email address associated with the user has been verified. And it has a method called `sendEmailVerification(completion:)` for sending a verification email to the user.

These are the things we need. With the property and the method call, they allow us to implement the email verification feature like this:

- When the user signs up an account, we can call `sendEmailVerification(completion:)` to

send a confirmation email with a verification link. The user has to click the verification link to complete the sign up.

- If the user tries to log into the app without confirming the email address, the app will show an error message telling the user to confirm his/her email address. With the `isEmailVerified` property, we can easily verify the status.
- If the user confirms the email address, he/she will be able to log into the app.

Let's first start with the modification of `SignUpViewController.swift`. As we need to send a verification email after user sign up, we will need to modify the `registerAccount` action method. Update the method like this:

```
@IBAction func registerAccount(sender: UIButton) {

    // Validate the input
    guard let name = nameTextField.text, name != "",
        let emailAddress = emailTextField.text, emailAddress != "",
        let password = passwordTextField.text, password != "" else {

            let alertController = UIAlertController(title: "Registration Error", m
essage: "Please make sure you provide your name, email address and password to com
plete the registration.", preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            present(alertController, animated: true, completion: nil)

            return
    }

    // Register the user account on Firebase
    Auth.auth().createUser(withEmail: emailAddress, password: password, completion
: { (user, error) in

        if let error = error {
            let alertController = UIAlertController(title: "Registration Error", m
essage: error.localizedDescription, preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            self.present(alertController, animated: true, completion: nil)
```

```
            return
        }

        // Save the name of the user
        if let changeRequest = Auth.auth().currentUser?.createProfileChangeRequest
() {
            changeRequest.displayName = name
            changeRequest.commitChanges(completion: { (error) in
                if let error = error {
                    print("Failed to change the display name: \(error.localizedDes
cription)")
                }
            })
        }

        // Dismiss keyboard
        self.view.endEditing(true)

        // Send verification email
        Auth.auth().currentUser?.sendEmailVerification(completion: { (error) in
                print("Failed to send verification email")
            })

        let alertController = UIAlertController(title: "Email Verification", messa
ge: "We've just sent a confirmation email to your email address. Please check your
 inbox and click the verification link in that email to complete the sign up.", pr
eferredStyle: .alert)
        let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: { (ac
tion) in
            // Dismiss the current view controller
            self.dismiss(animated: true, completion: nil)
        })
        alertController.addAction(okayAction)
        self.present(alertController, animated: true, completion: nil)

    })
}
```

Everything before the comment `// Send verification email` is the same as before. Instead of bringing up the home screen (i.e MainView) after creating the user account, we call `sendEmailVerification(completion:)` to send a verification email, and display an on-screen message informing the user to check the inbox.

At this point, the user can't access the home screen of the app. We force the user to go back to the welcome screen. He/she has to confirm their email address, and then log into the app again.

Next, we will need to modify the `LoginViewController.swift` file, which controls the logic of user login. Open the file and update the `login` action method like this:

```
@IBAction func login(sender: UIButton) {

    // Validate the input
    guard let emailAddress = emailTextField.text, emailAddress != "",
        let password = passwordTextField.text, password != "" else {

            let alertController = UIAlertController(title: "Login Error", message:
"Both fields must not be blank.", preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            present(alertController, animated: true, completion: nil)

            return
    }

    // Perform login by calling Firebase APIs
    Auth.auth().signIn(withEmail: emailAddress, password: password, completion: {
(result, error) in

        if let error = error {
            let alertController = UIAlertController(title: "Login Error", message:
 error.localizedDescription, preferredStyle: .alert)
            let okayAction = UIAlertAction(title: "OK", style: .cancel, handler: n
il)
            alertController.addAction(okayAction)
            self.present(alertController, animated: true, completion: nil)
```

```
            return
        }

        // Email verification
        guard let result = result, result.user.isEmailVerified else {
            let alertController = UIAlertController(title: "Login Error", message:
"You haven't confirmed your email address yet. We sent you a confirmation email wh
en you sign up. Please click the verification link in that email. If you need us t
o send the confirmation email again, please tap Resend Email.", preferredStyle: .a
lert)

            let okayAction = UIAlertAction(title: "Resend email", style: .default,
 handler: { (action) in

                Auth.auth().currentUser?.sendEmailVerification(completion: nil)
            })
            let cancelAction = UIAlertAction(title: "Cancel", style: .cancel, hand
ler: nil)
            alertController.addAction(okayAction)
            alertController.addAction(cancelAction)

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

            return
        }


        // Dismiss keyboard
        self.view.endEditing(true)

        // Present the main view
        if let viewController = self.storyboard?.instantiateViewController(withIde
ntifier: "MainView") {
            UIApplication.shared.keyWindow?.rootViewController = viewController
            self.dismiss(animated: true, completion: nil)
        }

    })

}
```

The code is very similar to that we implemented earlier, except that we add several lines of code to check if the email address has been verified or not. If the email is not verified, we will not allow the user to access the home screen or main view.

In the completion handler of the `signIn` method, we perform email verification by checking the `isEmailVerified` property of the current user. If its value is `false` (i.e the email address is not verified), we display an alert and give an option to resend the verification email.

The rest of the code for presenting the main view will only be executed if the user's email address is verified.

After all these changes, it is now ready to test the app again. Try to sign up a new account, and you will receive a confirmation email with a verification link. If you try to log in the app without clicking the verification link, you will end up with an error. But you can log in normally once you verify your email address.

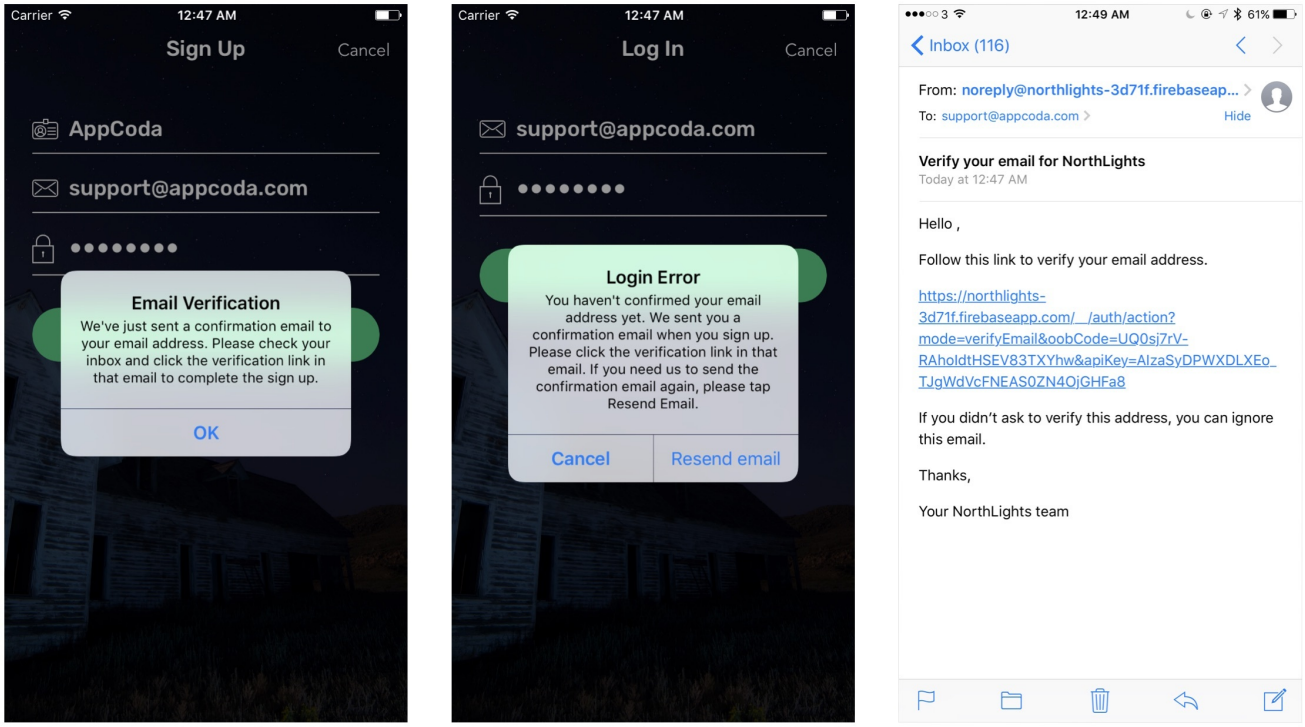


Figure 37.17. The demo app with email verification enabled

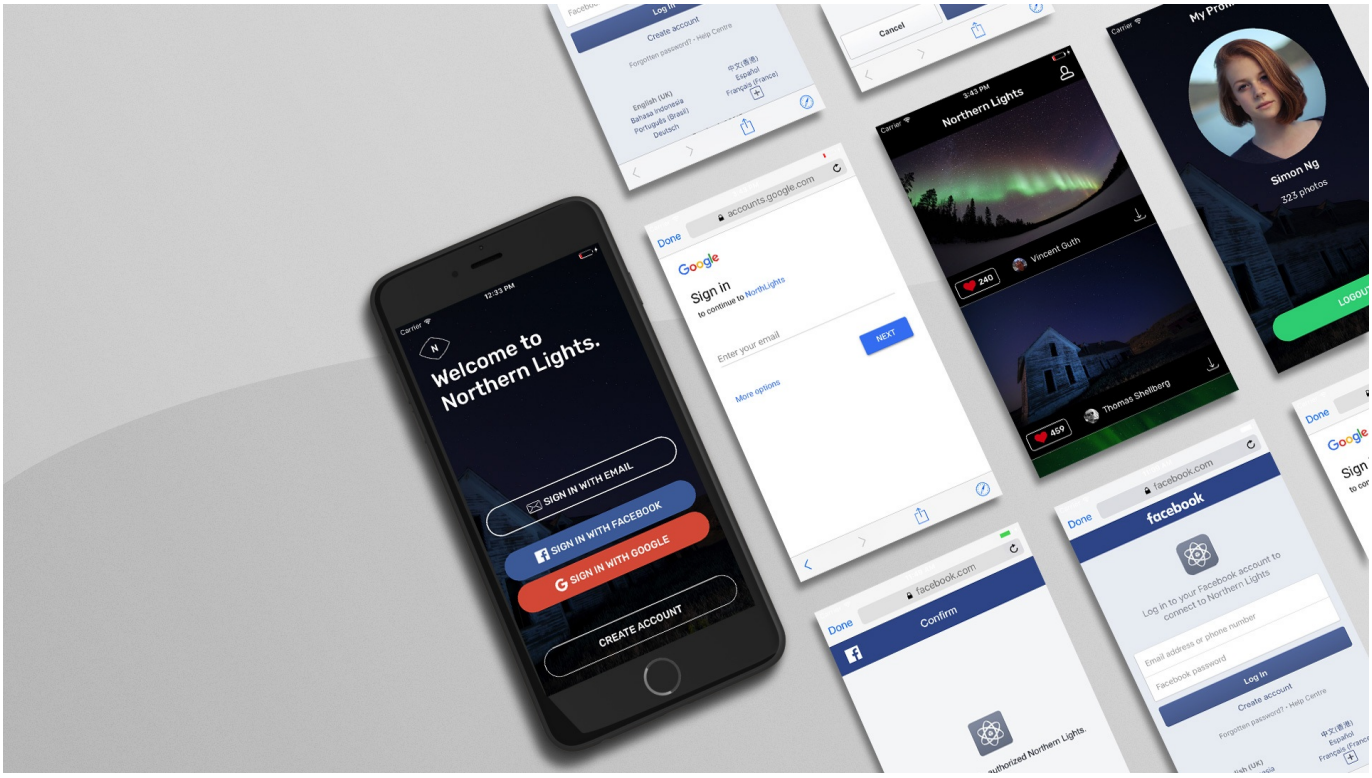
Summary

In this chapter, I walked you through the basics of Firebase. Firebase is now no longer just a database backend. It is a mobile platform that provides a suite of tools (e.g. user authentication) for developers to quickly develop great apps. As you learned in this chapter, you do not need to build your own backend for user authentication or storing user account information. Firebase, along with its SDK, gives everything you need.

Now I believe you understand how to implement sign up, login, logout and password reset using Firebase. If you need to provide user authentication in your apps, you may consider using Firebase as your mobile backend.

For reference, you can download the demo project from http://www.appcoda.com/resources/swift5/FirebaseLoginDemo.zip.

Chapter 38
Google and Facebook Authentication Using Firebase



Previously, I walked you through how to use Firebase for user authentication with email/password. It is very common nowadays for developers to utilize some federated identity provider credentials such as Google Sign-in and Facebook Login, and let users sign up the app with their own Google/Facebook accounts.

In this chapter, we will see how we can use Firebase Authentication to integrate with Facebook and Google Sign in.

Before diving into the implementation, you probably have a question. Why do we need Firebase Authentication? Why not directly use the Facebook SDK and Google SDK to implement user authentication?

Even if you are going to use Firebase Authentication, it doesn't mean you do not need the Facebook/Google SDK. You still need to install the SDK in your Xcode project. However, with Firebase Authentication, most of the time you interact with the Firebase SDK that you are familiar with. Let me give you an example. This is the code snippet for retrieving the user's display name after login:

```
if let currentUser = Auth.auth().currentUser {
    nameLabel.text = currentUser.displayName
}
```

You should be very familiar with the code above if you have read the previous chapter. Now let me ask you. How can you retrieve the user's name for Facebook Login? Probably you will have to go through the Facebook SDK documentation to look up the corresponding API.

Let me tell you if you use Firebase Authentication and perform some integrations with Facebook Login (or Google Sign-in), you can use the same Firebase API (as shown above) to retrieve the user's name from his/her Facebook profile. Does this sound good to you?

This is one of the advantages of using Firebase Authentication to pair with other secure authentication services. And, you can manage all users (whether they use email, Facebook or Google to login) in the same Firebase console.

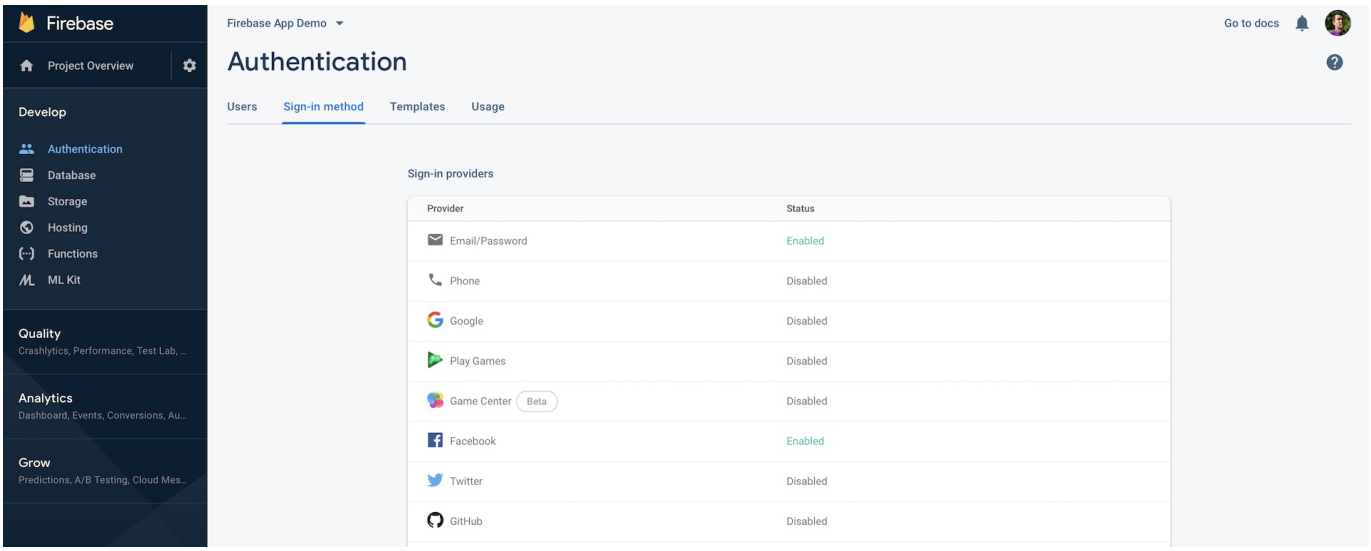


Figure 38.1. The Firebase console lets you enable/disable a certain sign-in method instantly

> **Prerequisite:** You better read Chapter 37 before going through this chapter.

Now let's begin to talk about the implementation. The implementation can be divided into three parts. Say, for Facebook Login, here is what we need to do: * **Configure your Facebook app** - to use Facebook login, you need to create a new app on its developer website, and go through some configurations such as App ID and App Secret. * **Set up Facebook Login in Firebase** - as you are going to use Firebase for Facebook Login, you will need to set up your app in Firebase console. * **Integrate Firebase & Facebook SDK into your Xcode project** - after all the configurations, the last step is to write code to implement Facebook Login through both Facebook and Firebase SDK. That looks complicated. I will say the coding part is fairly straightforward, however, it will take you some time to do the configurations and project setup. Okay, let's get started. ### The Demo Project We will use the final project of the demo app that we built in the previous chapter. If you do not have the final project, you can download the starter project from http://www.appcoda.com/resources/swift5/FirebaseLoginDemo.zip. But please make sure you replace the `GoogleService-Info.plist` file with your own. Please refer to the

previous chapter for the detailed procedures. Previously, we implemented the login function of the demo that allows users to sign up and sign in through email/password.

> **Note:** Once you download the starter project, open FirebaseDemo.xcworkspace and change the bundle identifier from com.appcoda.xxx to your own bundle identifier. This is important because this identifier must be unique.

The Facebook and Google Login buttons were left unimplemented. In this chapter, we will make both buttons functional, and provide alternate ways for users to sign in using their own Facebook/Google account.

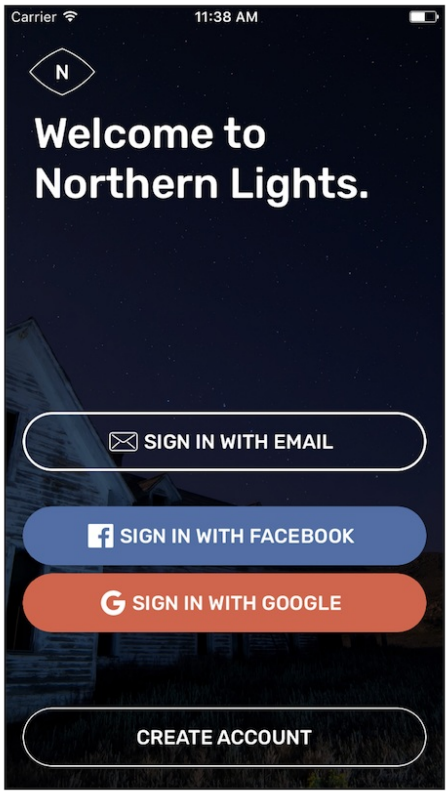


Figure 38.2. The Demo App - Northern Lights

The login process can be very similar to that of email address, and at the same time, quite different from the old-fashioned login approach. Say, when the user taps the *Sign in with Facebook* button for the first time, the app brings up a login screen for users. The login screen is provided by Facebook, and the user is required to sign in with his/her Facebook account. In addition to that, the user has to grant our app privileges to retrieve his/her email address and public profile (e.g. display name). Facebook usually caches the user's login, so the user does not need to sign in for subsequent login.

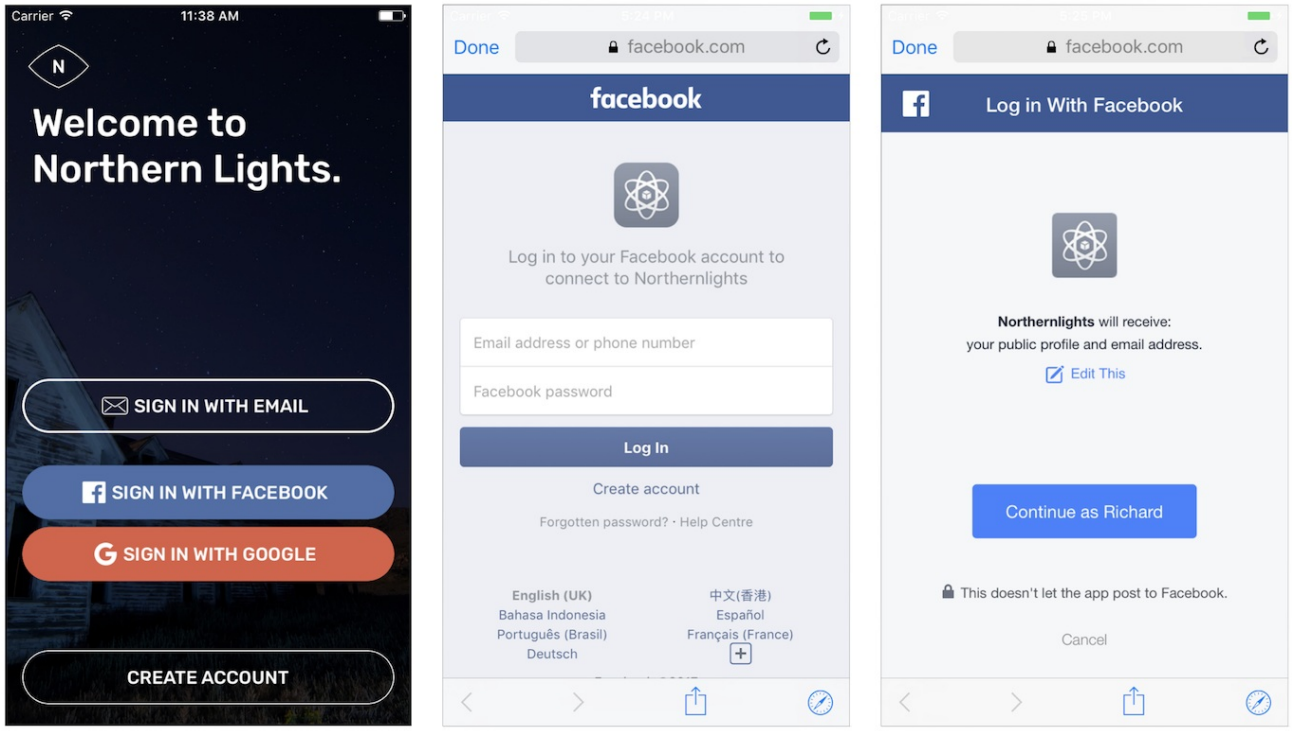


Figure 38.3. User Authentication Process with Facebook Login

Facebook Login

Let's first check out how to implement Facebook Login. In the core section that follows, I will walk you through how to integrate Google Sign In with the demo app.

Setting a New Facebook App

As mentioned before, you will have to go through several configuration procedures. First, we will set up a new Facebook app. Go to Facebook for Developers site (https://developers.facebook.com/), log in with your Facebook account. In the dropdown menu of My Apps, choose *Add a new app*. You will be prompted to give your app a display name and a category. I set the name to `NorthernLights` and choose `Photo` for category.

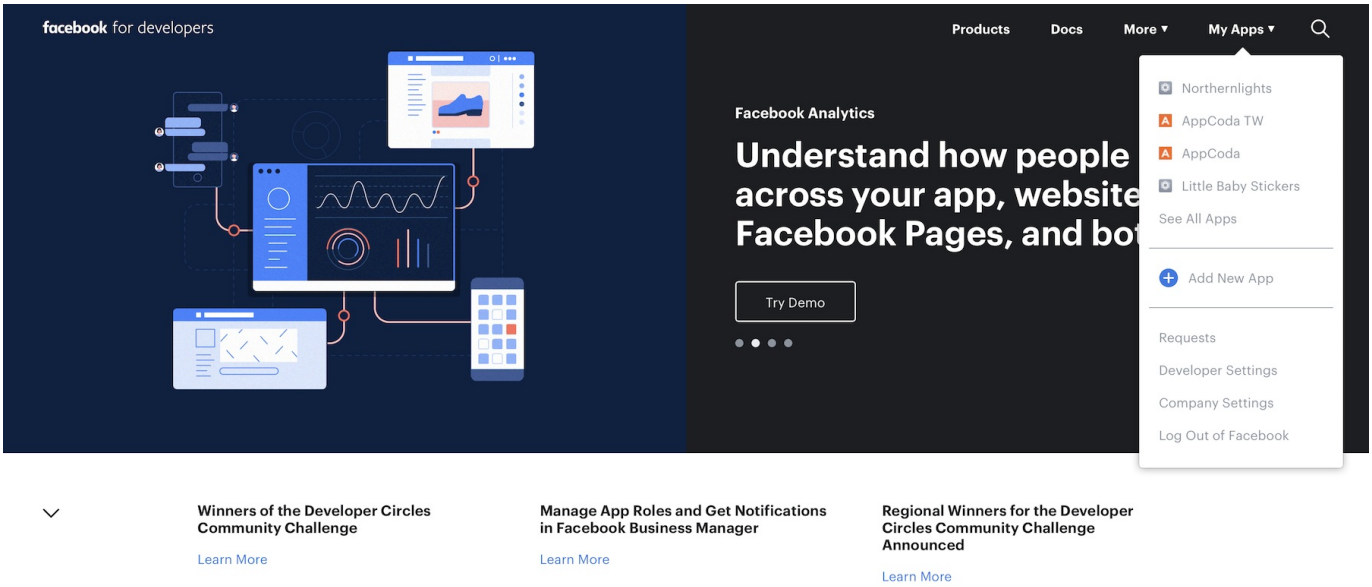


Figure 38.4. Adding a new Facebook app

Next, click `Create App ID` to proceed. You will then be brought to a dashboard that you can configure Facebook Login.

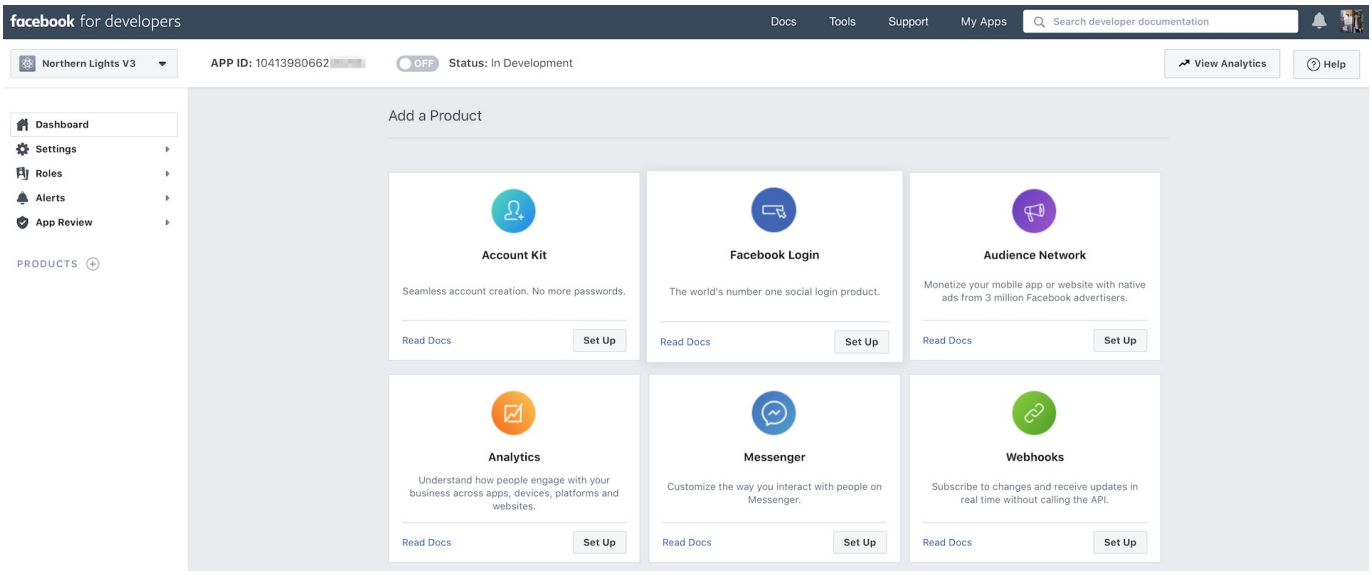


Figure 38.5. Dashboard for your Facebook app

Now click `Setup` of the *Facebook Login* option. Choose `iOS` and Facebook will guide you through the integration process. You can ignore the instructions of Facebook SDK installation. Later, I will show you to use CocoaPods to install it. Just click Continue to proceed to the next step. In step 2 of the configuration, you're required to provide the bundle ID of your project. Set it to the bundle ID you configured earlier and hit `Save` to save the change.

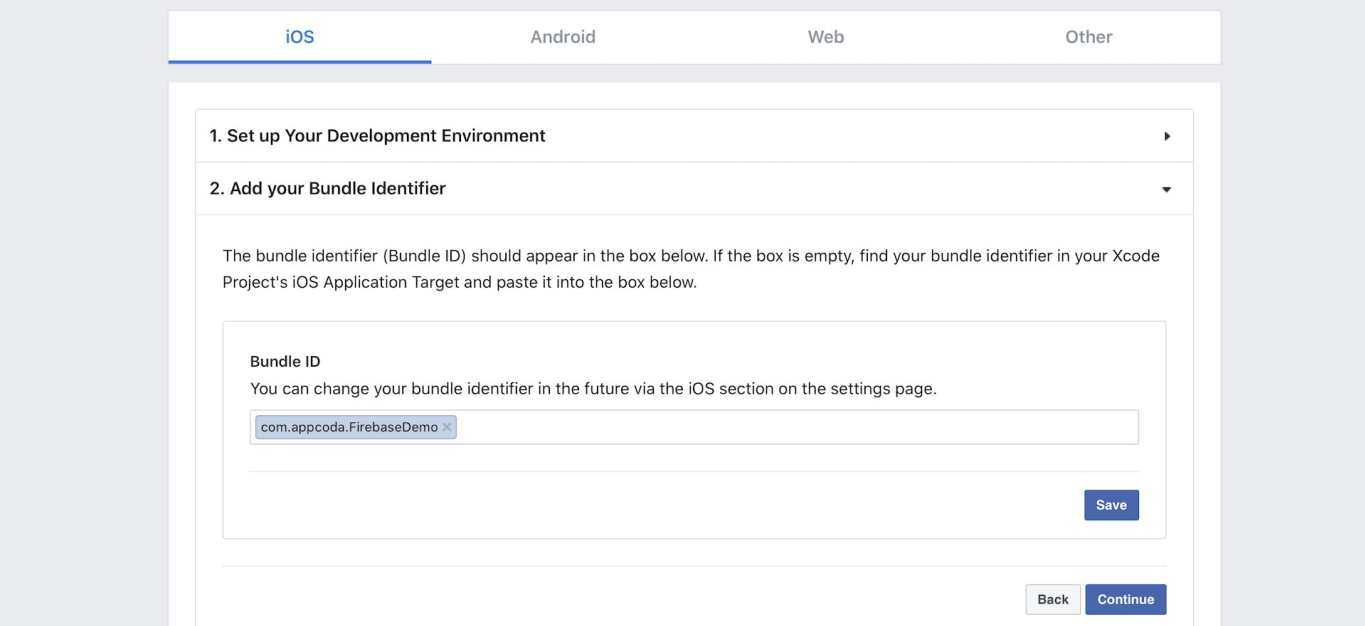


Figure 38.6. Adding your bundle identifier

That's it. You can skip the rest of the procedures. If you want to verify the bundle ID setting, click *Settings* > *Basic* in the side menu. You should see a section about iOS that shows the bundle ID.

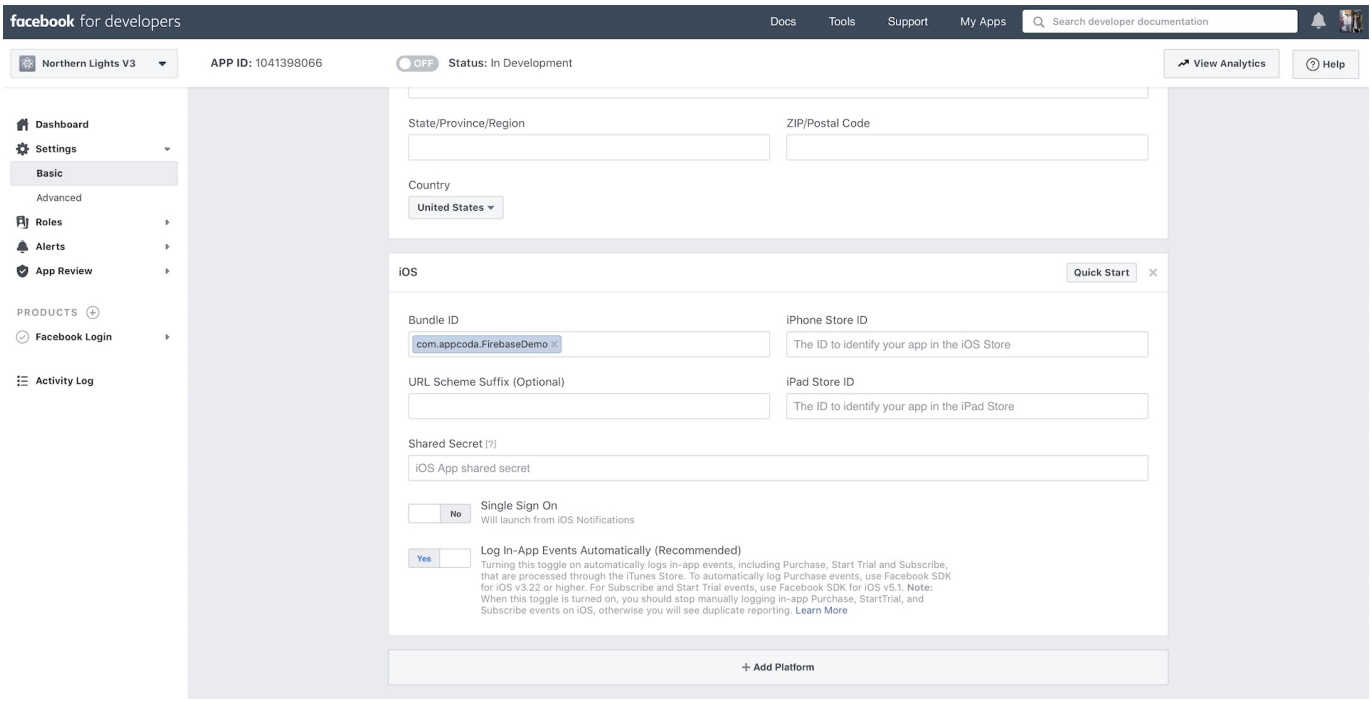


Figure 38.7. You can find the bundle ID in Settings

In the Settings screen, you should find your *App ID* and *App Secret*. By default, *App Secret* is masked. You can click the Show button to reveal it. We need these two pieces of information for Firebase configuration.

Configuring Facebook Login in Firebase

Now head back to the Firebase console (https://console.firebase.google.com) and select the app you created in the previous chapter (e.g. Northern Lights). In the side menu, choose *Authentication* and then select *Sign-in Method*.

> **Note:** If you haven't created an app in Firebase, please refer to the previous chapter for procedures.

Except for the *Email/Password* option, all the rest of the login methods are disabled. As we are now implementing Facebook Login, click *Facebook* and flip the switch to ON. You have to fill in two options here: *App ID* and *App Secret*. These are the values you revealed

in the settings of your Facebook app. Fill them in accordingly and hit `Save` to save the changes.

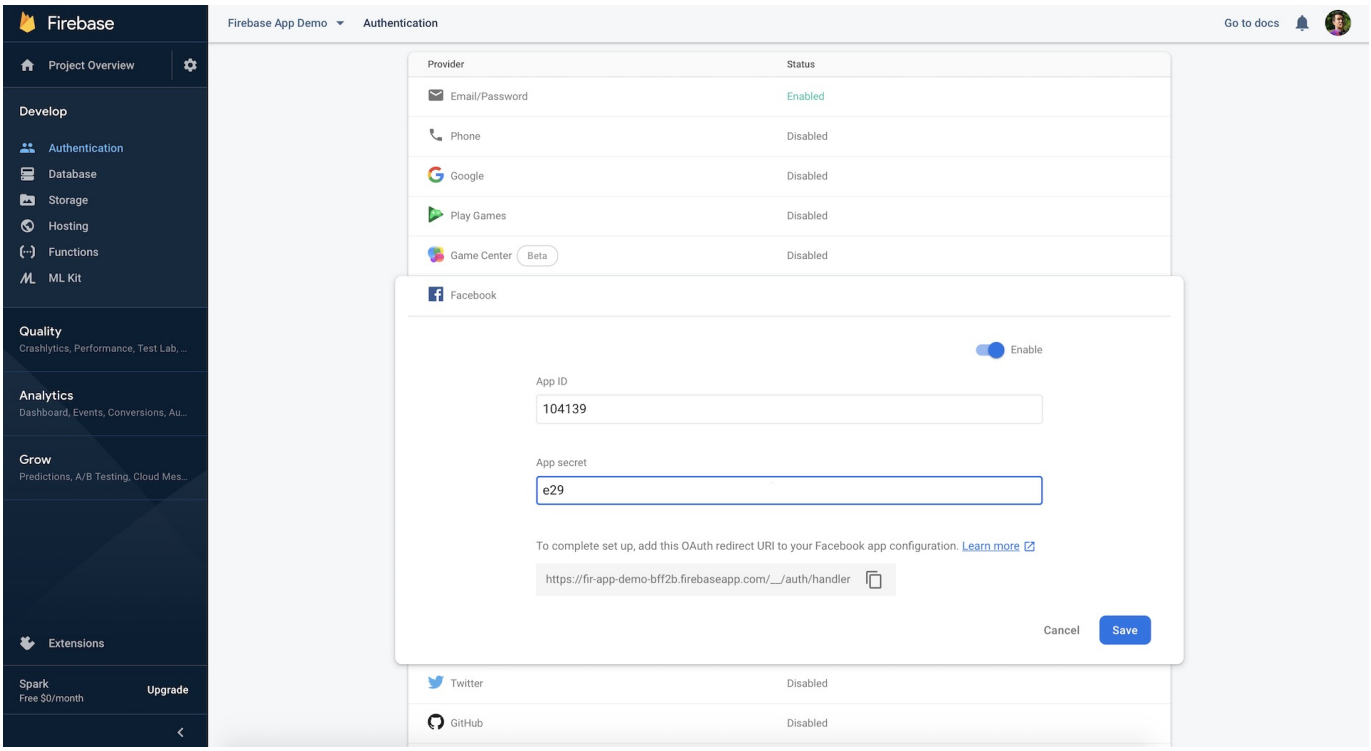


Figure 38.8. Enabling Facebook Login

You may notice the OAuth redirect URI. Google APIs use the OAuth 2.0 protocol for authentication and authorization. After a user logs in with his/her Facebook account and grants the access permissions, Facebook will inform Firebase through a callback URL. Here, the OAuth redirect URI is the URL.

You have to copy this URI and add it to your Facebook app configuration. Now go back to your Facebook app. Under Facebook Login in the side menu, choose *Settings*. Make sure you paste the URI that you copied earlier in the *Valid OAuth redirect URIs* field. Hit *Save Changes* to save the settings.

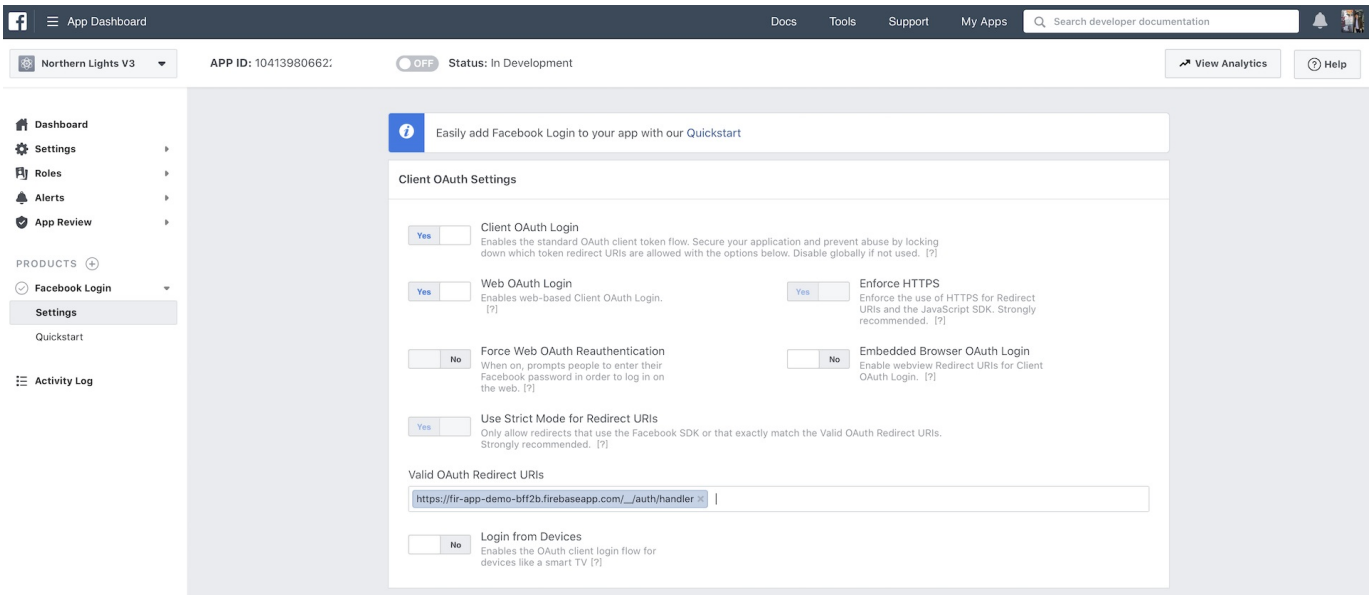


Figure 38.9. Setting the OAuth redirect URI

Great! You have completed the configuration of both your Facebook app and Firebase.

Setting Up the Project with Firebase and Facebook SDK

You will need both the Firebase and Facebook SDK. The easiest way to install both SDKs is by using CocoaPods. To add the SDKs to the project, edit the Podfile like this:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'FirebaseDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for FirebaseDemo
  pod 'Firebase/Core'
  pod 'Firebase/Auth'

  # Pods for Facebook
  pod 'FBSDKLoginKit'
end
```

Note: If you are using the starter project, it already installed with the Firebase SDK. So the Podfile comes with the Core and Auth pods of Firebase.

After the changes, open Terminal and change to the project folder. Then type the following command to install the pods (please make sure you close your Xcode project before running the command):

```
pod install
```

If everything goes smoothly, CocoaPods will download the specified SDKs and bundle them in the Xcode project.

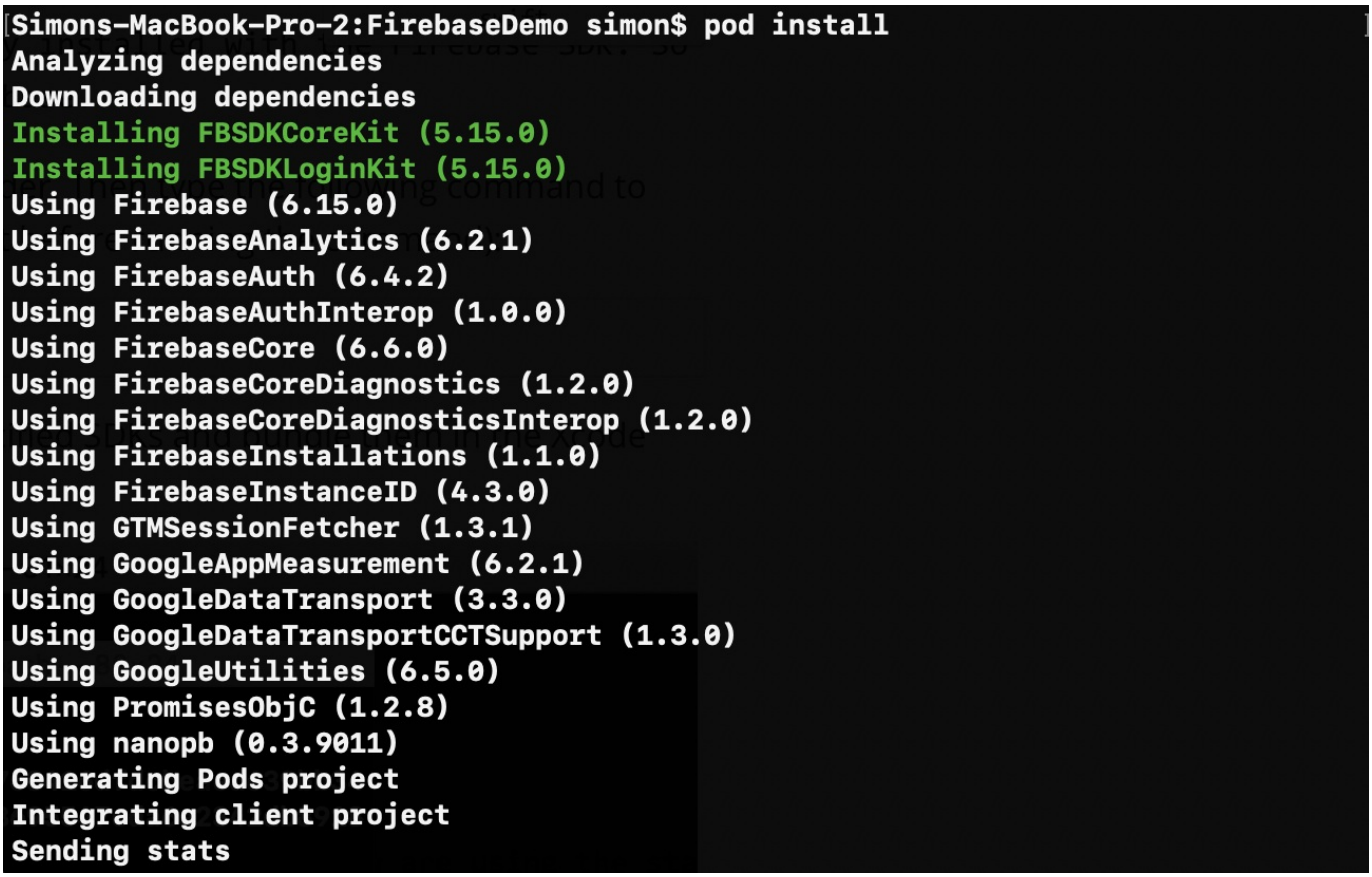


Figure 38.10. Installing the Firebase and Facebook SDKs using CocoaPods

Configuring the Xcode Project

There is one more configuration before we dive into the Swift code. Open `FirebaseDemo.xcworkspace`. In project navigator, right click the `Info.plist` file and choose *Open as* > *Source code*. This will open the file, which is actually an XML file, in text format.

Insert the following XML snippet before the `</dict>` tag:

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
  <key>CFBundleURLSchemes</key>
  <array>
    <string>fb238235556860065</string>
  </array>
  </dict>
</array>
<key>FacebookAppID</key>
<string>238235556860065</string>
<key>FacebookDisplayName</key>
<string>Northern Lights v3</string>
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>fbapi</string>
  <string>fb-messenger-share-api</string>
  <string>fbauth2</string>
  <string>fbshareextension</string>
</array>
```

The snippet above is my own configuration. Yours should be different from mine, so please make the following changes:

- Change the App ID (`238235556860065`) to your own ID. You can reveal this ID in the dashboard of your Facebook app.
- Change `fb238235556860065` to your own URL scheme. Replace it with `fb{your app ID}` .
- Change the display name of the app (i.e. Northern Lights) to your own name.

The Facebook APIs will read the configuration in `Info.plist` for connecting your Facebook app and managing the Facebook Login. You have to ensure the App ID matches the one you created in the earlier section.

The `LSApplicationQueriesSchemes` key specifies the URL schemes your app can use with the `canOpenURL:` method of the `UIApplication` class. If the user has the official Facebook app installed, it may switch to the app for login purpose. In such case, it is required to declare

the required URL schemes in this key, so that Facebook can properly perform the app switch.

Diving into the Swift Code

Finally, after all the cumbersome configurations, it is time to discuss the Swift code. Now open `AppDelegate.swift` and insert the following import statement:

```
import FBSDKCoreKit
```

In the `application(_:didFinishLaunchingWithOptions:)` method, insert a line of code to configure Facebook Login:

```
ApplicationDelegate.shared.application(application, didFinishLaunchingWithOptions:
 launchOptions)
```

In the same class, add the following method:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.Open
URLOptionsKey : Any] = [:]) -> Bool {

    let handled = ApplicationDelegate.shared.application(app, open: url, options:
options)

    return handled
}
```

What is the code above for? `ApplicationDelegate` is designed for post processing results from Facebook Login (e.g. switching over to the native Facebook app) or Facebook Dialogs (similar to the web interface of Facebook). It is required to call the `application` method in order to properly use the Facebook SDK in the `application(_:didFinishLaunchingWithOptions:)` method.

As mentioned before, the Facebook Login process can happen like this:

- The user taps the Sign in with Facebook button.
- Assuming that the user has the native Facebook app installed, the app will switch to the Facebook app to ask for the user's permission.
- After the user's approval, the Facebook app will switch back to our app, passing us the user's credential, and continue the login process.

When the Facebook app switches back to our app, the `application(_:open:options:)` method will be invoked. So we have to implement the method to handle the Facebook Login. To properly handle the login, it is required by Facebook to call the `application` method of `ApplicationDelegate`:

```
ApplicationDelegate.shared.application(app, open: url, options: options)
```

Now open to `WelcomeViewController.swift`. This is the view controller of the Welcome screen that shows the Facebook Login button. In the `WelcomeViewController` class, first add a couple of import statements to import the required SDK:

```
import FBSDKLoginKit
import Firebase
```

Then insert an action method for handling Facebook Login:

```
@IBAction func facebookLogin(sender: UIButton) {
    let fbLoginManager = LoginManager()
    fbLoginManager.logIn(permissions: ["public_profile", "email"], from: self) { (
result, error) in
        if let error = error {
            print("Failed to login: \(error.localizedDescription)")
            return
        }

        guard let accessToken = AccessToken.current else {
            print("Failed to get access token")
            return
        }

        let credential = FacebookAuthProvider.credential(withAccessToken: accessTo
```

```
ken.tokenString)

        // Perform login by calling Firebase APIs
        Auth.auth().signIn(with: credential, completion: { (result, error) in
            if let error = error {
                print("Login error: \(error.localizedDescription)")
                let alertController = UIAlertController(title: "Login Error", mess
age: error.localizedDescription, preferredStyle: .alert)
                let okayAction = UIAlertAction(title: "OK", style: .cancel, handle
r: nil)
                alertController.addAction(okayAction)
                self.present(alertController, animated: true, completion: nil)

                return
            }

            // Present the main view
            if let viewController = self.storyboard?.instantiateViewController(wit
hIdentifier: "MainView") {
                UIApplication.shared.keyWindow?.rootViewController = viewControlle
r
                self.dismiss(animated: true, completion: nil)
            }

        })

    }
}
```

If you've read the previous chapter, it is pretty similar to the code we use when implementing email/password authentication, except the code related to `LoginManager`. The `LoginManager` class provides methods for logging a user in and out. For login, you can call the `logIn` method and specify the read permission you want to ask for. Since we need the email address and the display name of the user, we will ask the user for the read permission of `public_profile` and `email`.

After the user signs in with Facebook, whether he/she grants our app permission or not, the complete handler will be called. Here we first check if there is any error. If not, we proceed to retrieve the access token for the user and convert it to a Firebase credential by

calling:

```
FacebookAuthProvider.credential(withAccessToken: accessToken.tokenString)
```

You should be very familiar with the rest of the code. We call the `signIn` method of `Auth` with the Facebook credential. If there is any error, we present an error dialog. Otherwise, we display the home screen and dismiss the current view controller.

Now switch to `Main.storyboard`, and choose the Welcome View Controller Scene. Control drag from the *Sign In With Facebook* button to the view controller icon of the dock. Release the buttons and select `facebookLoginWithSender:` in the popover menu to connect the action method.

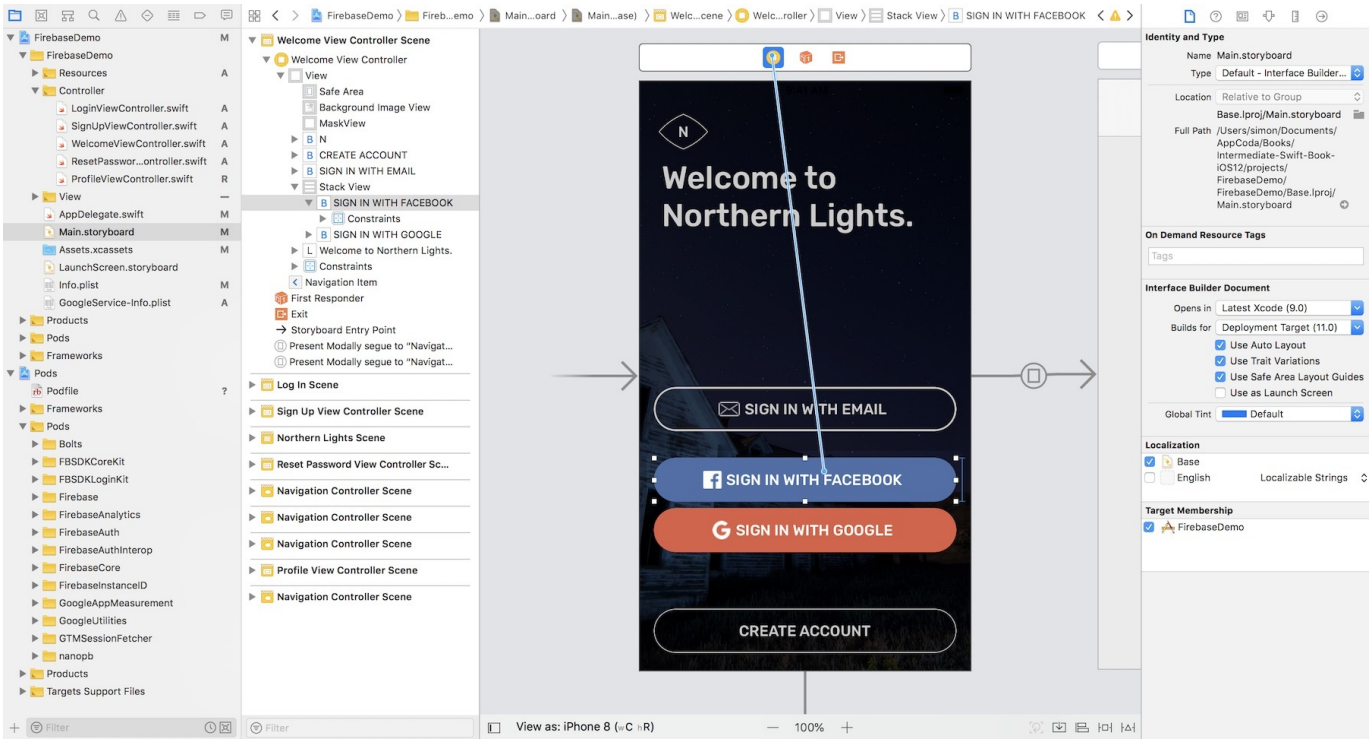


Figure 38.11. Connecting the button with the action method

Setting Up Test Accounts

It's time to test the app. If you tap the *Sign In with Facebook* button, you should see the login screen of Facebook. However, your Facebook app that you configured earlier is still in development mode. If you log into the app with your own Facebook account, you will see an error:

```
App Not Setup: This app is still in development mode, and you don't have access to
 it. Switch to a registered test user or ask an app admin for permissions.
```

Now you have two options to test the app:

1. **Switch your Facebook app to production mode** - you can change your Facebook app from development mode to production in the developer dashboard.
2. **Create a test user (or multiple test accounts) for testing purpose** - Facebook allows developers to create a test account to perform testing when the Facebook app is in development mode.

During the development stage of an app, and especially when authentication is involved in it, it's generally a bad habit to use your normal account to perform all the required testings. Therefore, we will first opt for option 2.

Now go back to the Facebook Developer dashboard (https://developers.facebook.com/apps) and select *Northern Lights*. In the side bar menu, choose Roles > Test Users.

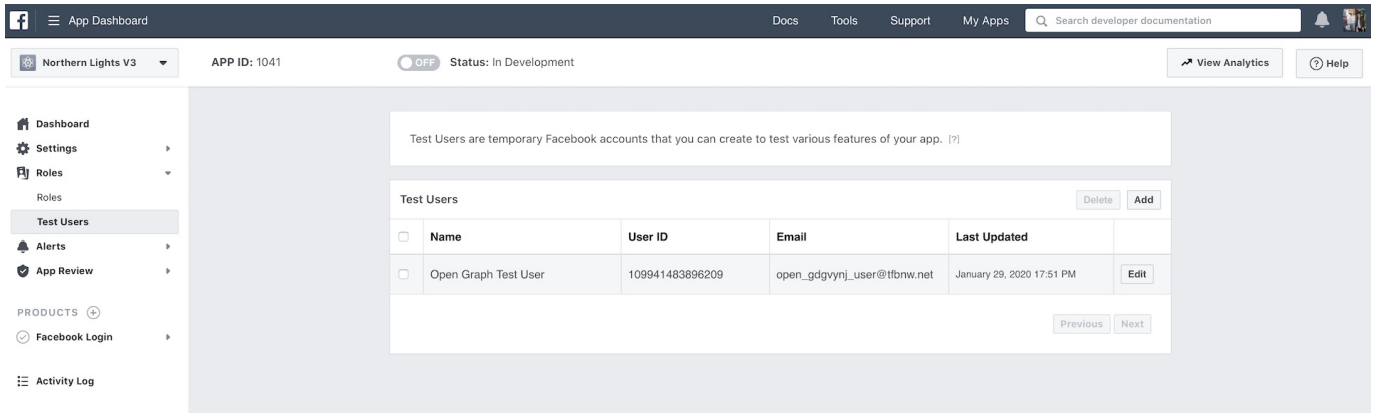


Figure 38.12. Adding a new test user

Here you can click the `Add` button to add a new test user. In the popover menu, set the number of test users to `1` and then hit the `Create Test Users` button. Facebook will then generate a test user with random name and email address. Click the `Edit` button next to the new test user and select *Change the name or password for this test user* to modify its password.

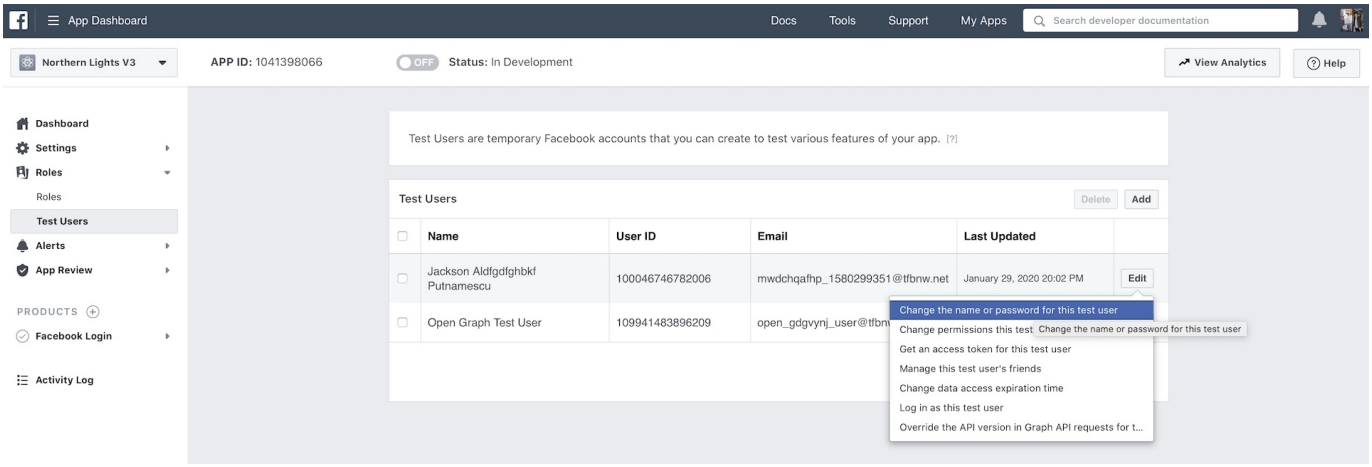


Figure 38.13. Updating the test user's password

Testing Facebook Login

Great! You are now ready to test the app. Run the app and select *Sign In with Facebook* when it launches. Log in with the test account you just created. If everything goes smoothly, you should be able to access the home screen.

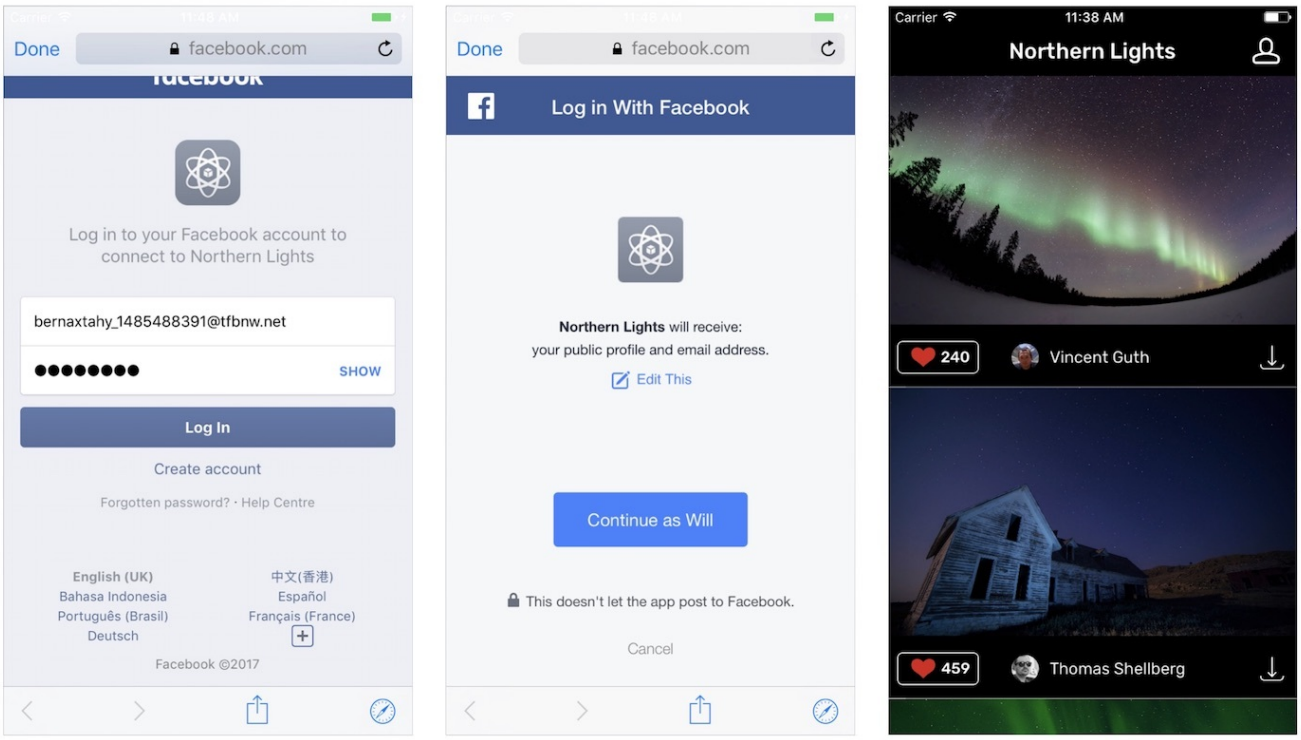


Figure 38.14. Logging into the app with Facebook Login

Facebook will cache your login status. So next time, you no longer need to sign the app with your test account (or Facebook account) again.

If you want to switch to another Facebook user to test, you have to open facebook.com using mobile Safari, and then log out the user. Next time when you log in the app again, it will prompt you to sign in with a legitimate Facebook account.

Logout and User Profile

How about logout and user profile? Do we need to make any code changes for these two features? If you tap the profile icon to bring up the profile screen, you should find that the display name is the same as the test user's name.

That's cool. So far we haven't made any code change for user profile, and the app is already able to retrieve the display name from the user's Facebook profile.

If you look into the `ProfileViewController.swift` file, the following code is responsible for retrieving the display name:

```
if let currentUser = Auth.auth().currentUser {
    nameLabel.text = currentUser.displayName
}
```

Firebase is smart enough to determine the appropriate display name in reference to the login method (whether it is from Facebook or uses email/password).

For logout, we can use the same API to log a user out.

```
Auth.auth().signOut()
```

That is the power of Firebase. You can utilize the same API calls, and Firebase will handle the handle the heavy lifting for you.

Switching to Production

When you finish testing and your app is ready for production, you can go up to the Facebook Developer dashboard. Flip the switch to *ON* to make the app available to the public. Please make sure you provide the privacy policy URL, which is a requirement to make the app public. Once changed, you can now log into the app using your production Facebook account.

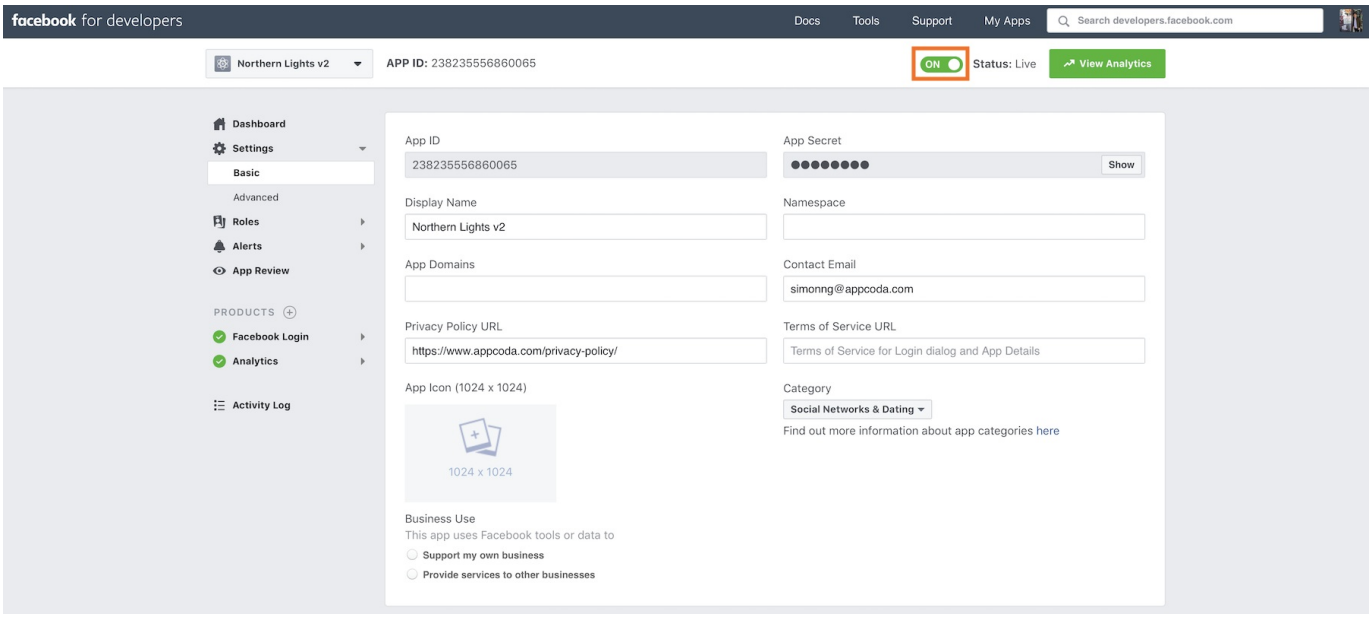


Figure 38.15. Changing the Facebook app to production

Google Sign In

Now that we have implemented the Facebook Login function, let's move onto Google Sign In. The implementation procedures are very similar to that of Facebook Login. But instead of using the Facebook SDK, we have to refer to the Google Sign In documentation and install the Google Sign In SDK. Since Firebase is now a product of Google, it will take you fewer steps to configure Google Sign In. Most of the implementation is related to the integration with Google Sign In SDK.

Okay, let's get started.

Enable Google Sign-In in Firebase Console

As compared to Facebook Login, the configuration is much less cumbersome. All you need to do is go up to your Firebase console of your app. Under *Authentication*, select *Sign-In Method* and then click *Google* to enable Google Sign In. Fill in the name of your project and the support email, and then click *Save* to proceed.

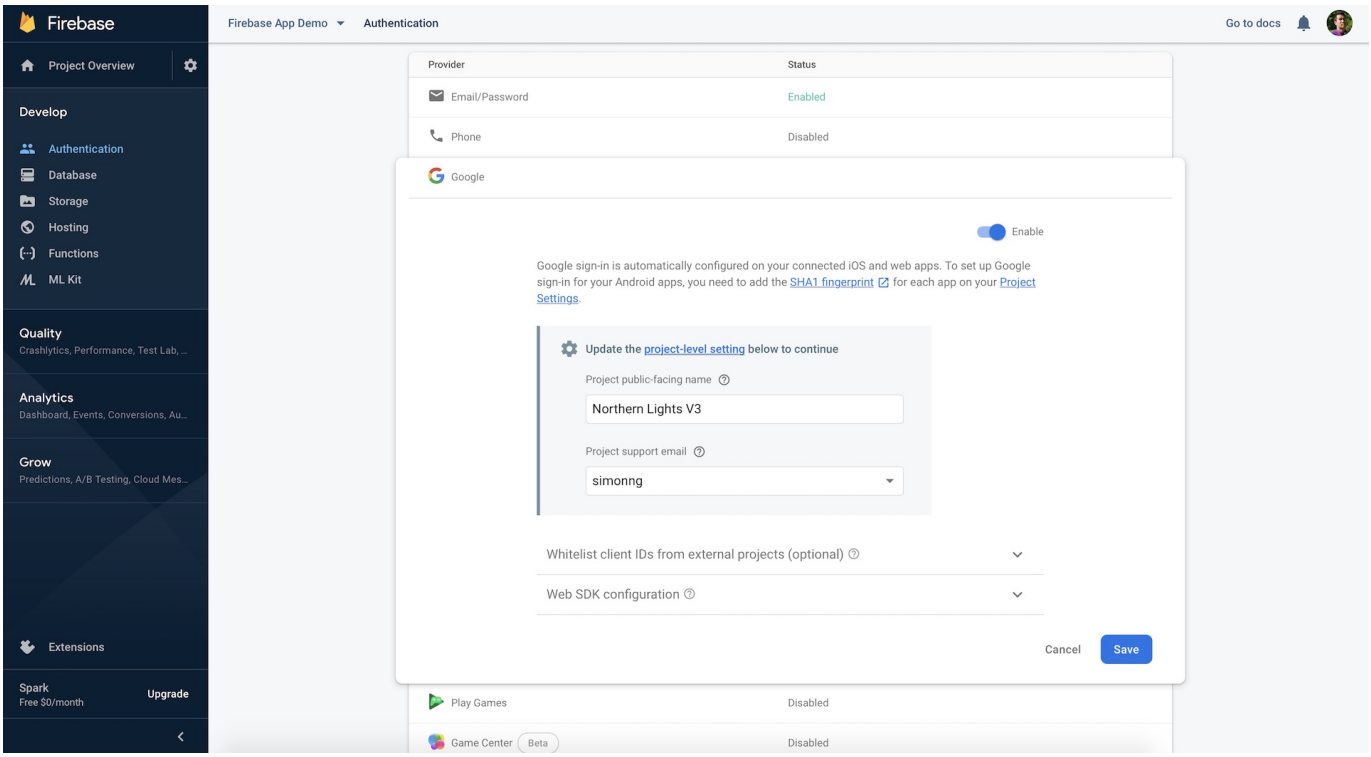


Figure 38.16. Enable Google Sign-In

Installing the Google Sign In SDK

That's it for the Firebase configuration. The next step is to install the Google Sign In SDK in your Xcode project. Again, the simplest way to do that is through CocoaPods.

Now close the Xcode project if you are still opening it. Open Terminal and go to the project folder. Edit `Podfile` like this to add the `GoogleSignIn` pod:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'FirebaseDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for FirebaseDemo
  pod 'Firebase/Core'
  pod 'Firebase/Auth'

  # Pods for Facebook
  pod 'FBSDKLoginKit'

  # Pods for Google Sign In
  pod 'GoogleSignIn'
end
```

Save the file and go back to Terminal. Key in the following command to initiate the pod installation:

```
pod install
```

CocoaPods will download the `GoogleSignIn` SDK and bundle it in the project workspace.

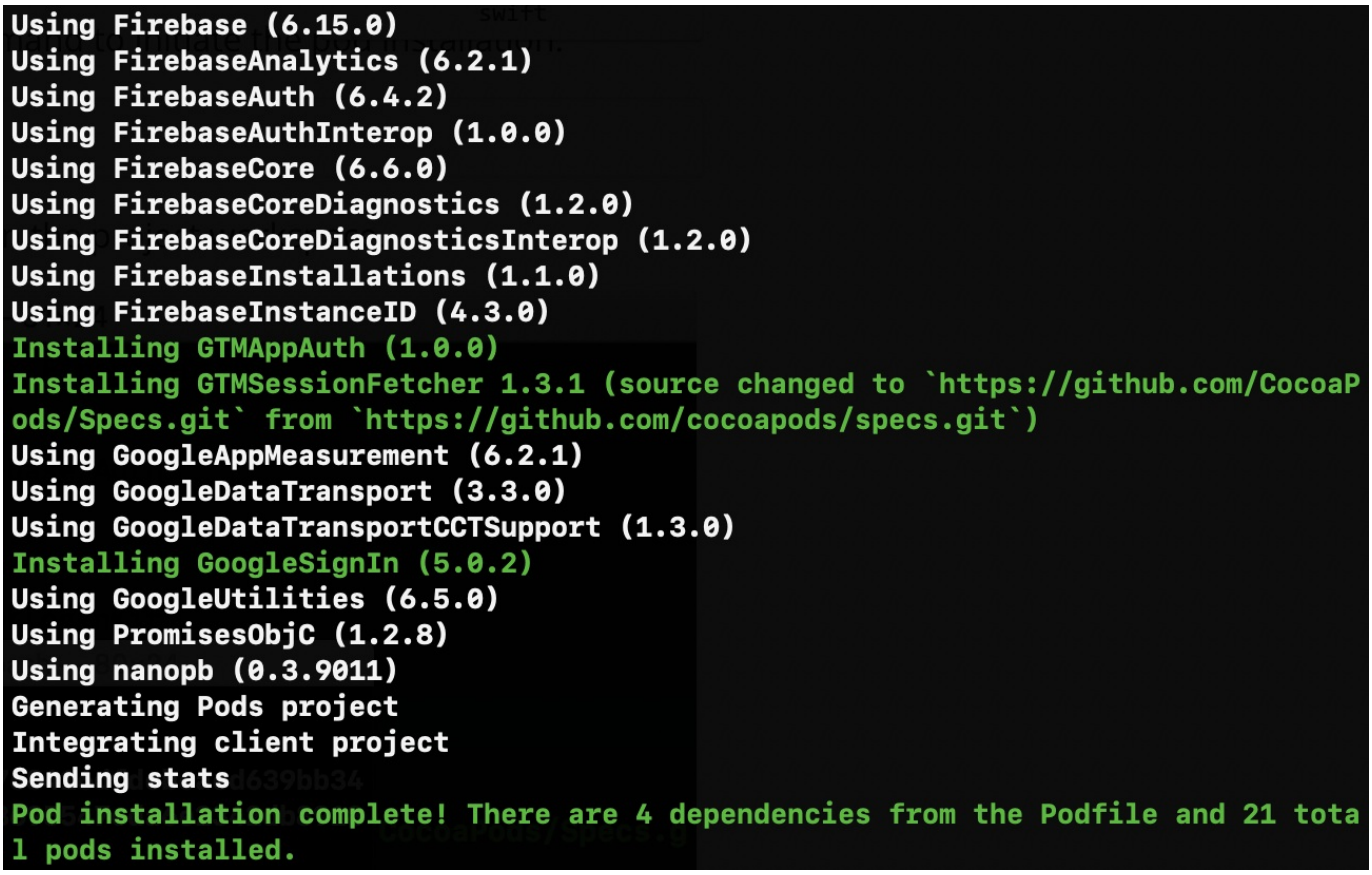


Figure 38.17. Installing the GoogleSignIn SDK using CocoaPods

When finished, open the `FirebaseDemo.xcworkspace` file in Xcode.

Adding Custom URL Schemes

Similar to Facebook Login, Google also performs inter-app communication through custom URL schemes. Therefore, you will need to add a custom URL scheme to the Xcode project. First, select the `GoogleService-Info.plist` file to look for the value of *REVERSED_CLIENT_ID*. This is the URL scheme for your own app. Copy the value first.

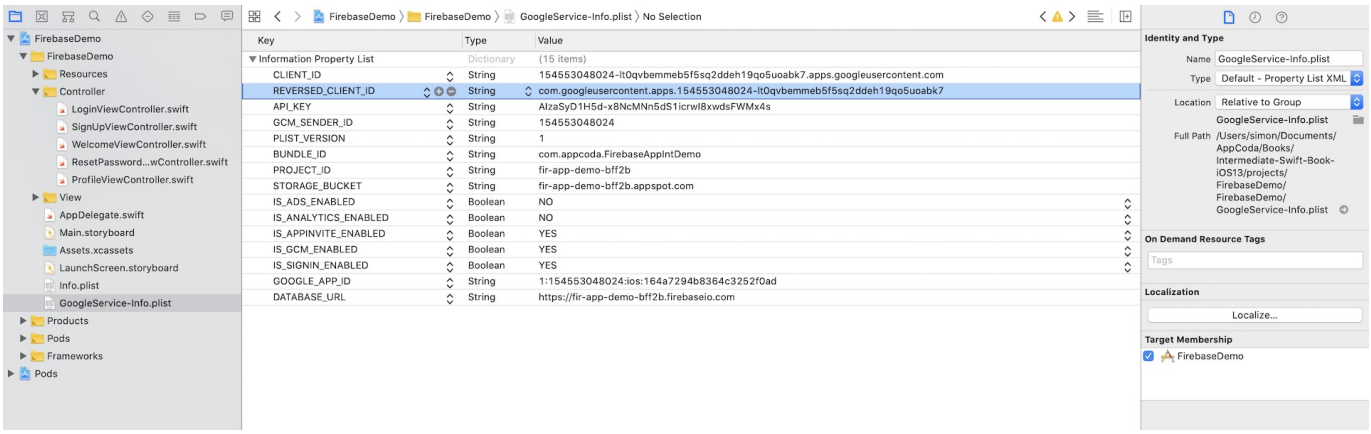


Figure 38.18. Look up the reversed client ID

Now select the `FirebaseDemo` project in the project navigator and then choose `FirebaseDemo` under Targets. Select the *Info* tab and expand the *URL Types* section. Here, you click the `+` icon to add a new custom URL scheme. Paste the value that you copied earlier into the *URL Schemes* field.

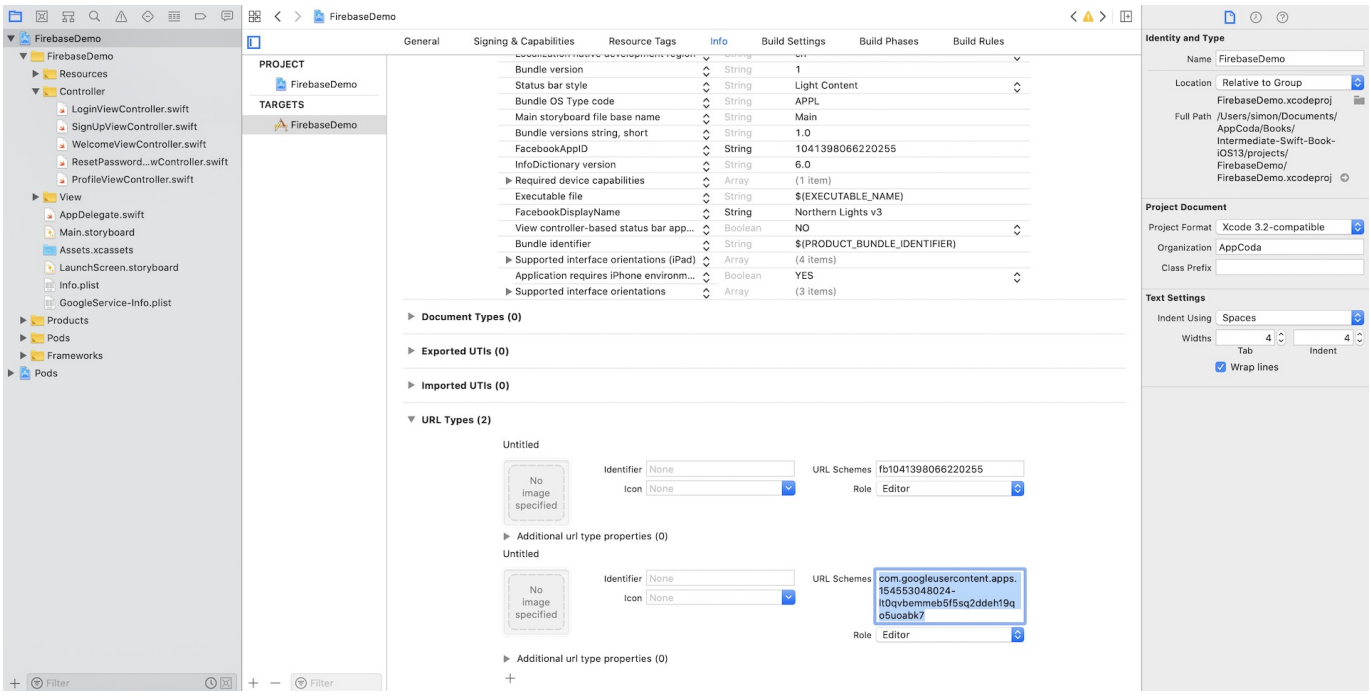


Figure 38.19. Adding a new custom URL scheme

Implementing the Google Sign In APIs

It is time to write some code to implement the Google Sign In function. Now open `AppDelegate.swift` and add an import statement at the very beginning of the file:

```
import GoogleSignIn
```

In order to use the Google Sign In APIs, you have to first import the `GoogleSignIn` package. Then insert a line of code in the `application(_:didFinishLaunchingWithOptions:)` method to initialize the client ID of Google Sign In:

```
GIDSignIn.sharedInstance().clientID = FirebaseApp.app()?.options.clientID
```

Your method should look like this after the change:

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

    // Set up the style and color of the common UI elements
    customizeUIStyle()

    // Configure Firebase
    FirebaseApp.configure()

    // Configure Facebook Login
    ApplicationDelegate.shared.application(application, didFinishLaunchingWithOpti
ons: launchOptions)

    // Configure Google Sign in
    GIDSignIn.sharedInstance().clientID = FirebaseApp.app()?.options.clientID

    return true
}
```

Now your app has to handle two types of URL. One is from Facebook, and the other is from Google. So you have to modify the `application(_:open:options:)` method like this:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.Open
URLOptionsKey : Any] = [:]) -> Bool {

    let handled: Bool

    if url.absoluteString.contains("fb") {
        handled = ApplicationDelegate.shared.application(app, open: url, options:
options)
    } else {
        handled = GIDSignIn.sharedInstance().handle(url)
    }

    return handled
}
```

For Google Sign In, it is required for the method to call the `handleURL` method of the `GIDSignIn` instance, which will properly handle the URL that your application receives at the end of the authentication process.

Let's move onto the implementation of the Google Sign In button. Select `WelcomeViewController.swift` and add the import statement:

```
import GoogleSignIn
```

To implement Google Sign In, you have to adopt the protocol `GIDSignInDelegate`. The `WelcomeViewController` class is the delegate for the protocol. In the `viewDidLoad` method, insert a couple lines to specify the delegate and the presenting view controller:

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

    self.title = ""

    GIDSignIn.sharedInstance().delegate = self
    GIDSignIn.sharedInstance()?.presentingViewController = self
}
```

To adopt both protocols, it is required to implement two methods to handle the sign-in and sign-out processes:

```
func sign(_ signIn: GIDSignIn!, didSignInFor user: GIDGoogleUser!, withError error
: Error!)

func sign(_ signIn: GIDSignIn!, didDisconnectWith user: GIDGoogleUser!, withError
error: Error!)
```

The first method will be called when the sign in process completes, while the latter method is invoked when the user is disconnected from the app.

Now implement both methods using an extension:

```
extension WelcomeViewController: GIDSignInDelegate {
    func sign(_ signIn: GIDSignIn!, didSignInFor user: GIDGoogleUser!, withError e
rror: Error!) {

        if error != nil {
            return
        }

        guard let authentication = user.authentication else {
            return
        }

        let credential = GoogleAuthProvider.credential(withIDToken: authentication
.idToken, accessToken: authentication.accessToken)

        Auth.auth().signIn(with: credential) { (result, error) in
            if let error = error {
                print("Login error: \(error.localizedDescription)")
                let alertController = UIAlertController(title: "Login Error", mess
age: error.localizedDescription, preferredStyle: .alert)
                let okayAction = UIAlertAction(title: "OK", style: .cancel, handle
r: nil)
                alertController.addAction(okayAction)
                self.present(alertController, animated: true, completion: nil)

                return
```

```
            }

            // Present the main view
            if let viewController = self.storyboard?.instantiateViewController(wit
hIdentifier: "MainView") {
                UIApplication.shared.keyWindow?.rootViewController = viewControlle
r
                self.dismiss(animated: true, completion: nil)
            }
        }

    }

    func sign(_ signIn: GIDSignIn!, didDisconnectWith user: GIDGoogleUser!, withEr
ror error: Error!) {

    }
}
```

When the `sign` method is called, we first check if there is any error. If not, we proceed to retrieve the Google ID token and Google access token from the `GIDAuthentication` object (i.e. `user.authentication`). Then we call `GoogleAuthProvider.credential` to exchange them for a Firebase credential which will be used for Firebase authentication. The rest of the code is self-explanatory, and is very similar to those we implemented earlier for Facebook Login.

When the user is disconnected from the app, we do not have any follow up action for this demo. So we just leave the method empty.

Both methods will not be called until we manually trigger the Google Sign In process. To do that, create a new action method called `googleLogin` in the `WelcomeViewController` class:

```
@IBAction func googleLogin(sender: UIButton) {
    GIDSignIn.sharedInstance().signIn()
}
```

This method will be called with the user taps the Google Sign In button. In the method, we simply call the `signIn()` method of `GIDSignIn` to start the sign-in process.

Lastly, go to `Main.storyboard`, and connect the *Sign In with Google* button with the `googleLogin` action method. Control-drag from the button to the view controller icon in the dock. Release both buttons and then choose `googleLoginWithSender:` to connect the method.

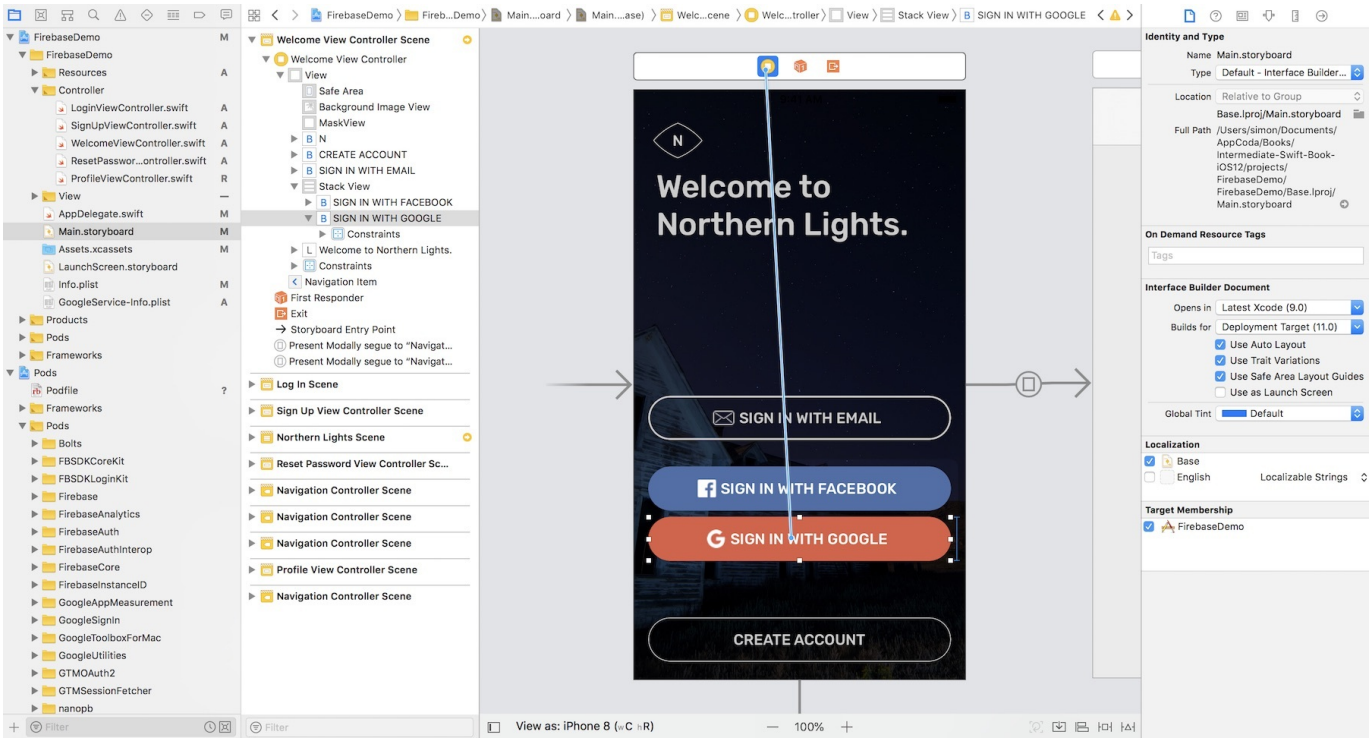


Figure 38.20. Connecting the Sign In With Google button with the action method

Great! It is time to test the Google Sign In function. Run the app on any simulator or your iPhone. Tap the *Sign In with Google* button to initiate the login process. You will see a modal dialog that asks you for a Google account. If everything works smoothly, you should be able to sign in the app with your Google account.

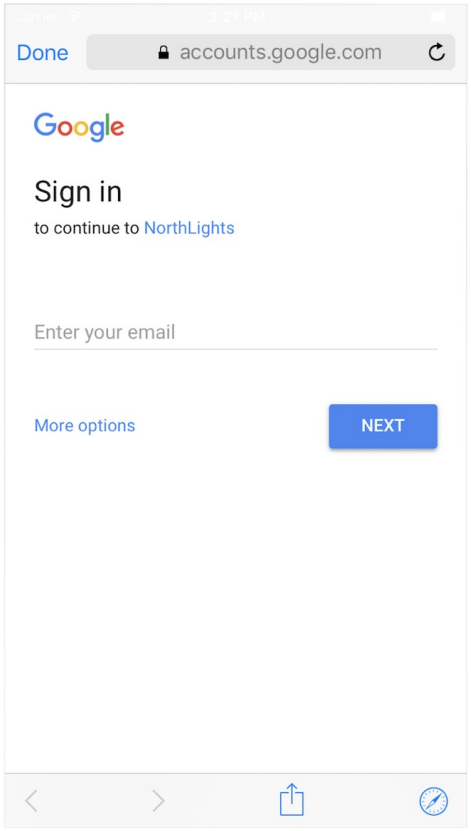


Figure 38.21. Signing in with your Google account

Logout and User Profile

Similarly to Facebook Login, you do not need to modify any code for the `ProfileViewController.swift` file. The logout and user profile features work without any changes. If you tap the *Logout* button of the user profile screen, you should be able to log out successfully.

To provide a seamless and streamlined sign-in flow, users do not need to re-enter their Google credentials to authorize your app for subsequent login. If you try to login the app again with Google Sign In, you will be able to sign into the main screen directly without providing any credential.

However, you may wonder if there is a way to completely sign out the user. For subsequent logins, the user still has to re-enter the Google credential.

To make it work, open `ProfileViewController.swift` and add the following import statement:

```
import GoogleSignIn
```

Modify the `logout` method and insert the code snippet below in the `do` block (before `try Auth.auth().signOut()`):

```
if let providerData = Auth.auth().currentUser?.providerData {
    let userInfo = providerData[0]

    switch userInfo.providerID {
    case "google.com":
        GIDSignIn.sharedInstance().signOut()

    default:
        break
    }
}
```

The `providerData` property allows us to determine the current sign-in method. If the sign-in method is *Google*, we explicitly invoke the `signOut()` of `GIDSignIn` to sign out the user.

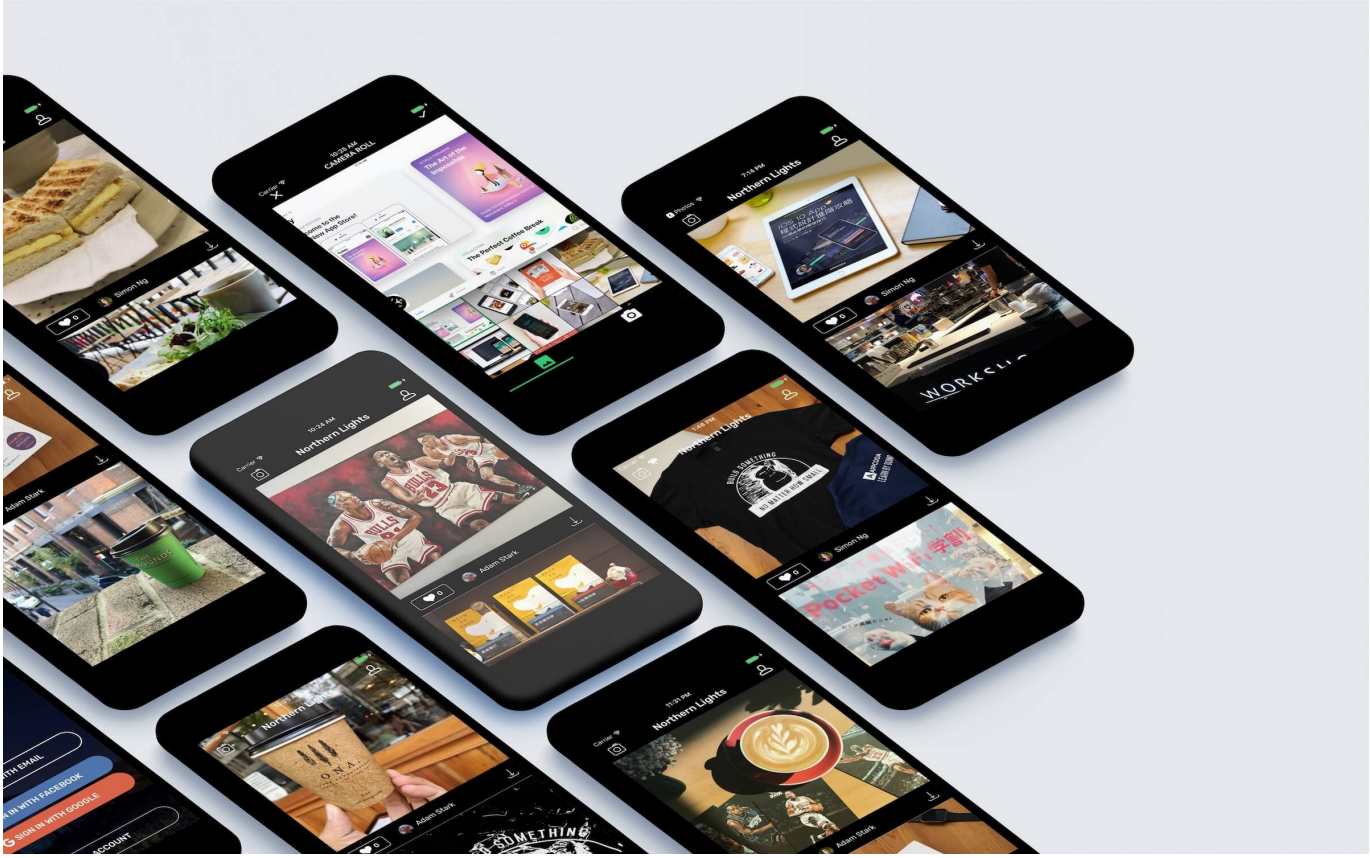
This time, if you test the app again, you're required to enter the Google account every time you sign in the app.

Summary

After going through the demo project, I hope you already understand how to integrate Facebook and Google Sign In using Firebase. These are just two of the many sign-in methods that Firebase supports. Later, if your app needs to support Twitter Sign In (or GitHub), you can also use Firebase to implement the sign-in method using a similar procedure.

For reference, you can download the demo project from http://www.appcoda.com/resources/swift5/FirebaseSocialLoginDemo.zip.

Chapter 39
Using Firebase Database and Storage to Build an Instagram-like App



First things first, I highly recommend you to read the previous two chapters if you haven't done so. Even though most of the chapters are independent, this chapter is tightly related to the other two Firebase chapters.

Assuming you have done that, you should understand how to use Firebase for user authentication. This is just one of the many features of the mobile development platform. In this chapter, we will explore two popular features of Firebase: *Database* and *Storage*. Again, I will walk you through with a project demo. We will build a simple version of Instagram with a public image feed.

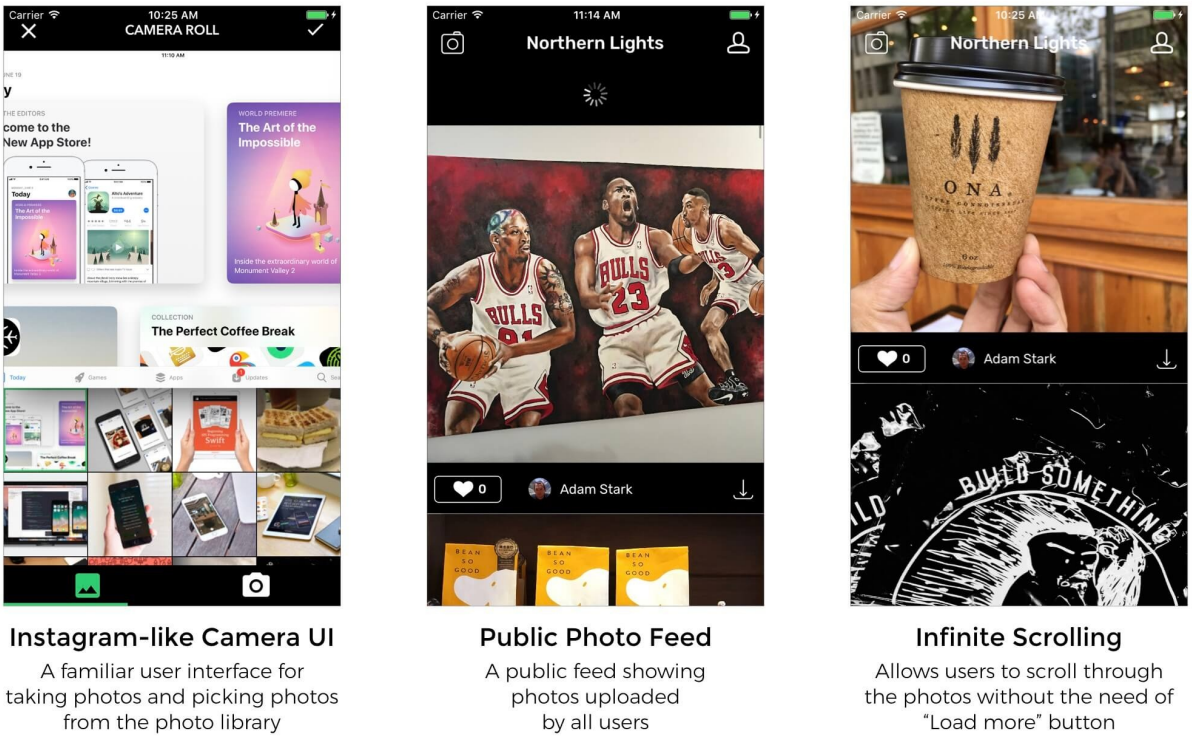


Figure 39.1. The Instagram-like Demo App

Figure 39.1 shows the demo app. As an app user, you can publish photos to the public feed. At the same time, you can view photos uploaded by other users. It is pretty much the same as what Instagram does but we strip off some of the features such as followers.

By building this Instagram-like app, you will learn a ton of new things:

- How to create an Instagram-like camera UI using a third-party library called *ImagePicker*
- How to use Firebase Database to save data (e.g. post information) and structure the JSON data in Firebase database
- How to use Firebase Storage to store images
- How to work with Firebase Database and Storage for data upload and download
- How to limit the number of records retrieved from Firebase Database
- How to implement infinite scrolling in table views

Cool, right? Let's get started.

Setting up the Starter Project

In chapter 37 and 38, you already built a project that supports user authentication through email/password and Google/Facebook login. We will build on top of that project to further implement the photo feed. However, in order to ensure we are on the same page, please download the starter project (http://www.appcoda.com/resources/swift5/FirebaseStorageDemoStarter.zip) before moving on.

The starter project is exactly the same as the one we have worked on previously, except that the table view controller of the home screen is now changed from a static table to a dynamic one.

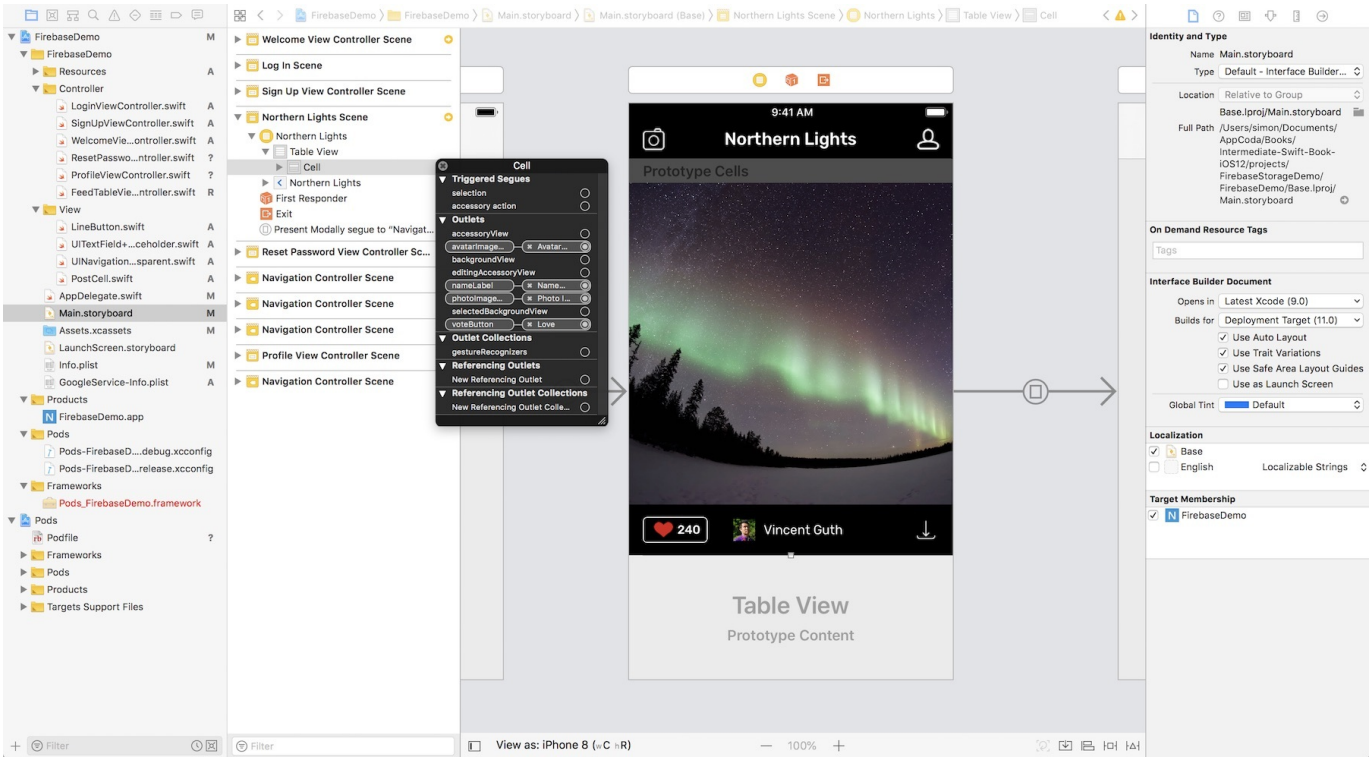


Figure 39.2. The table view controller of the home screen has been changed to dynamic

I have also added two new `.swift` files for the table view controller:

- `FeedTableViewController.swift` - the `FeedTableViewController` class now provides an empty implementation of the table view.

- `PostCell.swift` - the custom class for the table view cell. In the class, you will find a number of outlet variables (e.g. `photoImageView`). Each of the outlets has been connected with the corresponding UI component of the custom cell.

Before moving to the next section, please make sure you replace the `GoogleService-Info.plist` file and bundle identifier of the project with your own. If you forgot the procedures, go back to chapter 37 to check out the details. I also recommend you to test the starter project by building it once. This is to ensure the project can be compiled without errors. When you run the app, you should be able to login with your test accounts (as you configured in chapter 37/38) and access a blank home screen.

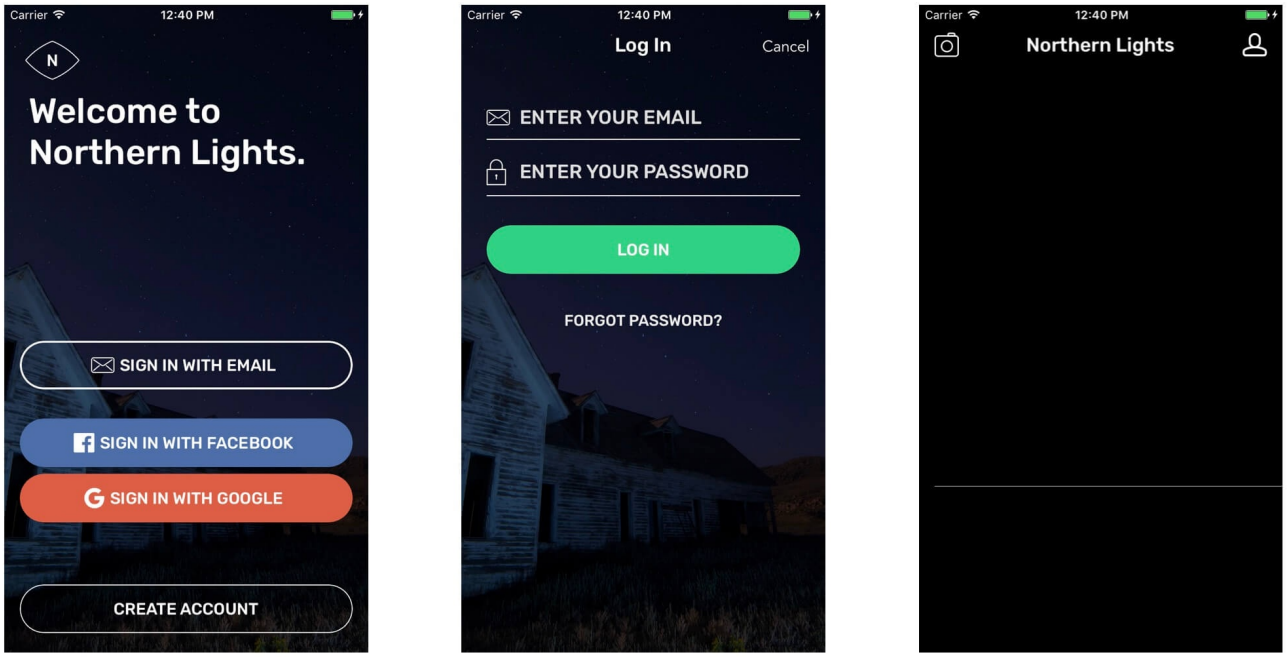


Figure 39.3. Running the starter project

Building the Camera Interface

Now that you have the starter project ready, let's begin with the camera interface before diving into Firebase. When the user taps the *Camera* button, it brings up an Instagram-like camera interface for the user to choose a photo from his/her photo library. Through the same interface, the user can easily take a picture.

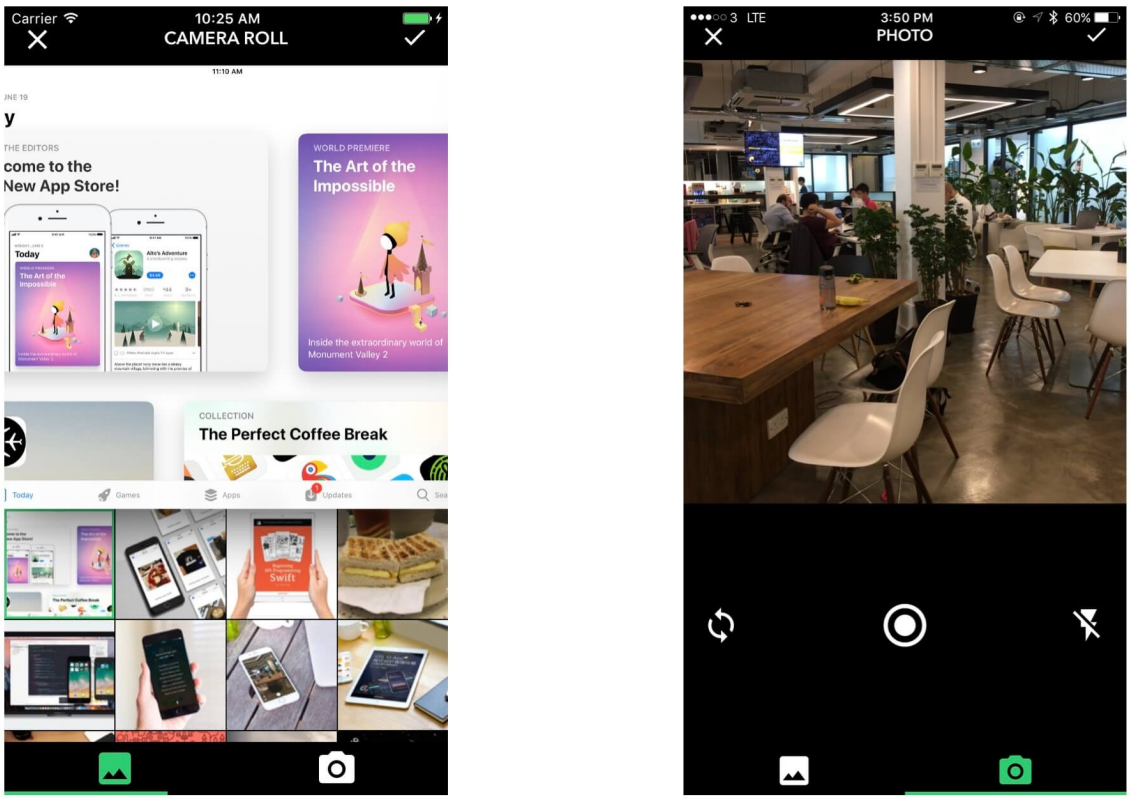


Figure 39.4. An Instagram-like camera UI

The iOS SDK has a built-in class named `UIImagePickerController` for accessing photo library and managing the camera interface. You certainly can use the class to implement the camera feature. However, I want to provide a custom camera UI that users can easily switch between the camera and the photo library. How can we implement that?

1. One option is to build a custom camera from scratch.
2. The other option is to make use of some existing Swift libraries.

In this demo, we will opt for the second approach by using an open source library called *ImagePicker*. You can find the library at https://github.com/hyperoslo/ImagePicker.

Note: There are quite a lot of discussions about whether we should use third-party libraries in Xcode projects lately. Some said we should always write our own code instead of using a third-party library. Here I am not going to discuss this issue, and it's not usually a simple right or wrong answer. It depends on your project background, requirements and other criteria.

ImagePicker is a very easy-to-use library, developed by Hyper, that allows developers to create an Instagram-like photo browser with just a few lines of code.

> ImagePicker is an all-in-one camera solution for your iOS app. It lets your users select images from the library and take pictures at the same time. As a developer you get notified of all the user interactions and get the beautiful UI for free, out of the box, it's just that simple.
>
> - Hyper, developer of ImagePicker

Installing ImagePicker

Like using other third-party libraries, the easiest way to integrate *ImagePicker* into our Xcode project is through CocoaPods.

Note: If you have no ideas about CocoaPods, please refer to chapter 33 for details.

The starter project already comes with a `Podfile`. Make sure you close the Xcode project, and then open `Podfile` with a text editor. Edit it like this:

```
target 'FirebaseDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for Firebase Authentication
  pod 'Firebase/Core'
  pod 'Firebase/Auth'

  # Pods for Facebook
  pod 'FBSDKLoginKit'

  # Pods for Google Sign In
  pod 'GoogleSignIn'

  # Pods for ImagePicker
  pod 'ImagePicker'
end
```

Here we just add the `ImagePicker` pod in the file. Next, go back to terminal and change to the `FirebaseStorageDemo` directory. Type the following command to download and install the library:

```
pod install
```

Once the installation completes, you're ready to open the workspace file (i.e. `FirebaseStorageDemo.xcworkspace`).

Using ImagePicker

ImagePicker is simple to use, as it claimed. You just need to write a few line of code, and you'll have an all-in-one camera in the app. Let's see how it is done.

First, insert an import statement at the beginning of the `FeedTableViewController.swift` file to import the library:

```
import ImagePicker
```

In the `FeedTableViewController` class, create an action method for the camera button:

```
// MARK: - Camera

@IBAction func openCamera(_ sender: Any) {

    let imagePickerController = ImagePickerController()
    imagePickerController.delegate = self
    imagePickerController.imageLimit = 1

    present(imagePickerController, animated: true, completion: nil)

}
```

In brief, here are the main procedures to use ImagePicker:

1. Instantiate an instance of `ImagePickerController`.
2. Define the delegate to perform operations after an image is selected from photo library or taken using the camera.
3. Call the `present` method to bring up the camera UI.
4. Optionally, you can configure the properties of the image picker controller. For example, we set the image limit to `1` so that the user can only select one image for upload.

As you can see, it just takes a few lines of code to create the Instagram-like camera interface.

`ImagePicker` has defined a protocol called `ImagePickerDelegate`. For the class that adopts the protocol, it should implement the following required methods:

- `func doneButtonDidPress(_ imagePicker: ImagePickerController, images: [UIImage])` - it is called when the user taps the Done button.
- `func cancelButtonDidPress(_ imagePicker: ImagePickerController)` - it is called when the user taps the Cancel button.

- `func wrapperDidPress(_ imagePicker: ImagePickerController, images: [UIImage])` - it is called when the user taps the stack button.

Since we have defined `self` as the delegate, we have to adopt all these methods in the `FeedTableViewController` class. To better structure the code, I used to put the code related to a protocol adoption in its own extension. Insert the following code snippet in the `FeedTableViewController.swift` file:

```
// MARK: - ImagePicker Delegate

extension FeedTableViewController: ImagePickerDelegate {

    func wrapperDidPress(_ imagePicker: ImagePickerController, images: [UIImage])
{

    }

    func doneButtonDidPress(_ imagePicker: ImagePickerController, images: [UIImage
]) {
        dismiss(animated: true, completion: nil)
    }

    func cancelButtonDidPress(_ imagePicker: ImagePickerController) {
        dismiss(animated: true, completion: nil)
    }

}
```

For the `wrapperDidPress` method, we just provide an empty implementation because we do not need to perform any operation after the stack button is tapped. For the rest of the methods, we just added a call to dismiss the camera interface. We will leave the actual implementation in the Firebase section.

Editing Info.plist

In iOS 10 (or later), Apple requires every app to ask for user permission before accessing the user's photo library or the device's camera. You have to add two keys in `Info.plist` to explain why the app needs to use the camera and photo library.

Open `Info.plist` and add two new rows:

- Set one's key to "Privacy - Camera Usage Description", and its value to "for capturing photos" (or whatever you want).
- Set another key to "Privacy - Photo Library Usage Description", and its value to "for you to choose photos to upload".

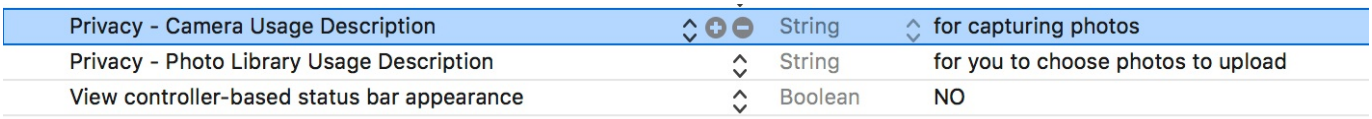


Figure 39.5. Adding two keys in Info.plist

Connect the Action Method

Lastly, switch to `Main.storyboard` and connect the *camera* button with the `openCamera` method we just created.

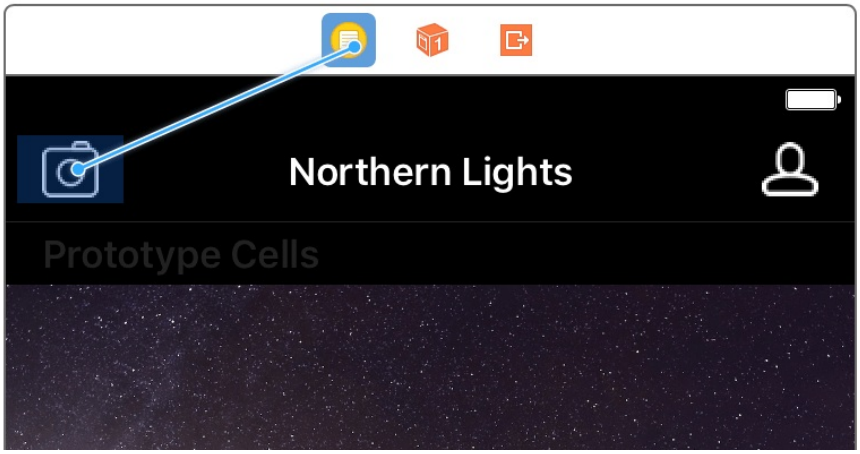


Figure 39.6. Connecting the camera button with the action method

If you build the project now, you will end up with multiple errors when compiling the *ImagePicker* pod. At the time of this writing, the ImagePicker library has not yet updated for Swift 5. It only supports version 4 of the programming language. Since our FirebaseDemo project is set to use Swift 5.1, Xcode is default to use 5.1 to compile all the pod libraries. Xcode 11 allows you to use a different version of Swift for a specific target. In order to compile the *ImagePicker* library, we have to modify its Swift version to 4.0.

In the project navigator, choose the *Pods* project and then select the *ImagePicker* target. Under *Build Settings*, look for the *Swift Language Version* option and change its value to *Swift 4*.

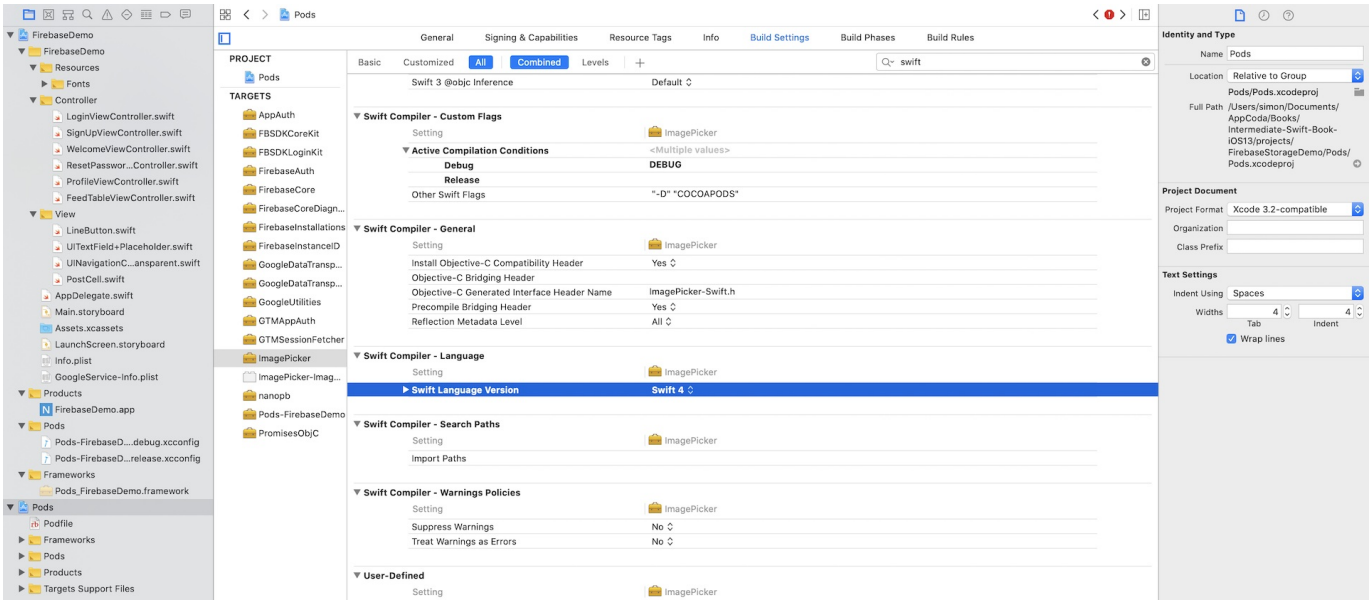


Figure 39.7. Changing the Swift version for ImagePicker

Now build and run the app. Log into the app and then tap the camera button. You should be able to bring up the photo library/camera. For first time use, the app should prompt you for permission. If you choose to disallow the access, the app will show you an alert requesting you to grant the access in Settings.

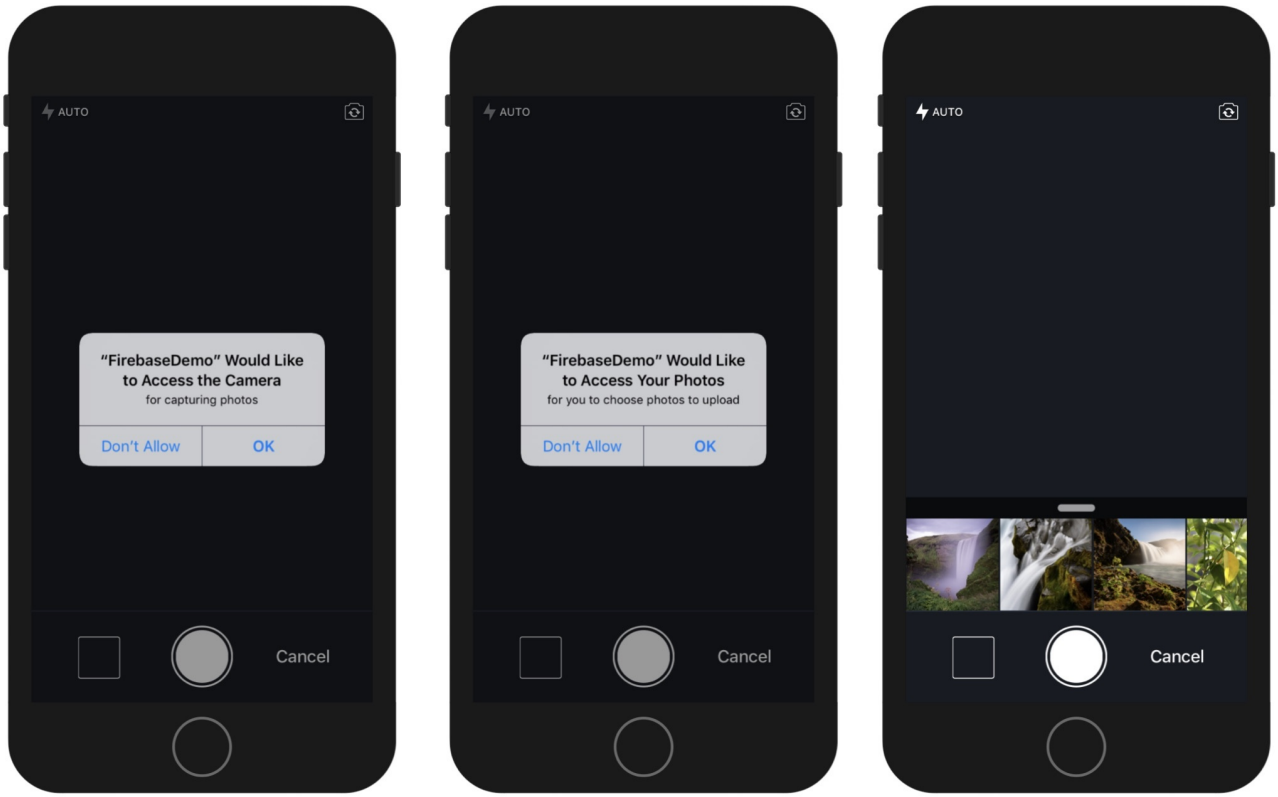


Figure 39.8. Testing the camera feature

A Quick Overview of Firebase Database

With the camera UI ready, it comes to the core part of this chapter. Let's talk about the two main features of Firebase: *Database* and *Storage*.

When Firebase was first started, its core product was a realtime database. Since Google acquired Firebase in late 2014, Firebase was gradually rebranded as a mobile application platform that offers a suite of development tools and backend services such as notification, analytics, and user authentication. Realtime Database and Storage are now just two core products of Firebase.

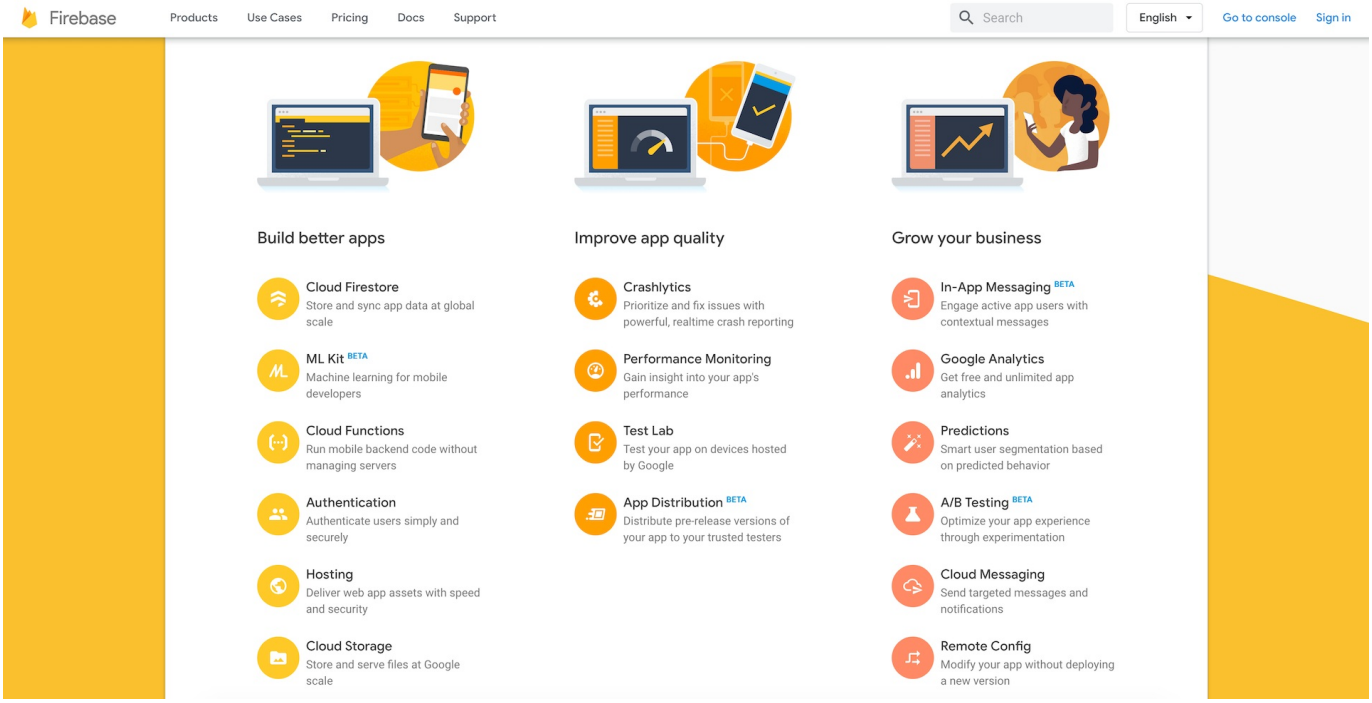


Figure 39.9. Firebase Products

In this section, we will dive into these two Firebase products, and see how to build a cloud-based app using Realtime Database and Storage.

Understanding how Data is Structured in Firebase

So, what's Firebase Realtime Database? Here is the official description from Google:

> The Firebase Realtime Database is a cloud-hosted NoSQL database that lets you store and sync data between your users in realtime.

You may know Parse or CloudKit. Both are mobile backend that lets you easily store and manage application data in the cloud. Firebase is similar to this kind of backend services. However, the way how the data is structured or stored is totally different.

If you have some database background, it is quite straightforward for you to understand the data structure of Parse or CloudKit. You create a table with multiple columns, and each record is a row in the table. Consider if you are going to store the photo posts of the

demo using Parse, you will probably create a `Post` table with columns like *user*, *image* and *votes*.

Figure 39.10. A Post class in Parse

The data representation is quite intuitive, even for beginners, because it is similar to an Excel table.

However, Firebase Realtime Database doesn't store data like that. As the company said, Firebase Database is a cloud-hosted NoSQL database. The approach to data management of NoSQL database is completely different from that of traditional relational database management systems (RDBMS).

Unlike SQL database, NoSQL database (or Firebase Database) has no tables or records. Data is stored as JSON in key-value pairs. You don't have to create the table structure (i.e. columns) before you're allowed to add the records since NoSQL does not have the *record* concept. You can save the data in key-value pairs in the JSON tree at any time.

Further reading: I will not go into the details of NoSQL database. If you are interested in learning more about NoSQL, please take a look at these articles:

- NoSQL Databases: An Overview (https://www.thoughtworks.com/insights/blog/nosql-databases-overview)

- NoSQL Databases Explained (https://www.mongodb.com/nosql-explained)

To give you a better idea, here is how the *Post* objects are structured in Firebase database.

```
{
  "posts" : {
    "-KmLSIrasVfvGDsvtYs1" : {
      "imageFileURL" : "https://firebasestorage.googleapis.com/v0/b/northlights-3d71f.appspot.com/o/photos%2F-KmLSIrasVfvGDsvtYs1.jpg?alt=media&token=123216de-8997-40a9-b9ae-c0784fa491c7",
      "timestamp" : 1497172886765,
      "user" : "Simon Ng",
      "votes" : 0
    },
    "-KmLSNCxkCC8T2xQgL9F" : {
      "imageFileURL" : "https://firebasestorage.googleapis.com/v0/b/northlights-3d71f.appspot.com/o/photos%2F-KmLSNCxkCC8T2xQgL9F.jpg?alt=media&token=2dacbf67-8bce-416b-9731-2c972d8a8012",
      "timestamp" : 1497172904579,
      "user" : "Simon Ng",
      "votes" : 2
    },
    "-KmMnVzKB-f3ZIZvAf9K" : {
      "imageFileURL" : "https://firebasestorage.googleapis.com/v0/b/northlights-3d71f.appspot.com/o/photos%2F-KmMnVzKB-f3ZIZvAf9K.jpg?alt=media&token=1f8659e5-1d18-42a5-a2fb-12fa55167644",
      "timestamp" : 1497195485301,
      "user" : "Adam Stark",
      "votes" : 3
    },
    "-KmMr6v7kX8eScNHlFsq" : {
      "imageFileURL" : "https://firebasestorage.googleapis.com/v0/b/northlights-3d71f.appspot.com/o/photos%2F-KmMr6v7kX8eScNHlFsq.jpg?alt=media&token=5572fd21-ced3-4d63-8727-e6f419d07104",
      "timestamp" : 1497196431352,
      "user" : "Shirley Jones",
      "votes" : 0
    }
}
```


Each child node of the *Posts* node is similar to a record of the *Post* table in Parse. As you can see, all data is stored in key-value pairs.

```
"-KmLSIrasVfvGDsvtYs1" : {
      "imageFileURL" : "https://firebasestorage.googleapis.com/v0/b/northlights-3d
71f.appspot.com/o/photos%2F-KmLSIrasVfvGDsvtYs1.jpg?alt=media&token=123216de-8997-
40a9-b9ae-c0784fa491c7",
      "timestamp" : 1497172886765,
      "user" : "Simon Ng",
      "votes" : 0
    }
```

Firebase Database supports the following native data types:

- `NSString`
- `NSNumber`
- `NSDictionary`
- `NSArray`

Further reading: Building a properly structured Firebase database requires quite a bit of planning. You can take a look at this article about how to structure your data in Firebase Database:

- Structure Your Database (https://firebase.google.com/docs/database/ios/structure-data)

Understanding Firebase Storage

You may wonder why we store the image's URL in the example above, instead of the image itself. The reason is that you can't save binary data or images directly into Firebase Database.

Unlike Parse, Firebase does not store images in its database. Instead, it has another product called *Storage*, which is specifically designed for storing files like images and videos.

Like our demo app, if your app needs to store images on Firebase Database, you will first need to upload your image to Firebase Storage. And then, you retrieve the download URL of that image, and save it back to Firebase Database for later retrieval.

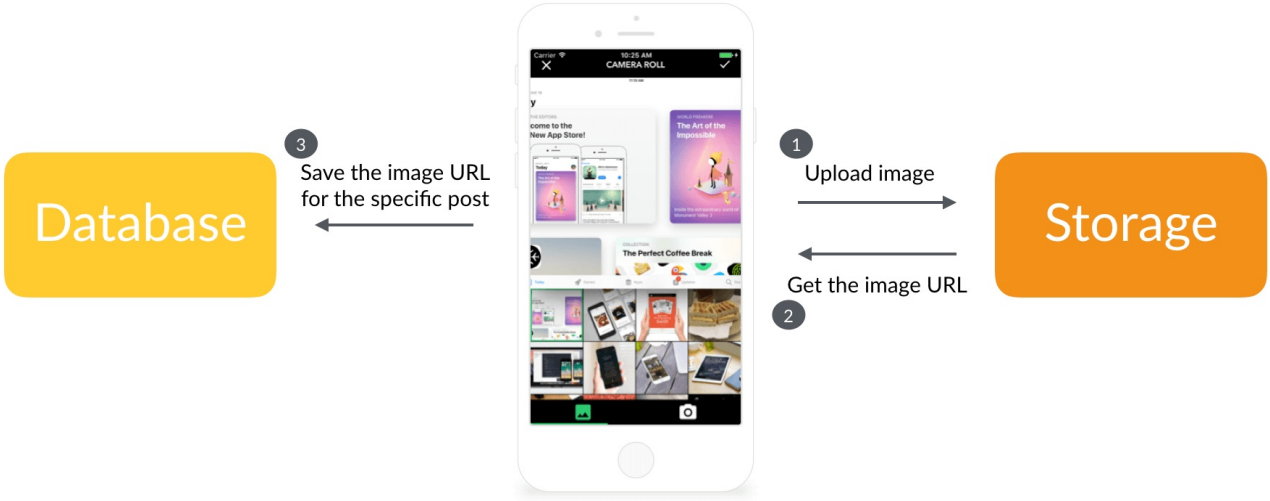


Figure 39.11. Saving images using Firebase Database and Storage

So, when your app displays a post with images, you first retrieve the post information from Firebase Database. With the image URL obtained, you go up to Firebase Storage to download the image, and then display it in the app.

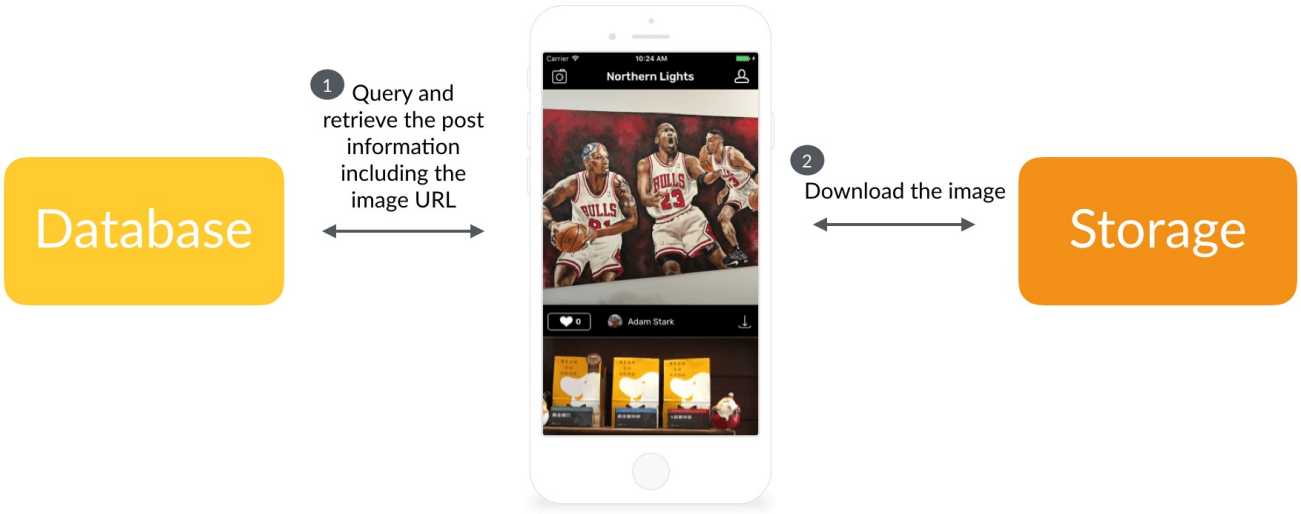


Figure 39.12. How to retrieve images from Firebase Database and Storage

Okay, I hope you now have some basic concepts of Firebase Database and Storage. The approach described in this section will be applied in our demo. I understand it will take you some time to figure out how to structure data as JSON objects, especially you are from relational database background.

Just take your time, revisit this section and check out the resources to strengthen your understanding.

Installing Firebase Database and Storage

Now it's time to get back to work. Let's start with the SDK installation.

Just like ImagePicker, we will use CocoaPods to install the required libraries of Firebase Database and Storage to our Xcode project.

First, close the Xcode project, and open `Podfile` with a text editor. Update the file like this:

```
target 'FirebaseDemo' do
  # Comment the next line if you're not using Swift and don't want to use dynamic
frameworks
  use_frameworks!

  # Pods for FirebaseDemo
  pod 'Firebase/Core'
  pod 'Firebase/Auth'
  pod 'Firebase/Database'
  pod 'Firebase/Storage'

  # Pods for Facebook
  pod 'FBSDKLoginKit'

  # Pods for Google Sign In
  pod 'GoogleSignIn'

  # Pods for ImagePicker
  pod 'ImagePicker'
end
```

Here we add two more pods including `Firebase/Database` and `Firebase/Storage`. Save the changes and open Terminal to execute the following command under the project folder:

```
pod install
```

CocoaPods will automatically download and integrate the libraries into your Xcode project.

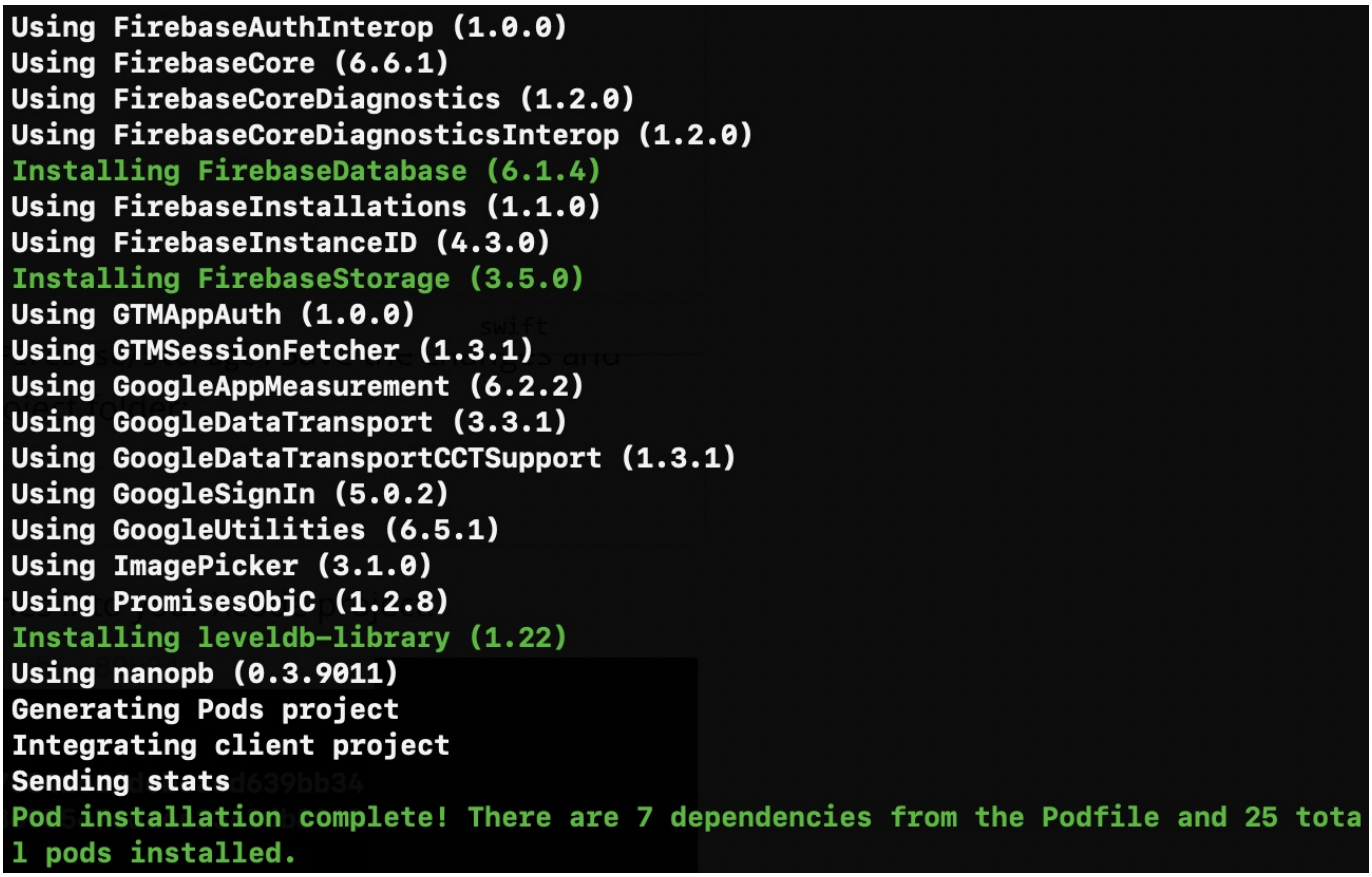


Figure 39.13. Installing the SDK of Firebase Database and Storage

After that, open `FirebaseDemo.xcworkspace` in Xcode. Build the project once to ensure everything works fine.

> **Note:** You may need to change the Swift version of ImagePicker to 4.0 again. After you run pod install, CocoaPods will reset the Swift version to 5.

Publishing a Post

As explained earlier, when a user selects and publishes a photo using the demo app, the photo will be first uploaded to Firebase Storage. With the returned image URL, we save the post information to Firebase Database. Regarding the post informati on, it includes:

- **Post ID** - a unique identifier to identify the post
- **Image File URL** - the URL of the image, which is returned by Firebase Storage
- **User** - the name of the user who publishes the photo
- **Votes** - the total number of upvotes (note: we will not discuss the upvote function in this chapter)
- **Timestamp** - the timestamp indicating when the photo was published

I know you already have a lot of questions in your head. Say, how can we upload the image from our app to Firebase Storage? How can we retrieve the image URL from Firebase Storage? How can we generate a unique post ID?

Resizing the Image

Let me start with the simple one - *resizing the image*.

Photos captured using the built-in camera are of high resolution with size over 1MB. To speed up the photo upload (as well as, download), I want to limit the resolution of the image and scale it down (if needed) before uploading it to Firebase Storage.

To do that, we will build an extension for `UIImage`. Extension in Swift is a powerful feature that lets you extend the feature of an existing class, structure, enumeration, or protocol type. Here, we are going to add a new method called `scale(newWidth:)` for the `UIImage` class using extensions.

Now go back to Xcode. In the project navigator, right click `FirebaseDemo` folder and choose *New Group*. Name the group `Util`. This is an optional step, but I want to better organize our Swift files.

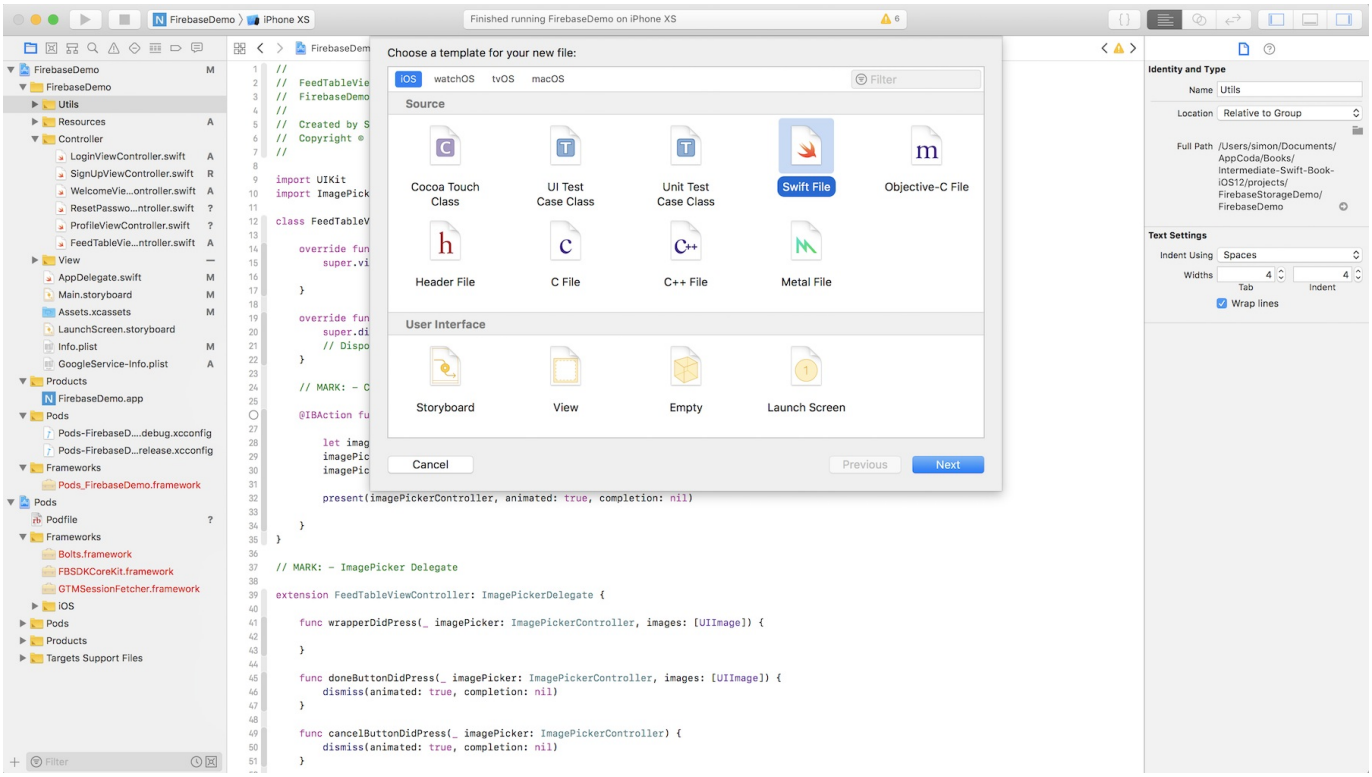


Figure 39.14. Creating a new Swift file

Next, right click `Util` folder, and select *New file*.... Choose the *Swift file* template and name the file `UIImage+Scale.swift`.

Once the file is created, replace it with the following code snippet:

```
import UIKit

extension UIImage {
    func scale(newWidth: CGFloat) -> UIImage {

        // Make sure the given width is different from the existing one
        if self.size.width == newWidth {
            return self
        }

        // Calculate the scaling factor
        let scaleFactor = newWidth / self.size.width
        let newHeight = self.size.height * scaleFactor
        let newSize = CGSize(width: newWidth, height: newHeight)

        UIGraphicsBeginImageContextWithOptions(newSize, false, 0.0);
        self.draw(in: CGRect(x: 0, y: 0, width: newWidth, height: newHeight))

        let newImage: UIImage? = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()

        return newImage ?? self
    }
}
```

This method takes the given width and resizes the image accordingly. We calculate the scaling factor based on the new width, so that we can keep the image's aspect ratio. Lastly, we create a new graphics context with the new size, draw the image and get the resized image.

Now that you have added a new method for the `UIImage` class, you can use it just like any other method in the original `UIImage` class:

```
let scaledImage = image.scale(newWidth: 960.0)
```

Uploading Files to Firebase Storage

Whether you write data to Firebase Database or Storage, it all starts with a database/storage reference. This reference serves as the entry point for accessing the database/storage.

If you go up to the Firebase console (https://console.firebase.google.com), choose your Firebase application > Storage. Click `Get Started` to proceed. You will then find a unique URL for your own storage (e.g. `gs://fir-clouddemo-cdfff.appspot.com`). All your files and folders will be saved under that location, which is known as a Google Cloud Storage bucket.

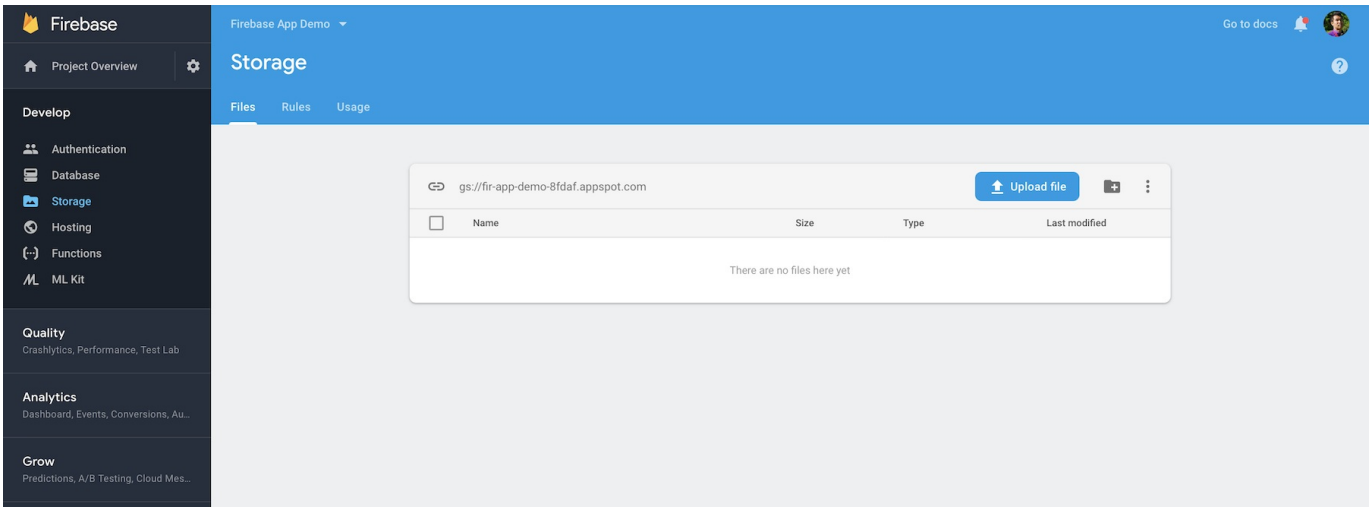


Figure 39.15. Storage of your Firebase application

There is a *Rules* option in the menu. If you go into the *Rules* section, you will see something like this:

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Firebase Storage seamlessly integrates with its authentication service that we covered in chapter 37. By default, users must be authenticated before he/she can manage files stored in the cloud storage. Firebase provides a declarative security language that allows you to modify the access rules. To learn more about how to configure the security rules, you can refer to this guide (https://firebase.google.com/docs/storage/security/start). For our demo, we will keep it unchanged.

So how can you write data to Firebase Storage?

The Firebase SDK provides the *Storage* framework that lets you interact with the cloud storage for file upload and download. To obtain a reference to the storage location of your application, you get an instance of the `Storage` class and call the `reference()` method to retrieve the root Firebase Storage location.

```
Storage.storage().reference()
```

What if you want to create something like a sub-folder? You call a `child` method with the name of your sub-folder. This creates a new reference pointing to a child object of the root storage location.

```
Storage.storage().reference().child("photos")
```

To upload a file (say, JPG image) to Storage, you instantiate a `StorageMetadata` object to specify the file's content type. And then, you call the `putData` method of the storage reference to upload the data (here, it is the image data). The data will be uploaded asynchronously to the location of the storage reference.

```
// Create the file metadata
let metadata = StorageMetadata()
metadata.contentType = "image/jpg"

// Prepare the upload task
let uploadTask = imageStorageRef.putData(imageData, metadata: metadata)
```

To monitor the progress of the upload, you attach observers to the upload task. Each observer listens to a specific `StorageTaskStatus` event. Here is an example:

```
uploadTask.observe(.success) { (snapshot) in
    // Perform operation when the upload is successful
}

uploadTask.observe(.progress) { (snapshot) in

    let percentComplete = 100.0 * Double(snapshot.progress!.completedUnitCount) /
Double(snapshot.progress!.totalUnitCount)
    print("Uploading... \(percentComplete)% complete")
}

uploadTask.observe(.failure) { (snapshot) in

    if let error = snapshot.error {
        print(error.localizedDescription)
    }
}
```

The first observer listens for the `.success` event, which will be fired when the upload has completed successfully. The second observer listens for the `.progress` event. If you need to display the progress of the upload task, you can add this observer to display the upload status. The last observer monitors the failure status of the upload.

The final question is: *how can you retrieve the URL of the saved photo?*

When an event is fired up, Firebase will pass you a snapshot of the task. You can access its `metadata` property that contains the download URL of the file.

```
snapshot.metadata?.downloadURL()?.absoluteString
```

Saving Data to Firebase Database

The way of writing data to Firebase Database is very similar to that of Storage. Instead of using the `Storage` framework, we use the `Database` framework.

Again, you first get a reference to the database of your Firebase application:

```
Database.database().reference()
```

You normally will not save your objects directly to the root location of the database. Say, in our case, we will not save each of the photo posts to the root location. Instead, we want to create a child key named `posts`, and save all the post objects under that path. To create and get a reference for the location at a specific relative path, you call the `child` method with the child key like this:

```
Database.database().reference().child("posts")
```

With the reference, it is very easy to save data to the database. You call the `setValue()` method of the reference object and specify the dictionary object you want to save:

```
let post: [String : Any] = ["imageFileURL" : imageFileURL, "votes" : Int(0), "user"
 : displayName, "timestamp" : timestamp]

postDatabaseRef.setValue(post)
```

We have discussed how we're going to structure the post data in earlier sections. The JSON tree looks like something like this:

```
{
  "posts" : {
    "-KmLSIrasVfvGDsvtYs1" : {
      "imageFileURL" : "https://firebasestorage.googleapis.com/v0/b/northlights-3d
71f.appspot.com/o/photos%2F-KmLSIrasVfvGDsvtYs1.jpg?alt=media&token=123216de-8997-
40a9-b9ae-c0784fa491c7",
      "timestamp" : 1497172886765,
      "user" : "Simon Ng",
      "votes" : 0
    },
    "-KmLSNCxkCC8T2xQgL9F" : {
      "imageFileURL" : "https://firebasestorage.googleapis.com/v0/b/northlights-3d
71f.appspot.com/o/photos%2F-KmLSNCxkCC8T2xQgL9F.jpg?alt=media&token=2dacbf67-8bce-
416b-9731-2c972d8a8012",
      "timestamp" : 1497172904579,
      "user" : "Simon Ng",
      "votes" : 2
    },

    ...
}
```

Each of the posts has a unique ID for identification, and is saved under the `/posts` path. Here, one question is: *how can you generate and assign a unique ID?*

There are various ways to do that. You can implement your own algorithm, but Firebase has provided an API for generating a new child location using a unique key. For instance, when you need to add a new post to the `/posts` location, you can call the `childByAutoId` method:

```
let postDatabaseRef =  Database.database().reference().child("posts").childByAutoI
d
```

Firebase will generate a unique key for you and return you with the generated location (e.g. `/posts/-KmLSNCxkCC8T2xQgL9F`).

With this location reference, you can save the post information under that location by calling `setValue` . Here is an example:

```
let postDatabaseRef = Database.database().reference().child("posts").childByAutoId
let post: [String : Any] = ["imageFileURL" : imageFileURL, "votes" : Int(0), "user"
 : displayName, "timestamp" : timestamp]

postDatabaseRef.setValue(post)
```

Implementing Photo Upload

Now, let's combine all the things we just learned together and build the upload function of the app.

Open `FeedTableViewController.swift`, and add the import statement:

```
import Firebase
```

Next, update the `doneButtonDidPress` method like this:

```
func doneButtonDidPress(_ imagePicker: ImagePickerController, images: [UIImage]) {

    // Get the first images
    guard let image = images.first else {
        dismiss(animated: true, completion: nil)

        return
    }

    // Generate a unique ID for the post and prepare the post database reference
    let postDatabaseRef = Database.database().reference().child("posts").childByAu
toId()

    guard let imageKey = postDatabaseRef.key else {
        dismiss(animated: true, completion: nil)

        return
    }

    // Use the unique key as the image name and prepare the storage reference
```

```
    let imageStorageRef = Storage.storage().reference().child("photos").child("\(i
mageKey).jpg")

    // Resize the image
    let scaledImage = image.scale(newWidth: 640.0)

    guard let imageData = scaledImage.jpegData(compressionQuality: 0.9) else {
        dismiss(animated: true, completion: nil)

        return
    }

    // Create the file metadata
    let metadata = StorageMetadata()
    metadata.contentType = "image/jpg"

    // Prepare the upload task
    let uploadTask = imageStorageRef.putData(imageData, metadata: metadata)

    // Observe the upload status
    uploadTask.observe(.success) { (snapshot) in

        guard let displayName = Auth.auth().currentUser?.displayName else {
            return
        }

        snapshot.reference.downloadURL(completion: { (url, error) in
            guard let url = url else {
                return
            }

            // Add a reference in the database
            let imageFileURL = url.absoluteString
            let timestamp = Int(Date().timeIntervalSince1970 * 1000)

            let post: [String : Any] = ["imageFileURL" : imageFileURL,
                                        "votes" : Int(0),
                                        "user" : displayName,
                                        "timestamp" : timestamp
                                        ]

            postDatabaseRef.setValue(post)
```

```
        })


        self.dismiss(animated: true, completion: nil)
    }

    uploadTask.observe(.progress) { (snapshot) in

        let percentComplete = 100.0 * Double(snapshot.progress!.completedUnitCount
) / Double(snapshot.progress!.totalUnitCount)
        print("Uploading \(imageKey).jpg... \(percentComplete)% complete")
    }


    uploadTask.observe(.failure) { (snapshot) in

        if let error = snapshot.error {
            print(error.localizedDescription)
        }
    }
}
```

To recap, this method is called after the user selects a photo from the photo library or takes a picture. When the method is invoked, we retrieve the selected photo and upload it to Firebase.

The code snippet is almost the same as what we have discussed in the earlier sections. But I want to highlight a couple of things:

1. As mentioned, ImagePicker is capable to handle multiple photo selection. For the demo, we limit the user to select one photo. Therefore, we retrieve the first photo at the very beginning of the method.
2. We use the unique key generated by `childByAutoId()` as the image's name. This makes sure the image filename is unique.

    ```
    let imageStorageRef = Storage.storage().reference().child("photos").child("\(image
    Key).jpg")
    ```

3. We generate a timestamp for each post, which indicates when the post is published. Later, we will display the most recent posts in the feed. This timestamp is very useful for ordering the post list in reverse chronological order (i.e. the most recent post shows first).

```
let timestamp = Int(Date().timeIntervalSince1970 * 1000)
```

Now build and run the app to have a test. After the app launches, tap the camera icon and choose a photo. Once you confirm, the app should upload the photos to Firebase and go back to the home screen.

For now, we haven't implemented the download function. Therefore, the home screen is still blank after your upload. However, if you look at the console, it should see something like this:

```
Uploading -KnNx1urzoFoKOtdXcE4.jpg... 0.0% complete
Uploading -KnNx1urzoFoKOtdXcE4.jpg... 0.0229582717013063% complete
Uploading -KnNx1urzoFoKOtdXcE4.jpg... 2.70973201137418% complete
Uploading -KnNx1urzoFoKOtdXcE4.jpg... 100.0% complete
```

This indicates the upload task has completed. Furthermore, you can go back to the Firebase console and check out your Firebase app. Look into the Storage option. You should find that the images are stored in the `photos` folder.

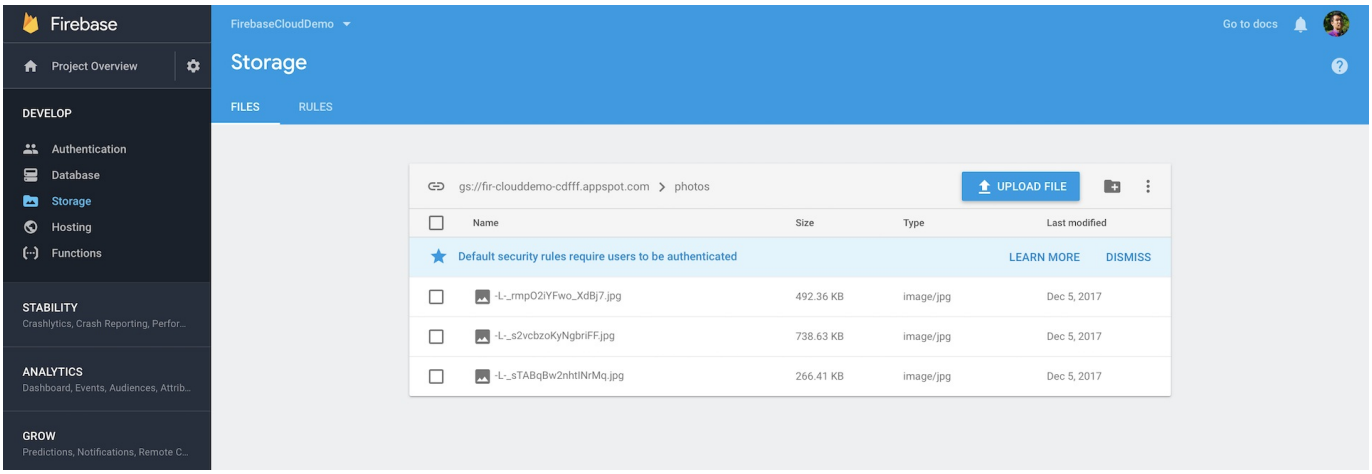


Figure 39.16. All images are stored in the Storage

Switch over to the Database option. You can also find that all your post objects are put under the `/posts` path. If you click the + button, you can reveal the details of each post. The value of *imageFileURL* is the download URL of the image. You can copy the link and paste it in any browser window to verify the image.

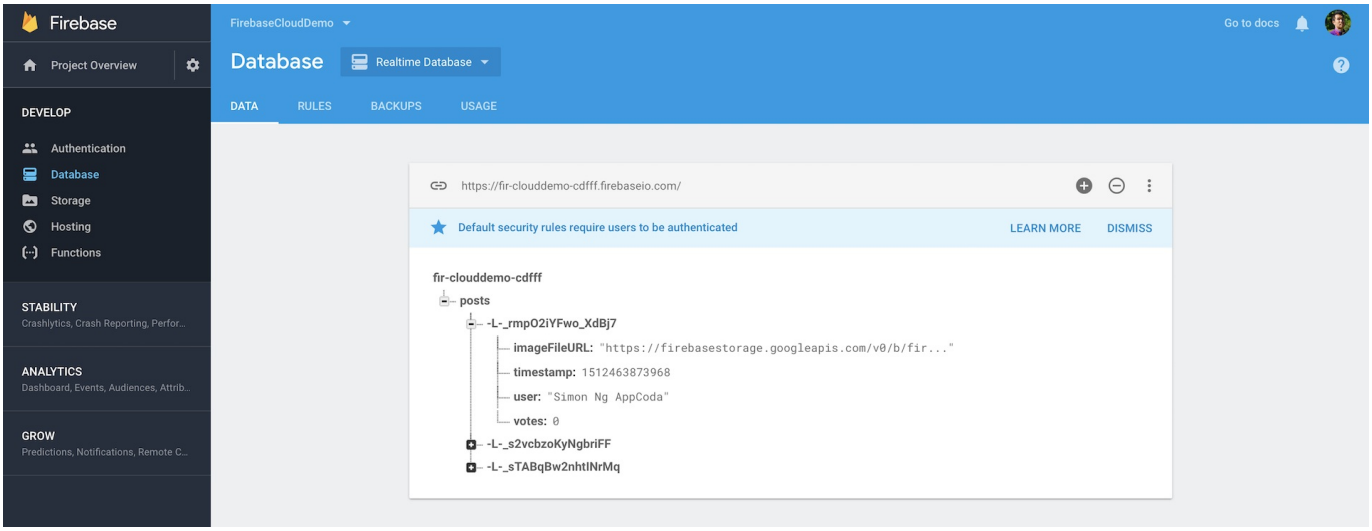


Figure 39.17. All images are stored in the Storage

For later testing, I suggest you to upload at least 15 photos.

Reading Data from Firebase Database

We've talked about upload. How about reading data from the cloud database? The Firebase Database framework provides both the `observeSingleEventOfType` and `observe` methods for developers to retrieve data. These methods are event-based, meaning that they both listen for a certain event. When that event is fired, the callback function will be invoked with a `snapshot` containing all data retrieved. You can then further process the data for display.

For example, to retrieve all the posts under `/posts` path, this is the code snippet you need:

```
let postDatabaseRef = Database.database().reference().child("posts")
postDatabaseRef.observeSingleEvent(of: .value, with: { (snapshot) in

    print("Total number of posts: \(snapshot.childrenCount)")
    for item in snapshot.children.allObjects as! [DataSnapshot] {
        let postInfo = item.value as? [String: Any] ?? [:]

        print("-------")
        print("Post ID: \(item.key)")
        print("Image URL: \(postInfo["imageFileURL"] ?? "")")
        print("User: \(postInfo["user"] ?? "")")
        print("Votes: \(postInfo["votes"] ?? "")")
        print("Timestamp: \(postInfo["timestamp"] ?? "")")
    }

})
```

The `snapshot` variable contains all the posts retrieved from `/posts` path. The `childrenCount` property tells you the total number of objects available. All the post objects are stored in `snapshot.children.allObjects` as an array of dictionaries. The key of each dictionary object is the post ID. The value of that is another dictionary containing the post information.

You can insert the code snippet above in the `viewDidLoad` method of `FeedTableViewController` to have a test. Even though we haven't populated the data in the table view, you should be able to see something like this in the console:

```
Total number of posts: 3
-------
Post ID: -KnNw1n1gFEd9vnuOpqP
Image URL: https://firebasestorage.googleapis.com/v0/b/northlights-3d71f.appspot.c
om/o/photos%2F-KnNw1n1gFEd9vnuOpqP.jpg?alt=media&token=673a2f97-191f-4ac3-b450-2c7
63052a650
User: Simon Ng
Votes: 0
Timestamp: 1498288239662

...
...
...
```

This is the way you retrieve data from Firebase Database.

The Database framework also provides the `queryOrdered` method to retrieve the JSON objects in a certain order. For example, to get the post objects in chronlogical order, you can write the following line of code:

```
let postDatabaseRef = Database.database().reference().child("posts").queryOrdered(
byChild: "timestamp")
```

The above call to the `queryOrdered(byChild:)` method specifies the child key to order the results by. Here it is the `timestamp`. This query will get the posts in chronological order.

Consider that your database has stored over 10,000 posts, you will probably aware that there is a potential issue here. As your users publish more posts to the database, it will take a longer time to download all the posts.

To prevent the potential performance issues, it would be better to set a limit for the posts to be retrieved. Firebase provides the `queryLimited(toFirst:)` and `queryLimited(toLast:)` methods to set a limit. For example, if you want to get the first 10 posts of a query, you

can use the `queryLimited(toFirst:)` method:

```
Database.database().reference().child("posts").queryLimited(toFirst: 10)
```

You can combine both `queryOrdered` and `queryLimited` methods together to form a more complex query. Say, for the demo app, we have to show the 5 most recent posts after the app is launched. We can write the query like this:

```
var postQuery = Database.database().reference().child("posts").queryOrdered(byChil
d: "timestamp")
postQuery = postQuery.queryLimited(toLast: 5)
```

We specify that the post objects should be ordered by *timestamp*. Since Firebase can only sort things in ascending order, the most recent post (with a larger value of timestamp) is the last object of an array. So we use `queryLimited(toLast: 5)` to retrieve the last 5 objects, which represents the 5 most recent posts.

Refactoring Our Code

Now that you have implemented the upload feature, you should also have some ideas about how to retrieve posts from Firebase.

Instead of jumping straight into coding, let's step back a little bit and review the code once again. If you look into the `doneButtonDidPress` function and compare with the code for data retrieval we just discussed, you will find a lot of things in common.

Figure 39.18. Reviewing the fusumaImageSelected function

First, whether for writing or reading data, we need to have a reference to Firebase Database (or Storage). I foresee the following lines of code will be written everywhere whenever we need to interact with Firebase.

```
Database.database().reference()
Database.database().reference().child("posts")
Storage.storage().reference().child("photos")
```

Secondly, it is the dictionary object holding the post data. It will also be used everywhere.

```
let post: [String : Any] = ["imageFileURL" : imageFileURL,
                            "votes" : Int(0),
                            "user" : displayName,
                            "timestamp" : timestamp
```

When writing post data to Firebase, we have to create a dictionary object of post information. Conversely, when we retrieve data from Firebase, we have to save the dictionary object and extract the post information from that object.

For now, we hardcode the key, and do not have a model class for a post. Obviously, hardcoding the key in our code is prone to error.

Lastly, the code of the `doneButtonDidPress` function is mostly related to data upload. Let me ask you. What if you have another button or class which also needs to upload a photo? Will you just duplicate the same code snippet to that button or class? You can do that but this is not a good programming practice and will make your code less manageable.

When you plan to copy and paste the same piece of code from one class to another, always ask yourself: *What if you need to modify that piece of code in the future?* If the same piece of code is scattered across several classes, you will have to modify every piece of the code. This will be a disaster.

Base on what we have reviewed, there are a couple of changes that can make our code better and more manageable:

1. Create a model structure to represent a post. This structure can take a dictionary object of the post, and convert it into a more meaningful `Post` object.
2. Create a service class to manage all the interactions between Firebase database and storage. I want to centralize all the upload and download functions of Firebase database into a single service class. Whenever you need to read/write data to the Firebase cloud, you refer to this service class and call the appropriate method. This will prevent code duplication.

These are some of the high-level changes. Now let's dive in and refactor our existing code.

Creating the Model

First, we start with the model structure. In order to better organize the project, create a folder in project navigator by right clicking the `FirebaseDemo` folder and choose *New Group*. Name the group `Model`.

Next, right click *Model* and then select *New File...*. Choose the *Swift File* template and name the file `Post.swift`. Replace its content like this:

```
import Foundation

struct Post {

    // MARK: - Properties

    var postId: String
    var imageFileURL: String
    var user: String
    var votes: Int
    var timestamp: Int

    // MARK: - Firebase Keys

    enum PostInfoKey {
        static let imageFileURL = "imageFileURL"
        static let user = "user"
        static let votes = "votes"
        static let timestamp = "timestamp"
    }

    // MARK: - Initialization

    init(postId: String, imageFileURL: String, user: String, votes: Int, timestamp
: Int = Int(Date().timeIntervalSince1970 * 1000)) {
        self.postId = postId
        self.imageFileURL = imageFileURL
        self.user = user
        self.votes = votes
        self.timestamp = timestamp
    }
```

```
    init?(postId: String, postInfo: [String: Any]) {
        guard let imageFileURL = postInfo[PostInfoKey.imageFileURL] as? String,
            let user = postInfo[PostInfoKey.user] as? String,
            let votes = postInfo[PostInfoKey.votes] as? Int,
            let timestamp = postInfo[PostInfoKey.timestamp] as? Int else {

                return nil
        }

        self = Post(postId: postId, imageFileURL: imageFileURL, user: user, votes:
 votes, timestamp: timestamp)
    }
}
```

The `Post` structure represents a basic photo post. It has various properties, such as imageFileURL, for storing the post information. The keys used in Firebase Database are constants. So we create an enum named `PostInfoKey` to store the key names. For any reason we need to alter the key name in the future, this is the single file we have to change.

Furthermore, we have created two initialization methods. When you initialize an instance of `Post`, you can pass each of the required properties including post ID, image file URL, user and the number of votes. Alternatively, you can simply pass the dictionary object of the post to create an instance.

Creating the Service Class

Now let's move on to the service class. Again in the project navigator, right click the `FirebaseDemo` folder and choose *New Group*. Name the group `Service`. Right click the *Service* folder and select *New File...*. Use the *Swift file* template and name the file `PostService.swift`.

Once the file has been created, replace the content like this:

```
import Foundation
import Firebase
import FirebaseDatabase
import FirebaseStorage

final class PostService {

    // MARK: - Properties

    static let shared: PostService = PostService()

    private init() { }

    // MARK: - Firebase Database References

    let BASE_DB_REF: DatabaseReference = Database.database().reference()

    let POST_DB_REF: DatabaseReference = Database.database().reference().child("po
sts")

    // MARK: - Firebase Storage Reference

    let PHOTO_STORAGE_REF: StorageReference = Storage.storage().reference().child(
"photos")

}
```

To recap, this service class is created to centralize the access of the Firebase Database/Storage reference, and the upload/download operations.

Therefore, in the class, we declare three constants:

- Database reference for accessing the root database location
- Database reference for accessing the `/posts` location
- Storage reference for accessing the `/photos` folder

In the code above, we apply the Singleton pattern for designing the `PostService` class. The singleton pattern is very common in the iOS SDK, and can be found everywhere in the Cocoa Touch frameworks (e.g. `UserDefaults.standard`, `UIApplication.shared`,

`URLSession.shared`). Singleton guarantees that only one instance of a class is instantiated. At any time of the application lifecycle, we want to have only a single `PostService` to refer to. This is why we apply the Singleton pattern here.

To write a Singleton in Swift, we define the `init()` method as `private` to prevent other objects from creating an instance of the class. To let other objects use `PostService`, we provide a static `shared` property that contains the instance of `PostService`.

```
final class PostService {

    static let shared: PostService = PostService()

    private init() { }
}
```

Later, if you need to use `PostService`, you can access the property like this:

```
PostService.shared.POST_DB_REF
```

Now it's time to refactor the code related to post upload. We are going to create a general method for uploading photos to Firebase. Insert a new method called `uploadImage` in the `PostService` structure:

```
func uploadImage(image: UIImage, completionHandler: @escaping () -> Void) {

    // Generate a unique ID for the post and prepare the post database reference
    let postDatabaseRef = POST_DB_REF.childByAutoId()

    // Use the unique key as the image name and prepare the storage reference
    guard let imageKey = postDatabaseRef.key else {
        return
    }

    let imageStorageRef = PHOTO_STORAGE_REF.child("\(imageKey).jpg")

    // Resize the image
    let scaledImage = image.scale(newWidth: 640.0)
```

```
guard let imageData = scaledImage.jpegData(compressionQuality: 0.9) else {
    return
}

// Create the file metadata
let metadata = StorageMetadata()
metadata.contentType = "image/jpg"

// Prepare the upload task
let uploadTask = imageStorageRef.putData(imageData, metadata: metadata)

// Observe the upload status
uploadTask.observe(.success) { (snapshot) in

    guard let displayName = Auth.auth().currentUser?.displayName else {
        return
    }

    // Add a reference in the database
    snapshot.reference.downloadURL(completion: { (url, error) in
        guard let url = url else {
            return
        }

        // Add a reference in the database
        let imageFileURL = url.absoluteString
        let timestamp = Int(Date().timeIntervalSince1970 * 1000)

        let post: [String : Any] = ["imageFileURL" : imageFileURL,
                                    "votes" : Int(0),
                                    "user" : displayName,
                                    "timestamp" : timestamp
        ]

        postDatabaseRef.setValue(post)

    })

    completionHandler()
}
```

```
        uploadTask.observe(.progress) { (snapshot) in

            let percentComplete = 100.0 * Double(snapshot.progress!.completedUnitCount
    ) / Double(snapshot.progress!.totalUnitCount)
            print("Uploading... \(percentComplete)% complete")
        }

        uploadTask.observe(.failure) { (snapshot) in

            if let error = snapshot.error {
                print(error.localizedDescription)
            }
        }
    }
```

As you can see, the body of the method is nearly the same as the code in the `doneButtonDidPress` method except that:

1. We refer to the database reference constants defined in `PostService`.
2. We no longer hardcode the key of the post dictionary. Instead, we refer to the constants defined in `PostInfoKey` of the `Post` structure.
3. The method has a parameter called `completionHandler`, which is a closure. The caller of this method can pass a function, that will be executed after the post has been uploaded.

Modifying the Existing Code

Now that we have created both the model and service classes, let's modify the `doneButtonDidPress` of the `FeedTableViewController` class to make use of the new classes.

Update the method like this:

```
func doneButtonDidPress(_ imagePicker: ImagePickerController, images: [UIImage]) {

    // Get the first images
    guard let image = images.first else {
        dismiss(animated: true, completion: nil)

        return
    }

    // Upload image to the cloud
    PostService.shared.uploadImage(image: image) {
        self.dismiss(animated: true, completion: nil)
    }

}
```

That's it! You can build and run the project to have a test. From the user perspective, everything is exactly the same as before.

However, the code now looks much cleaner, and is easier to maintain.

Implementing the Post Feed

With a well organized code structure, it is time to continue to work on the download feature.

We have already discussed how we can read data from Firebase, and retrieve the post objects. Let's first create a new method in `PostService` for downloading the recent posts. I foresee this method will be used for these two situations:

1. When the app is first launched, the method will be used to retrieve the 5 most recent posts.
2. The app has a pull-to-refresh feature for refreshing the photo feed. In this case, we want to retrieve the posts newer than the most recent post in the post feed.

To fulfill the requirements mentioned above, this method is designed to accept three parameters:

- `start` - It will have an optional `start` parameter that takes a timestamp. If the caller of the method specifies the timestamp, we will retrieve the post objects with a timestamp newer than the given value.
- `limit` - The maximum number of post objects to be retrieved.
- `completionHandler` - a closure to be executed after the post objects are retrieved. We will pass an array of posts (in reverse chronological order) as the parameter of the closure.

Now open the `PostService.swift` file, and create the method `getRecentPosts`:

```
func getRecentPosts(start timestamp: Int? = nil, limit: UInt, completionHandler: @
escaping ([Post]) -> Void) {

    var postQuery = POST_DB_REF.queryOrdered(byChild: Post.PostInfoKey.timestamp)
    if let latestPostTimestamp = timestamp, latestPostTimestamp > 0 {
        // If the timestamp is specified, we will get the posts with timestamp new
er than the given value
        postQuery = postQuery.queryStarting(atValue: latestPostTimestamp + 1, chil
dKey: Post.PostInfoKey.timestamp).queryLimited(toLast: limit)
    } else {
        // Otherwise, we will just get the most recent posts
        postQuery = postQuery.queryLimited(toLast: limit)
    }

    // Call Firebase API to retrieve the latest records
    postQuery.observeSingleEvent(of: .value, with: { (snapshot) in

        var newPosts: [Post] = []
        for item in snapshot.children.allObjects as! [DataSnapshot] {
            let postInfo = item.value as? [String: Any] ?? [:]

            if let post = Post(postId: item.key, postInfo: postInfo) {
                newPosts.append(post)
            }
        }

        if newPosts.count > 0 {
            // Order in descending order (i.e. the latest post becomes the first p
ost)
            newPosts.sort(by: { $0.timestamp > $1.timestamp })
        }

        completionHandler(newPosts)

    })

}
```

You should be familiar with part of the code. The idea is that we build a query with using `POST_DB_REF` and retrieve the posts in chronological order. If no `timestamp` is given, we simply call `queryLimited(toLast:)` to get the most recent posts.

> **Note:** You may wonder why we call *queryLimited(toLast:)* instead of *queryLimited(toFirst:)*. Firebase sorts the posts in chronological order. If you consider the posts as an array of objects, the first item will be the oldest post, while the most recent post will be the last item of the array. Hence, we use *queryLimited(toLast:)* to get the most recent posts.

When the caller of the method passes us a `timestamp`, we will retrieve the post objects with timestamp larger than the given value. So we build a query like this:

```
postQuery = postQuery.queryStarting(atValue: latestPostTimestamp + 1, childKey: Po
st.PostInfoKey.timestamp).queryLimited(toLast: limit)
```

The `queryStarting(atValue:)` method is used to generate a reference to a limited view of the data. Here only those posts with the value of `timestamp` larger than the given timestamp will be included in the result of the query.

Figure 39.19 shows you an example. Assuming there are just two posts displayed in the post feed, the most recent post has the timestamp `10002`. When you call `queryStarting(atValue: 10002 + 1)`, it will retrieve all the posts with a timestamp value greater than and equal to `10003`.

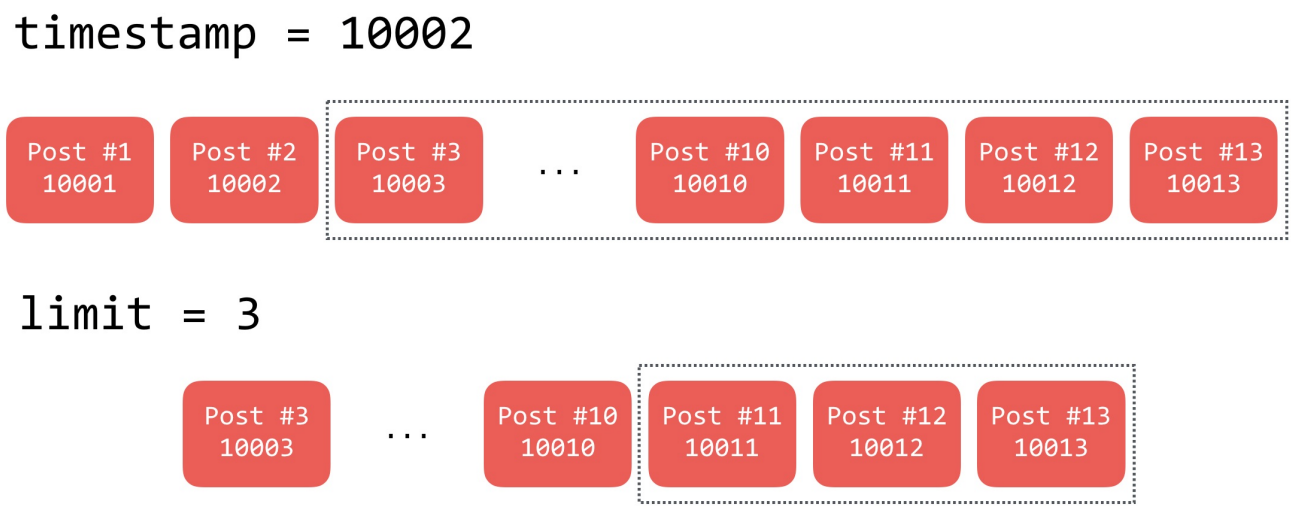


Figure 39.19. How the query works

In the query, we also combine the `queryLimited` method to limit the total number of posts retrieved. Say, if we set the limit to `3`, only the 3 most recent posts will be downloaded.

After the query is prepared, we call `observeSingleEvent` to execute the query and retrieve the post objects. All the objects returned are saved in the `newPosts` array. Since Firebase sorts the posts in chronological order, we order it into an array of `Post` object in reverse chronological order. In other words, the most recent post becomes the first object in the array.

```
newPosts.sort(by: { $0.timestamp > $1.timestamp })
```

Lastly, we call the given `completionHandler` and pass it the array of posts for further processing.

In the next section, we will discuss how to populate the posts in `FeedTableViewController`. That said, if you can't wait to test the method, update the `viewDidLoad` method of the `FeedTableViewController` class like below:

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

    PostService.shared.getRecentPosts(limit: 3) { (newPosts) in

        newPosts.forEach({ (post) in
            print("-------")
            print("Post ID: \(post.postId)")
            print("Image URL: \(post.imageFileURL)")
            print("User: \(post.user)")
            print("Votes: \(post.votes)")
            print("Timestamp: \(post.timestamp)")
        })
    }
}
```

Run the app, and you will see messages similar to the following after your login:

```
Post ID: -L-aAYItxhmVeFv_EUff
Image URL: https://firebasestorage.googleapis.com/v0/b/fir-clouddemo-cdfff.appspot
.com/o/photos%2F-L-aAYItxhmVeFv_EUff.jpg?alt=media&token=8624ef61-b3cc-4894-b1a0-0
ece9d13ac38
User: Simon Ng AppCoda
Votes: 0
Timestamp: 1512469052775

...
```

This indicates the method is already working and retrieving the 3 most recent posts from your Firebase database.

Populating Photos into the Table View

I assume you understand how to work with `UITableView` and `UITableViewController`, so this section will be much easier compared the previous section.

To populate the posts in the table view, we have to implement a few things:

1. Override the required methods of `UITableViewDataSource` to display the post content.
2. The table view cell should be able to download the photo from the given image URL in the background.
3. To improve the table view performance, we will cache the images in memory after they are downloaded.

Building the Cache Manager

Let's start with point 3 and implement the cache manager. The iOS SDK has provided a handy class called `NSCache` for caching objects in key-value pairs. We will create a `CacheManager` class to centralize the cache management of the posts.

In the project navigator, right click the *Service* folder and choose *New File*.... Select the *Swift File* template and name the file `CacheManager.swift`.

Insert the following code snippet in the file:

```
enum CacheConfiguration {
    static let maxObjects = 100
    static let maxSize = 1024 * 1024 * 50
}

final class CacheManager {

    static let shared: CacheManager = CacheManager()
    private static var cache: NSCache<NSString, AnyObject> = {
        let cache = NSCache<NSString, AnyObject>()
        cache.countLimit = CacheConfiguration.maxObjects
        cache.totalCostLimit = CacheConfiguration.maxSize

        return cache
    }()

    private init() { }

    func cache(object: AnyObject, key: String) {
        CacheManager.cache.setObject(object, forKey: key as NSString)
    }

    func getFromCache(key: String) -> AnyObject? {
        return CacheManager.cache.object(forKey: key as NSString)
    }
}
```

`NSCache` has two properties for managing the cache size. You can define the maximum number of objects and the maximum size of the objects it can hold. For this, we define a `CacheConfiguration` enum holding constants for these two values.

Similar to `PostService`, the `CacheManager` class is a singleton. It has an instance of `NSCache` that can hold up to 100 objects with a size limit of 50MB.

The class provides two methods for adding an object to cache, and retrieving an object from the cache by specifying the key of the object.

Displaying the Post Information and Downloading the Post Image in PostCell

With the cache implementation ready, let's now update the `PostCell` class to implement the image download. `PostCell` is a custom class of the prototype cell in `FeedTableViewController`.

To configure the cell for a particular post, we will create a `configure` method that takes in a `Post` object. Insert the following code in the `PostCell` class:

```
func configure(post: Post) {

    // Set the cell style
    selectionStyle = .none

    // Set name and vote count
    nameLabel.text = post.user
    voteButton.setTitle("\(post.votes)", for: .normal)
    voteButton.tintColor = .white

    // Reset image view's image
    photoImageView.image = nil

    // Download post image
    if let image = CacheManager.shared.getFromCache(key: post.imageFileURL) as? UI
Image {
        photoImageView.image = image

    } else {
        if let url = URL(string: post.imageFileURL) {

            let downloadTask = URLSession.shared.dataTask(with: url, completionHan
dler: { (data, response, error) in

                guard let imageData = data else {
                    return
                }

                OperationQueue.main.addOperation {
                    guard let image = UIImage(data: imageData) else { return }
```

```
                    self.photoImageView.image = image

                    // Add the downloaded image to cache
                    CacheManager.shared.cache(object: image, key: post.imageFileUR
L)
                }

            })

            downloadTask.resume()
        }
    }
}
```

The first few lines of the code above set the publisher's name of the photo, and the vote count.

The given `post` object contains the image's URL. We first check if it can found in the cache. If that is true, we display the image right away. Otherwise, we go up to Firebase and download the image by creating a dataTask of `URLSession`.

Implementing the UITableViewDataSource Methods for Displaying the Post Cells

Now that we have both the cache manager and the post cell ready, let's implement the required methods of `UITableViewDataSource` to display the post cells.

Go back to `FeedTableViewController.swift` and declare two properties in the class:

```
var postfeed: [Post] = []
fileprivate var isLoadingPost = false
```

The `postfeed` property keeps all the current posts (in reverse chronological order) for displaying in the table view. By default, it is empty. The `isLoadingPost` property indicates whether the app is downloading posts from Firebase. It will be used later for implementing infinite scrolling.

Note: In Swift, it provides various access modifier such as public, private, internal for controlling the access of a type. `fileprivate` was first introduced in Swift 3 to restrict the access of an entity to its own defining source file. Here, `isLoadingPost` can only be accessed by entities defined in the FeedTableViewController.swift file.

Next, create an extension of `FeedTableViewController` to override the default implementation of these methods:

```
// MARK: - UITableViewDataSource Methods

extension FeedTableViewController {

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: Inde
xPath) -> UITableViewCell {

        let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: inde
xPath) as! PostCell

        let currentPost = postfeed[indexPath.row]
        cell.configure(post: currentPost)

        return cell
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection sectio
n: Int) -> Int {
        return postfeed.count
    }

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

The code above is very straightforward. It is just a standard implementation for populating data in the table view. We specify the total number of rows to display and tell each cell what to display in the `tableView(_:cellForRowAt:)` method.

As you may notice, there is one important thing that is missing. We haven't retrieved the posts from Firebase.

Now add two new methods named `loadRecentPosts` and `displayNewPosts` in the `FeedTableViewController` class:

```
// MARK: - Managing Post Download and Display

fileprivate func loadRecentPosts() {

    isLoadingPost = true

    PostService.shared.getRecentPosts(start: postfeed.first?.timestamp, limit: 10)
 { (newPosts) in

        if newPosts.count > 0 {
            // Add the array to the beginning of the posts arrays
            self.postfeed.insert(contentsOf: newPosts, at: 0)
        }

        self.isLoadingPost = false

        self.displayNewPosts(newPosts: newPosts)

    }
}

private func displayNewPosts(newPosts posts: [Post]) {
    // Make sure we got some new posts to display
    guard posts.count > 0 else {
        return
    }

    // Display the posts by inserting them to the table view
    var indexPaths:[IndexPath] = []
    self.tableView.beginUpdates()
    for num in 0...(posts.count - 1) {
        let indexPath = IndexPath(row: num, section: 0)
        indexPaths.append(indexPath)
    }
    self.tableView.insertRows(at: indexPaths, with: .fade)
    self.tableView.endUpdates()
}
```

In the `loadRecentPosts` method, we call the `getRecentPosts` method of `PostService` to get the 10 most recent posts. We then pass the posts to another new method `displayNewPosts` for display.

To insert the new posts into the table view, we use the `insertRows` method of `UITableView`, along with the `beginUpdates()` and `endUpdates()` to perform a batch insertions.

We are almost done.

The final thing is to modify the `viewDidLoad` method to call `loadRecentPosts()`.

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

    // Load recent posts
    loadRecentPosts()
}
```

You've made it! It is time to hit the Run button to try out the app. If you have everything correctly, the app should show you the 10 most recent posts.

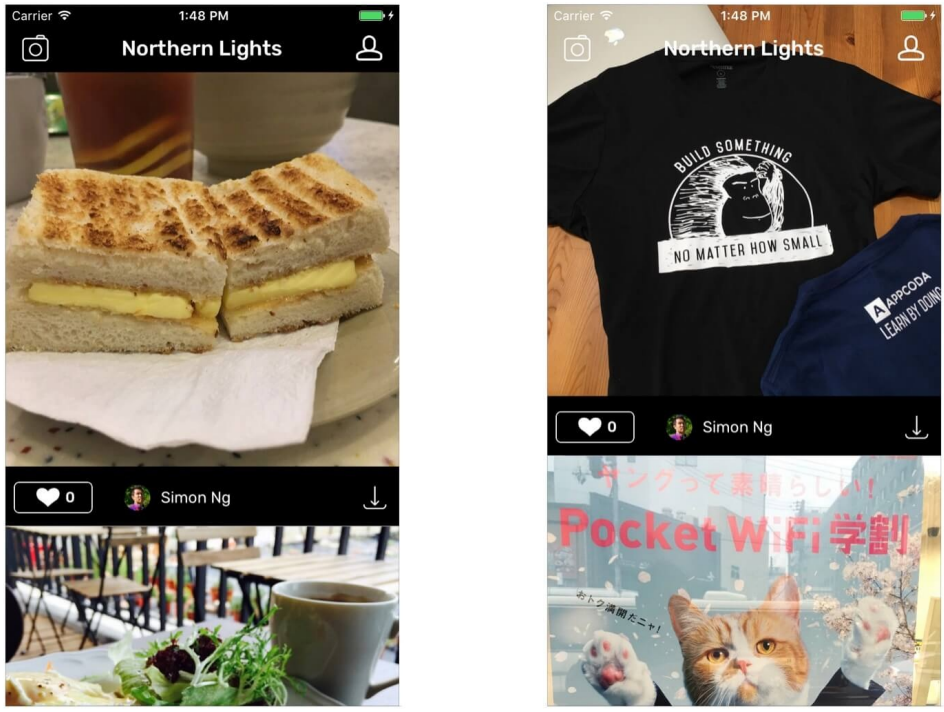


Figure 39.20. Testing the demo app

Fixing the Image Loading Issues

If you scroll the table view at normal speed, the app works pretty well. Now re-run the app. This time, try to scroll the table view very quickly. You will probably discover an undesirable behaviour, especially your network speed is slow. The image of the reused cells got overwritten twice (or even more) before showing the final image.

The app is functionally correct. At the end of the loading, the cells display the correct images. You just experience a minor flickering effect.

What happens here? Figure 39.21 explains why there is an image loading issue.

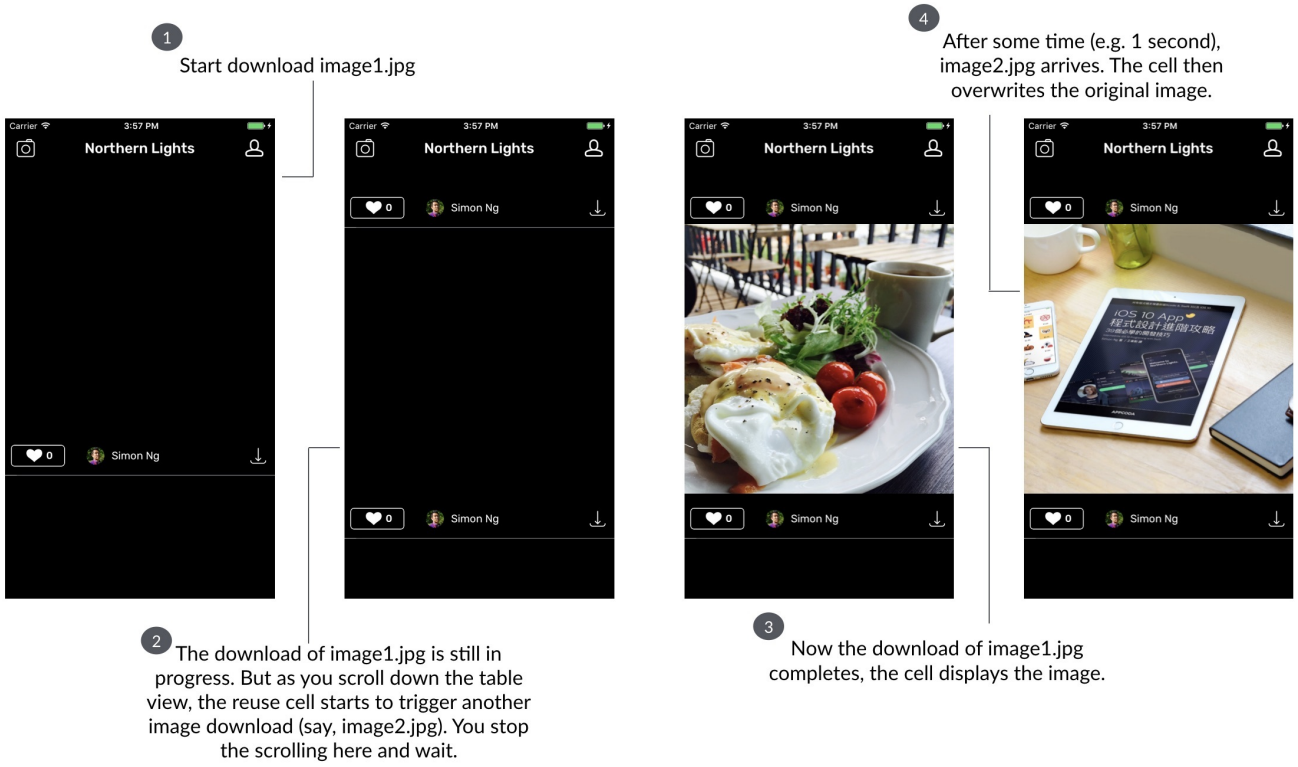


Figure 39.21. An illustration showing you why the image of a reuse cell got overwritten

Cell reuse in table views is a way to optimize resources and keep the scrolling smooth. However, you will have to take special care of situations like this.

How can you resolve this issue?

There are multiple ways to fix the issue. Let me show you one of the simple solutions.

First, think again about the root cause of the issue. Stop here, don't look at the solution. I really want you to think.

Okay, let's take a look at figure 39.21 again. Let me call that reuse cell `Cell A`. When the app starts, `Cell A` starts to download image1.jpg. The user quickly scrolls down the table view and reaches another cell. This cell reuses `Cell A` for rendering the cell's content, and it triggers another download operation for image2.jpg. Now we have two download tasks in progress. When the download of image1.jpg completes, `Cell A` immediately displays the image.

Wait, do you smell a problem here?

The reuse version of `Cell A` is supposed to display image2.jpg instead of image1.jpg. So even the download of image1.jpg completes, the cell shouldn't display the image. It should only display image2.jpg.

To resolve the issue, what we can do is to add a verification right before displaying the cell's image. Each of the cells should only display the image it supposes to be displayed.

Now open `PostCell.swift` to modify some of the code. To let the cell know which post image it is responsible for, declare a new property to save the current post:

```
private var currentPost: Post?
```

At the beginning of the `configure` method, insert the following lines to set the value of `currentPost`:

```
// Set current post
currentPost = post
```

Lastly, replace this line of code defined in `addOperation`:

```
self.photoImageView.image = image
```

With:

```
if self.currentPost?.imageFileURL == post.imageFileURL {
    self.photoImageView.image = image
}
```

In the completion handler of the download task, we add a simple verification to ensure that we only display the right image. For the image that is not supposed to be displayed by the current cell, we just keep it in the cache.

That is how we fix the undesirable behaviour of the cells. You can now test the app again to ensure the image loading issue is resolved.

Pull to Refresh the Post Feed

Do you spot another problem with the app?

Yes! After you upload a photo, the post feed doesn't refresh to load your photo.

To fix that, update the `doneButtonDidPress` method like this:

```
func doneButtonDidPress(_ imagePicker: ImagePickerController, images: [UIImage]) {

    // Get the first images
    guard let image = images.first else {
        dismiss(animated: true, completion: nil)

        return
    }

    // Upload image to the cloud
    PostService.shared.uploadImage(image: image) {
        self.dismiss(animated: true, completion: nil)
        self.loadRecentPosts()
    }

}
```

All you need to do is to call the `loadRecentPosts()` method to load the latest posts.

We also want to provide the Pull-to-refresh feature for users to refresh the feed anytime they want. I believe you're very familiar with this build-in control (i.e. `UIRefreshControl`).

In the `FeedTableViewController` class, update the `viewDidLoad` method to set up pull-to-refresh:

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

    // Configure the pull to refresh
    refreshControl = UIRefreshControl()
    refreshControl?.backgroundColor = UIColor.black
    refreshControl?.tintColor = UIColor.white
    refreshControl?.addTarget(self, action: #selector(loadRecentPosts), for: UICon
trol.Event.valueChanged)

    // Load recent posts
    loadRecentPosts()
}
```

We instantiate an instance of `UIRefreshControl`, configure its color, and specify which method to call when the user triggers pull-to-refresh.

Once you add the code snippet, Xcode indicates there is an error.

Figure 39.22. Xcode indicates there is an error for the selector

The problem is the selector. Selectors are a feature of Objective-C and can only be used with methods that are exposed to the dynamic Objective-C runtime. What Xcode is complaining is that `loadRecentPosts` is not exposed to Objective-C. All you need to do is to add the `@objc` attribute in the method declaration. Replace your `loadRecentPosts()` method with the following code:

```
@objc fileprivate func loadRecentPosts() {

    isLoadingPost = true

    PostService.shared.getRecentPosts(start: postfeed.first?.timestamp, limit: 10)
{ (newPosts) in

        if newPosts.count > 0 {
            // Add the array to the beginning of the posts arrays
            self.postfeed.insert(contentsOf: newPosts, at: 0)
        }

        self.isLoadingPost = false

        if let _ = self.refreshControl?.isRefreshing {
            // Delay 0.5 second before ending the refreshing in order to make the
animation look better
            DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + 0.5, exec
ute: {
                self.refreshControl?.endRefreshing()
                self.displayNewPosts(newPosts: newPosts)
            })
        } else {
            self.displayNewPosts(newPosts: newPosts)
        }

    }
}
```

In additional to the `@objc` attribute, we also modify the method to support pull to refresh. As you can see, before calling `displayNewPosts`, we check if the refresh control is active. If yes, we call its `endRefreshing()` method to disable it.

Okay, hit Run to test the app. To test the pull-to-refresh feature, you better deploy the app to two devices (or 1 device + 1 simulator). While one device publishes new posts, the other device can try out the pull-to-refresh feature.

Infinite Scrolling in Table Views

Presently, the app can only display the 10 most recent posts when it is first loads up. It is quite sure that your database would have more than 10 photos.

So, how can the user view the old posts or photos?

When you scroll to the end of the table, some apps display a *Load more* button for users to load more content. Some apps, like Facebook and Instagram, automatically load new content as you approach the bottom of the table view. This later feature is usually known as *infinite scrolling*.

For the demo app, we will implement infinite scrolling for the post feed.

Let's begin with the `PostService` class. In order to load older posts, we will create a new method named `getOldPosts` for this purpose:

```
func getOldPosts(start timestamp: Int, limit: UInt, completionHandler: @escaping (
[Post]) -> Void) {

    let postOrderedQuery = POST_DB_REF.queryOrdered(byChild: Post.PostInfoKey.time
stamp)
    let postLimitedQuery = postOrderedQuery.queryEnding(atValue: timestamp - 1, ch
ildKey: Post.PostInfoKey.timestamp).queryLimited(toLast: limit)

    postLimitedQuery.observeSingleEvent(of: .value, with: { (snapshot) in

        var newPosts: [Post] = []
        for item in snapshot.children.allObjects as! [DataSnapshot] {
            print("Post key: \(item.key)")
            let postInfo = item.value as? [String: Any] ?? [:]

            if let post = Post(postId: item.key, postInfo: postInfo) {
                newPosts.append(post)
            }
        }

        // Order in descending order (i.e. the latest post becomes the first post)
        newPosts.sort(by: { $0.timestamp > $1.timestamp })

        completionHandler(newPosts)

    })

}
```

Similar to the implementation of `getRecentPosts`, the method takes in a `timestamp`, `limit` and `completionHandler`. What this method does is that it retrieves posts older than the given timestamp. Let me give you an example as illustrated in figure 39.23. Assuming post #12 is the last post displayed in the table view, we call `getOldPosts` with the timestamp of post #12 to retrieve some older posts.

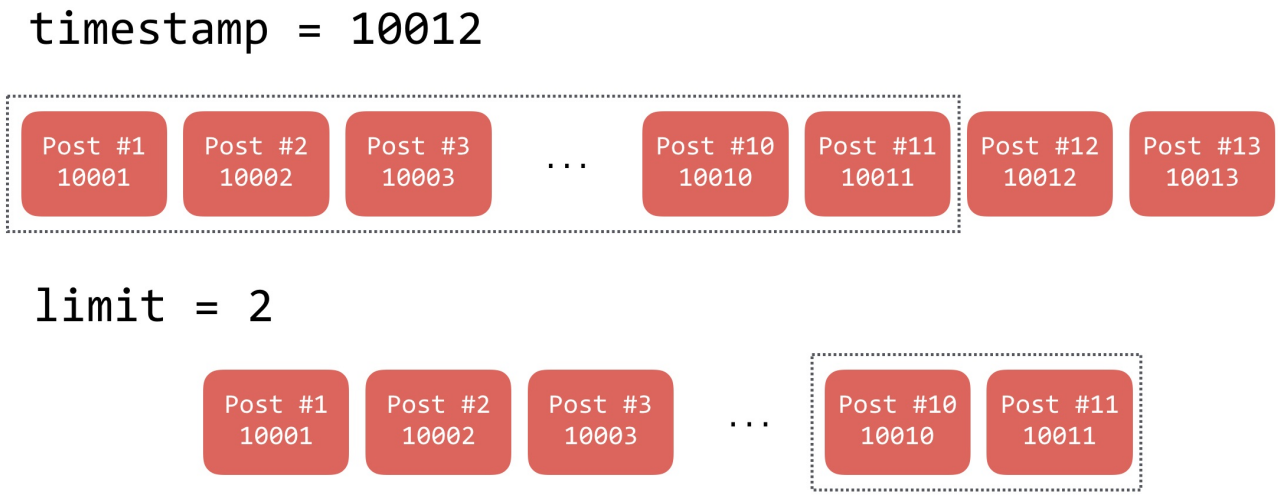


Figure 39.23. A sample usage of the getOldPosts method

This method first gets all the post objects with a timestamp smaller (i.e. older) than the given timestamp. If you refer to the code above, this is achieved by the `postOrderedQuery.queryEnding(atValue:)` method.

The `limit` parameter controls the maximum number of posts objects to be retrieved. Say, the limit is set to 2, we only get last two objects.

Now that we have prepared the service method, how can we implement infinite scrolling in the table view? A better question is how do we know the user approaches the last item of the table view?

The `tableView(_:willDisplay:forRowAt:)` method of the `UITableViewDelegate` protocol is perfect for this purpose:

```
optional func tableView(_ tableView: UITableView,
            willDisplay cell: UITableViewCell,
               forRowAt indexPath: IndexPath)
```

Whenever the table view is about to draw a cell for a particular row, this method will be called.

In the extension of `FeedTableViewController`, implement the method like this:

```
override func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCel
l, forRowAt indexPath: IndexPath) {

    // We want to trigger the loading when the user reaches the last two rows
    guard !isLoadingPost, postfeed.count - indexPath.row == 2 else {
        return
    }

    isLoadingPost = true

    guard let lastPostTimestamp = postfeed.last?.timestamp else {
        isLoadingPost = false
        return
    }

    PostService.shared.getOldPosts(start: lastPostTimestamp, limit: 3) { (newPosts
) in
        // Add new posts to existing arrays and table view
        var indexPaths:[IndexPath] = []
        self.tableView.beginUpdates()
        for newPost in newPosts {
            self.postfeed.append(newPost)
            let indexPath = IndexPath(row: self.postfeed.count - 1, section: 0)
            indexPaths.append(indexPath)
        }
        self.tableView.insertRows(at: indexPaths, with: .fade)
        self.tableView.endUpdates()

        self.isLoadingPost = false
    }
}
```

At the beginning of the method, we verify if the user has almost reached the end of the table. If the result is positive and the app is not loading new posts, we call `getOldPosts` of the `PostService` class to retrieve the older posts, and insert them into the table view.

That's it! Build and run the project to try it out. The app keeps showing you new posts as you scroll the table, until all posts are displayed.

Index Your Data in Firebase

Congratulations! You've made it this far. The post feed has been implemented with infinite scrolling. Everything just works.

However, if you look into the console, it keeps showing you the following message when querying data from Firebase:

```
2017-06-26 22:08:15.074 FirebaseDemo[54130] <Warning> [Firebase/Database][I-RDB034
028] Using an unspecified index. Consider adding ".indexOn": "timestamp" at /posts
 to your security rules for better performance
```

Firebase lets you query your data without indexing. But indexing can greatly improve the performance of your queries.

For our queries, we tell Firebase to order the post objects by `timestamp`. The warning message shown above informs you that you should tell Firebase to index the `timestamp` key at `/posts`.

To index the data, you can define the indexes via the `.indexOn` rule in your Firebase Realtime Database Rules. Now open Safari and access the Firebase console (https://console.firebase.google.com). In your Firebase application, choose the *Database* option and then select the *Rules* tab.

Update the rules like this:

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null",
    "posts": {
      ".indexOn": ["timestamp"]
    }
  }
}
```

By adding the `.indexOn` rule for the `timestamp` key, it tells Firebase to optimize queries for timestamp. Hit the *Publish* button to save the changes.

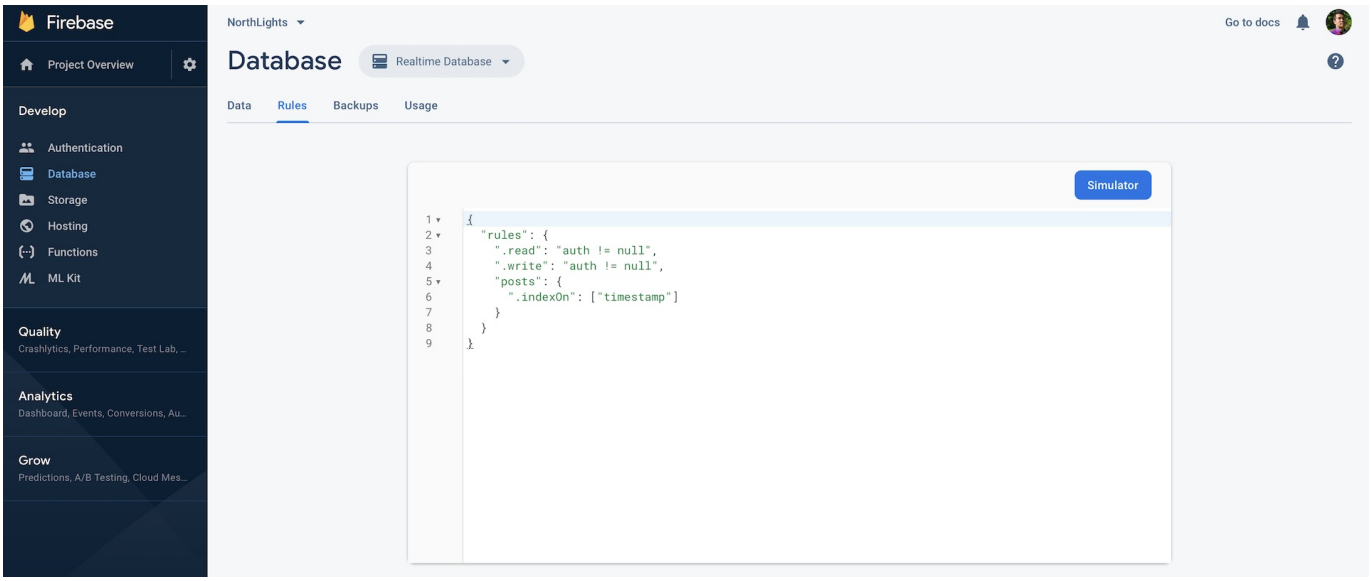


Figure 39.24. Adding rules for indexing your data

If you re-run your app, the warning message disappears. As your data grows, this index will definitely help you speed up your queries.

Summary

This is a huge chapter. We cover a lot of stuff in this single chapter. By now, I hope you fully understand how to use Firebase as your mobile backend. The NoSQL database of Firebase is very powerful and efficient. If you come from the world of SQL database, it will take you some time to digest the material. Just don't get discouraged, I know you will appreciate the beauty of Firebase database.

For further reading, refer to the official document of Firebase at https://firebase.google.com/docs/database/.

The full project of the demo app can be downloaded from http://www.appcoda.com/resources/swift5/FirebaseStorageDemo.zip.

Chapter 40
Building a Real-time Image Recognition App Using Core ML

In iOS 11, Apple released a lot of anticipated frameworks and APIs for developers to use. Other than ARKit, which we will talk about in the next chapter, Core ML is definitely another new framework that got the spotlight.

Over the past years, machine learning has been one of the hottest topics, with tech giants like Google, Amazon, and Facebook competing in this field and trying to add AI services to differentiate their offerings. Other than Siri, the intelligent personal assistant, Apple

has been quite silent about their view on AI and how developers can apply machine learning on iOS apps. Core ML is one of the Apple's answers, empowering developers with simple APIs to make their apps more intelligent.

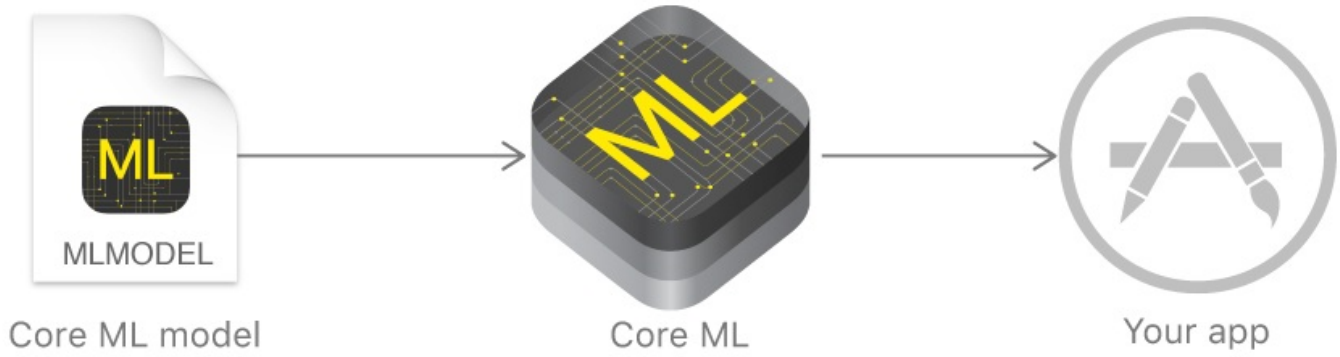


Figure 40.1. Integrate a machine learning model using Core ML

With the Core ML framework, developers can easily integrate trained machine learning models into iOS apps. In brief, machine learning is an application of artificial intelligence (AI) that allows a computer program to learn from historical data and then make predictions. A *trained ML model* is a result of applying a machine learning algorithm to a set of training data.

> Core ML lets you integrate a broad variety of machine learning model types into your app. In addition to supporting extensive deep learning with over 30 layer types, it also supports standard models such as tree ensembles, SVMs, and generalized linear models. Because it's built on top of low level technologies like Metal and Accelerate, Core ML seamlessly takes advantage of the CPU and GPU to provide maximum performance and efficiency. You can run machine learning models on the device so data doesn't need to leave the device to be analyzed.
>
> - Apple's official documentation about Core ML

Let's say, you want to build an app with a feature to identify a person's emotion (e.g. happy, angry, sad). You will need to train a ML model that can make this kind of predictions. To train the model, you will feed it a huge set of data that teaches it how a

happy face looks or how an angry face looks like. In this example, the trained ML model takes an image as the input, analyze the facial expression of the person in that image, and then predicts the person's emotion as the output.

Figure 40.2. A sample set of images for training the model

Before the introduction of Core ML, it is hard to incorporate the trained ML model in iOS apps. Now, with this new framework, you can convert the trained model into Core ML format, integrate it into your app, and use the model to make your app more intelligent. Most importantly, as you will see in a while, it only takes a few lines of code to use the model.

Note: If you are new to machine learning and artificial intelligence, I highly recommend you to check out this beginner's guide - Machine Learning for Humans written by Vishal Maini (https://medium.com/machine-learning-for-humans/why-machine-learning-matters-6164faf1df12).

In this tutorial, we will focus on using some readily available models for building our Core ML demo. Regarding how a machine learning model is trained, this is out of our scope for this article. But if you are interested to learn training your own model, you can take a look at the following references:

- How to Use Machine Learning to Predict the Quality of Wines (https://medium.freecodecamp.org/using-machine-learning-to-predict-the-quality-of-wines-9e2e13d7480d)
- A simple deep learning model for stock price prediction using TensorFlow (https://medium.com/mlreview/a-simple-deep-learning-model-for-stock-price-prediction-using-tensorflow-30505541d877)

App Demo Overview

Now that I have given you a quick overview of Core ML, let's dive into the implementation. As a demo, we will build a simple app which captures real-time images, analyzes them, and predicts what the object is in the images. Figure 40.3 will give you some ideas about the app demo.

Figure 40.3. A real-time image recognition app

As mentioned earlier, we will not build our own ML model. Instead, we rely on a ready-to-use trained model. If you go up to Apple's Machine Learning website (https://developer.apple.com/machine-learning/models/), you will find a number of Core ML models including:

- **MobileNetV2** - for detecting the dominant objects present in an image from a set of 1000 categories such as trees, animals, food, vehicles, people, and more.
- **SqueezeNet** - similar to MobileNet, it is used for detecting the dominant objects present in an image from a set of 1000 categories.
- **DeeplabV3** - for segmenting the pixels of a camera frame or image into a predefined set of classes..
- **ResNet50** - for detecting objects from a set of 1000 categories such as trees, animals.
- **YOLOv3** - for classifying 80 different types of objects

While some of the ML models listed above have the same purpose, the detection accuracy varies. In this demo, we will use the *MobileNetV2* model. So, download the model from https://docs-assets.developer.apple.com/coreml/models/Image/ImageClassification/MobileNetV2/MobileNetV2.mlmodel and save it to your preferred folder.

Preparing the Starter Project

We will not build the app demo from scratch. Instead, I have created a starter project with a prebuilt user interface, so we can focus on discussing the integration of Core ML models.

First, download the starter project from http://www.appcoda.com/resources/swift5/ImageRecognitionStarter.zip. Once you open the project in Xcode, try to build and deploy it on your iPhone. The current project is very similar to what you built in chapter 11 and 13. When you run the app, it will ask you for permission to access the device's camera. Please make sure you grant the permission. For now, other than showing the live image feed, the app does nothing.

This is what we are going to implement that the app will analyze the object you point at and predict what it is. The result may not be perfect, but you will get an idea how you can apply Core ML in your app.

Importing the Core ML Model

Recalled that you have downloaded the *MobileNetV2* model from Apple, you will need to import it into the Xcode project in order to use. To import the Core ML model, all you need to do is drag the `MobileNetV2.mlmodel` file to the project navigator. To better organize your project, I recommend you to create a group named `MLModel` (or any other name you prefer) and put the file under that group.

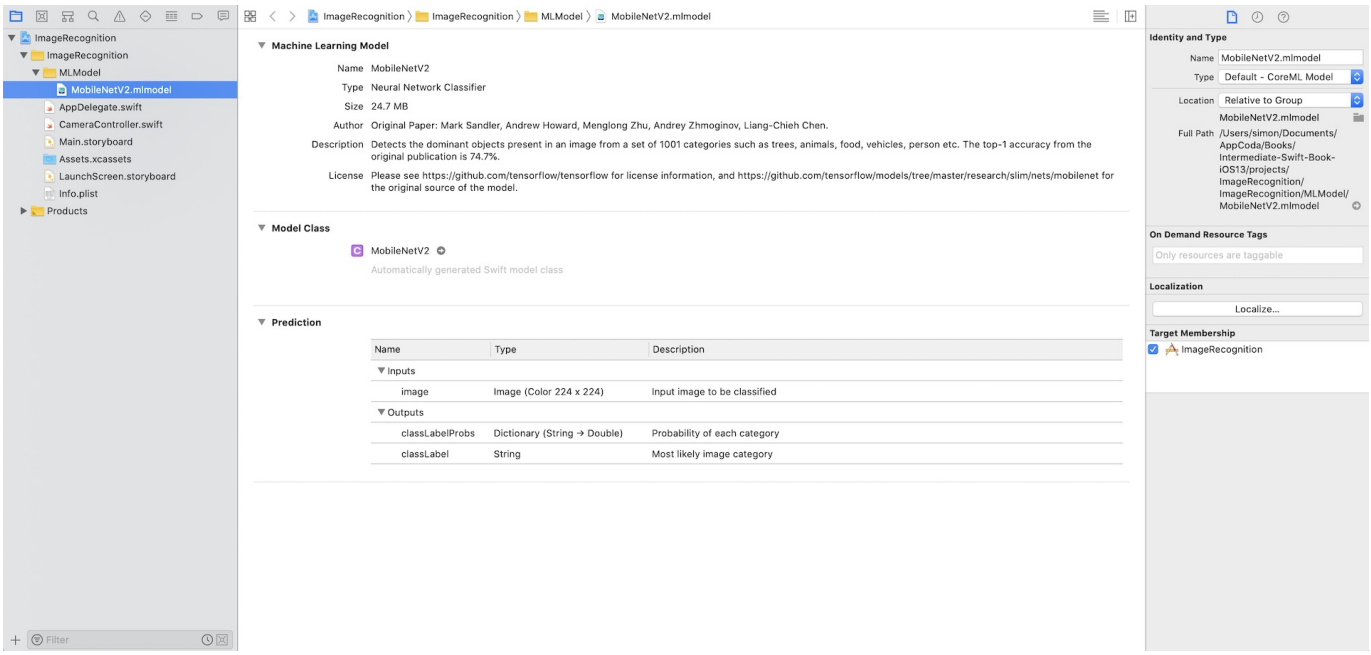


Figure 40.4. MobileNetV2.mlmodel

Once you imported the model, select the `MobileNetV2.mlmodel` file to reveal its details including the model type, author, description, and license. The *Model Class* section shows you the name of the Swift class for this model. Later, you can instantiate the model object like this:

```
let model = MobileNetV2()
```

The *Model Evaluation Parameters* section describes the input and output of the model. Here, the Core ML model takes an image of the size 224x224 as an input and gives you two outputs:

1. The most likely image category, which is the best guess of the object identified in the given image.
2. A dictionary containing all the possible predictions and the corresponding probability. Figure 40.5 illustrates what this dictionary is about.

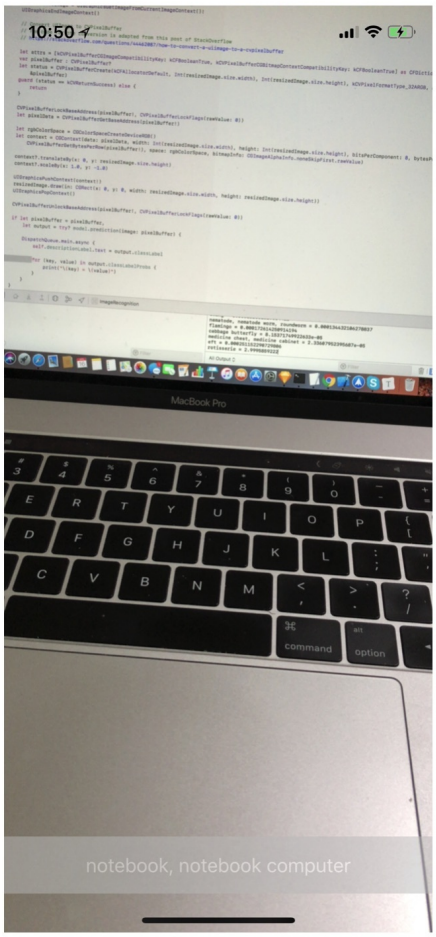


Figure 40.5. The possible predictions and its corresponding probability

Implementing the Real-time Image Recognition

Now that you have a basic idea about how to use a Core ML model, let's talk about the implementation. The starter project already implements the camera access to capture real-time videos. As you remember, the Core ML model accepts a still image as the input. Therefore, in order to use the model for image recognition, we have to provide the following implementation:

1. Processes the video frames and turn them into a series of still images such that they conform to the requirement of the Core ML model. In other words, the images should have the width of 224 and the height of 224.
2. Passes the images to the Core ML model for predictions.

3. Displays the most likely answer on screen.

Okay, it is time to turn the description above into code.

First, open `CameraController.swift` and insert the following code snippet after `captureSession.addInput(input)` in the `configure()` method:

```
let videoDataOutput = AVCaptureVideoDataOutput()
videoDataOutput.setSampleBufferDelegate(self, queue: DispatchQueue(label: "imageRe
cognition.queue"))
videoDataOutput.alwaysDiscardsLateVideoFrames = true
captureSession.addOutput(videoDataOutput)
```

The starter project only defines the input of the capture session. To process the video frames, we first create an object of `AVCaptureVideoDataOutput` to access the frames. And then we set the delegate for the video sample buffer such that every time when a new video sample buffer is received, it will be sent to the delegate for further processing. In the code above, the delegate is set to `self` (i.e. `CameraController`). After the video data output is defined, we add it to the capture session by invoking `addOutput`.

Meanwhile, Xcode should show you an error because we haven't implemented the sample buffer delegate which should conform to the `AVCaptureVideoDataOutputSampleBufferDelegate` protocol. We will use an extension to adopt it like this:

```
extension CameraController: AVCaptureVideoDataOutputSampleBufferDelegate {
    func captureOutput(_ output: AVCaptureOutput, didOutput sampleBuffer: CMSample
Buffer, from connection: AVCaptureConnection) {
        connection.videoOrientation = .portrait

        // Resize the frame to 224x224
        // This is the required size of the inceptionv3 model
        guard let imageBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) else {
            return
        }

        let ciImage = CIImage(cvPixelBuffer: imageBuffer)
        let image = UIImage(ciImage: ciImage)

        UIGraphicsBeginImageContext(CGSize(width: 224, height: 224))
        image.draw(in: CGRect(x: 0, y: 0, width: 224, height: 224))
        let resizedImage = UIGraphicsGetImageFromCurrentImageContext()!
        UIGraphicsEndImageContext()
    }
}
```

When a new sample video buffer is received, the `captureOutput(_:didOutput:from:)` method of the delegate is called. In the code above, we retrieve the video frame data, create an image from the data, and then convert the image to the required size (i.e. 224x224).

With the processed image, we can now create the ML model and pass the image to the model. First, declare a variable in the `CameraController` class to hold the ML model:

```
var mlModel = MobileNetV2()
```

If you look into the `MobileNetV2` class, you will find a method called `prediction(image:)`. This is the method we will use for identifying the object in a given image.

```
func prediction(image: CVPixelBuffer) throws -> MobileNetV2Output {
    let input_ = MobileNetV2Input(image: image)
    return try self.prediction(input: input_)
}
```

If you look even closer, the parameter `image` has the type `CVPixelBuffer`. Therefore, in order to use this method, we have to convert the resized image from `UIImage` to `CVPixelBuffer`. Insert the following code after `UIGraphicsEndImageContext()` to perform the conversion:

```
// Convert UIImage to CVPixelBuffer
// The code for the conversion is adapted from this post of StackOverflow
// https://stackoverflow.com/questions/44462087/how-to-convert-a-uiimage-to-a-cvpixelbuffer

let attrs = [kCVPixelBufferCGImageCompatibilityKey: kCFBooleanTrue, kCVPixelBufferCGBitmapContextCompatibilityKey: kCFBooleanTrue] as CFDictionary
var pixelBuffer : CVPixelBuffer?
let status = CVPixelBufferCreate(kCFAllocatorDefault, Int(resizedImage.size.width), Int(resizedImage.size.height), kCVPixelFormatType_32ARGB, attrs, &pixelBuffer)
guard (status == kCVReturnSuccess) else {
    return
}

CVPixelBufferLockBaseAddress(pixelBuffer!, CVPixelBufferLockFlags(rawValue: 0))
let pixelData = CVPixelBufferGetBaseAddress(pixelBuffer!)

let rgbColorSpace = CGColorSpaceCreateDeviceRGB()
let context = CGContext(data: pixelData, width: Int(resizedImage.size.width), height: Int(resizedImage.size.height), bitsPerComponent: 8, bytesPerRow: CVPixelBufferGetBytesPerRow(pixelBuffer!), space: rgbColorSpace, bitmapInfo: CGImageAlphaInfo.noneSkipFirst.rawValue)

context?.translateBy(x: 0, y: resizedImage.size.height)
context?.scaleBy(x: 1.0, y: -1.0)

UIGraphicsPushContext(context!)
resizedImage.draw(in: CGRect(x: 0, y: 0, width: resizedImage.size.width, height: resizedImage.size.height))
UIGraphicsPopContext()

CVPixelBufferUnlockBaseAddress(pixelBuffer!, CVPixelBufferLockFlags(rawValue: 0))
```

Once you converted the image to `CVPixelBuffer`, you can pass it to the Core ML model for predictions. Continue to insert the following code in the same method:

```
if let pixelBuffer = pixelBuffer,
    let output = try? mlModel.prediction(image: pixelBuffer) {

    DispatchQueue.main.async {
        self.descriptionLabel.text = output.classLabel
    }
}
```

We call the `prediction(image:)` method to predict the object in the given image. The best possible answer is stored in the `classLabel` property of the output. We then set the class label to the description label.

That's how we implement the real-time image recognition. Now it is ready to test the app. Build the project and deploy it on a real iPhone. After the app is launched, point it to a random object. The app should show you what the object is. The detected result may not be correct because the ML model was trained to detect objects from a set of 1000 categories. That said, you should have an idea about how to integrate a ML model in your app.

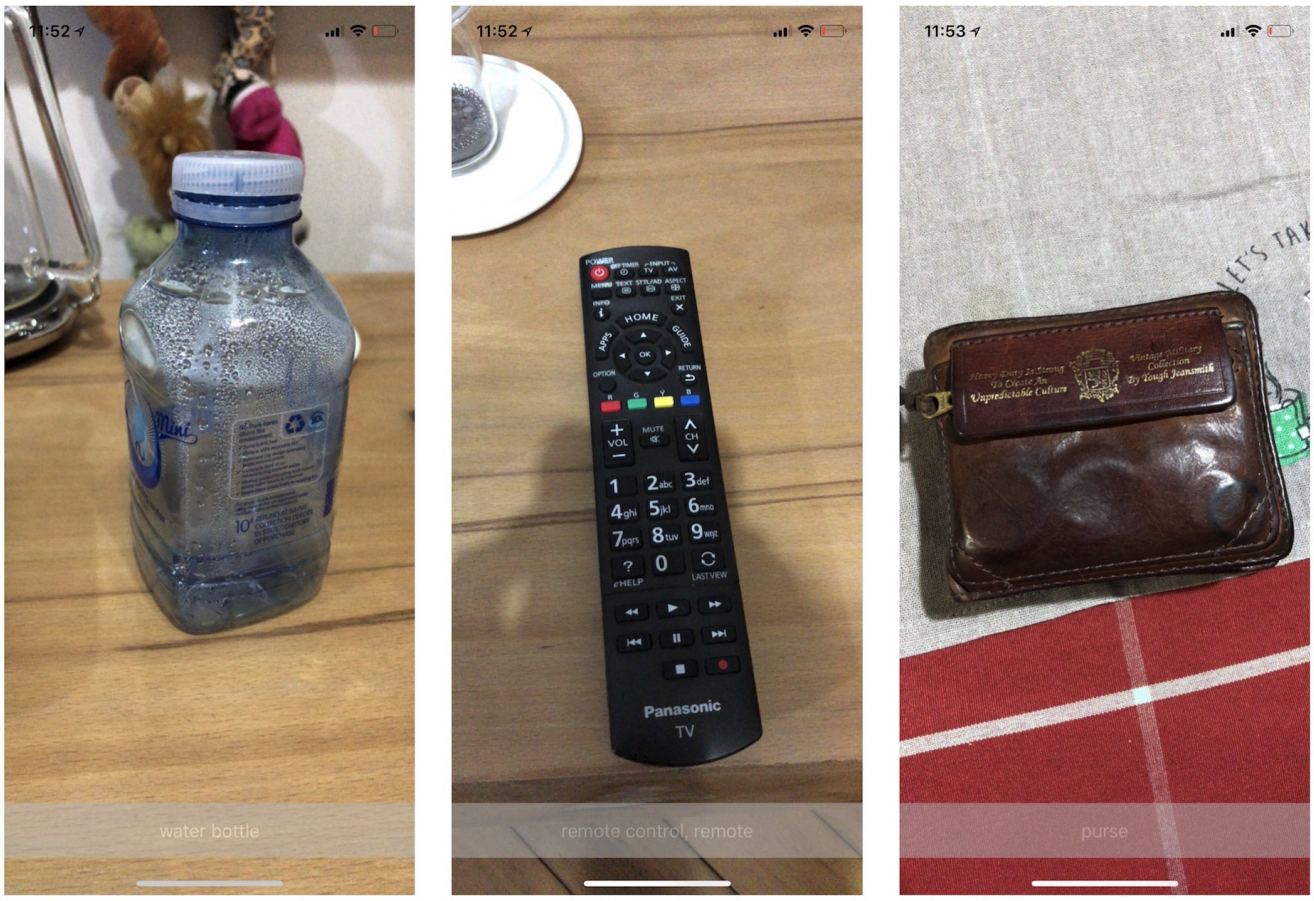


Figure 40.6. Image Recognition Demo

There is one last thing I want to mention. Other than `classLabel`, you can access the values of `classLabelProbs` to retrieve all the possible guesses. Each of the predictions is associated with the corresponding probability. Insert the following code in the `DispatchQueue.main.async` block:

```
for (key, value) in output.classLabelProbs {
    print("\(key) = \(value)")
}
```

Test the app again. You should see all the predictions of the object you are pointing at, each with the probability for your reference, in the console message.

```
coil, spiral, volute, whorl, helix = 0.00494262157008052
hot pot, hotpot = 0.000246078736381605
beer bottle = 0.000806860974989831
half track = 0.000261949375271797
water snake = 0.000204006355488673
balloon = 0.022151293233037
```

Summary

I hope you enjoyed reading this chapter and now understand how to integrate Core ML in your apps. This is just a brief introduction to Core ML. If you are interested in training your own model, take a look at the following free tutorials:

- A Beginner's Guide to Core ML Tools: Converting a Caffe Model to Core ML Format (https://www.appcoda.com/core-ml-tools-conversion)
- Creating a Custom Core ML Model Using Python and Turi Create (https://www.appcoda.com/core-ml-model-with-python)

For reference, you can download the complete Xcode project from http://www.appcoda.com/resources/swift5/ImageRecognition.zip.

Chapter 41
Building AR Apps with ARKit and SpriteKit

First things first, what is Augmented Reality? In brief, this means you can place virtual objects in a real-world environment. The very best example of AR application is Pokemon Go! At the time, this well-known game was not developed using ARKit, but it showcases one of the many applications of augmented reality.

Figure 41.1. Pokemon Go with AR enabled (images from Niantic Labs)

Another great example of AR application is the ARMeasure app (http://armeasure.com/) that allows users to measure nearly anything. You no longer need a physical ruler to measure an object, whether it's a suitcase or a photo frame hanging on a wall. Just point your iPhone's camera to the object you want to measure and the virtual ruler will do the measurement for you. If you haven't tried the app, I recommend you to check it out and play around with it.

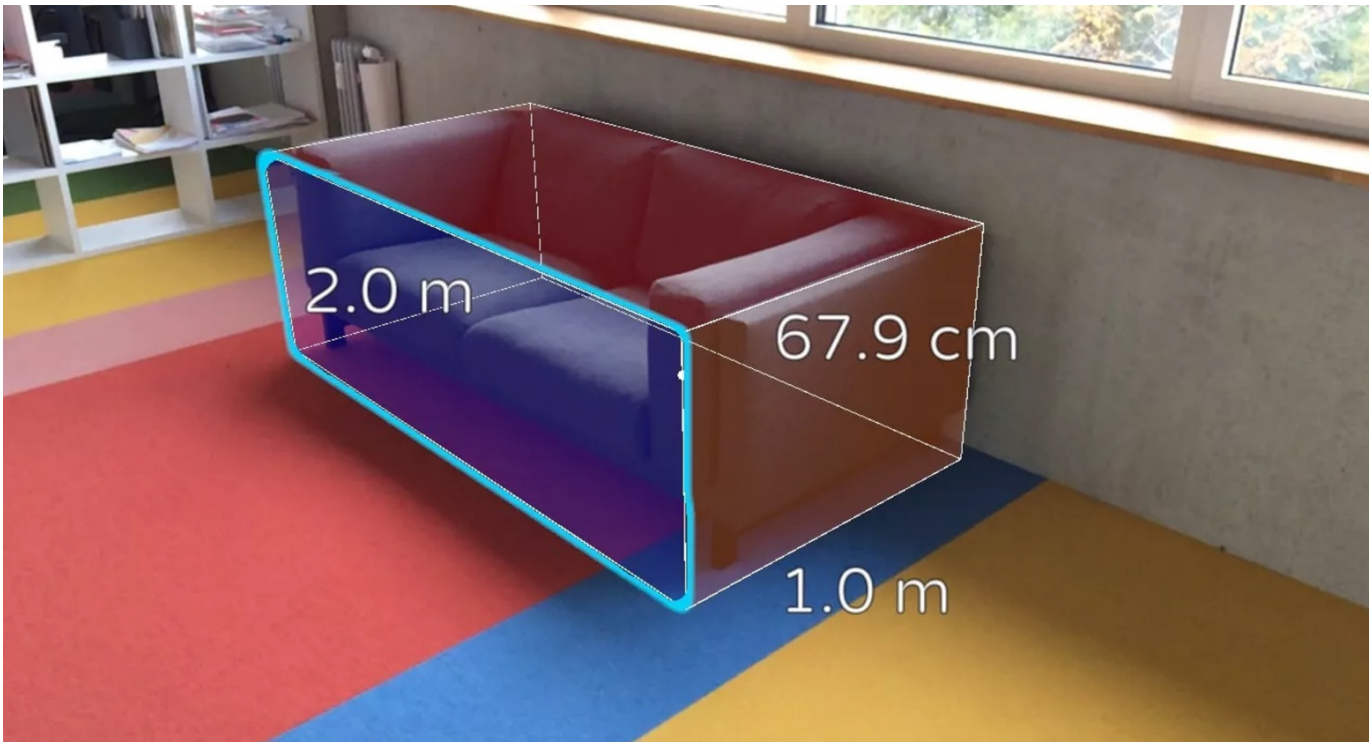


Figure 41.2. ARMeasure app

Amazing, right? There is no shortage of AR applications. Some AR applications integrate with other technologies to take the user experience to a whole next level. One example is the ARKit + CoreLocation application (https://github.com/ProjectDent/ARKit-CoreLocation), created by Andrew Hart.

While Google Maps or other map applications can show you the direction from point A to point B, the app demonstrates a whole different experience to show landmarks and directions by combining the power of AR and Core Location technologies. What's that building at the end of the street? Simply point the camera at that building, the app will give you the answer by annotating the landmark. Need to know how to get from here to there? The app shows you turn-by-turn directions displayed in augmented reality. Take a look at the figures below and check out the demo at https://github.com/ProjectDent/ARKit-CoreLocation. You will know what this app does and understand the power of combining AR and other technologies.

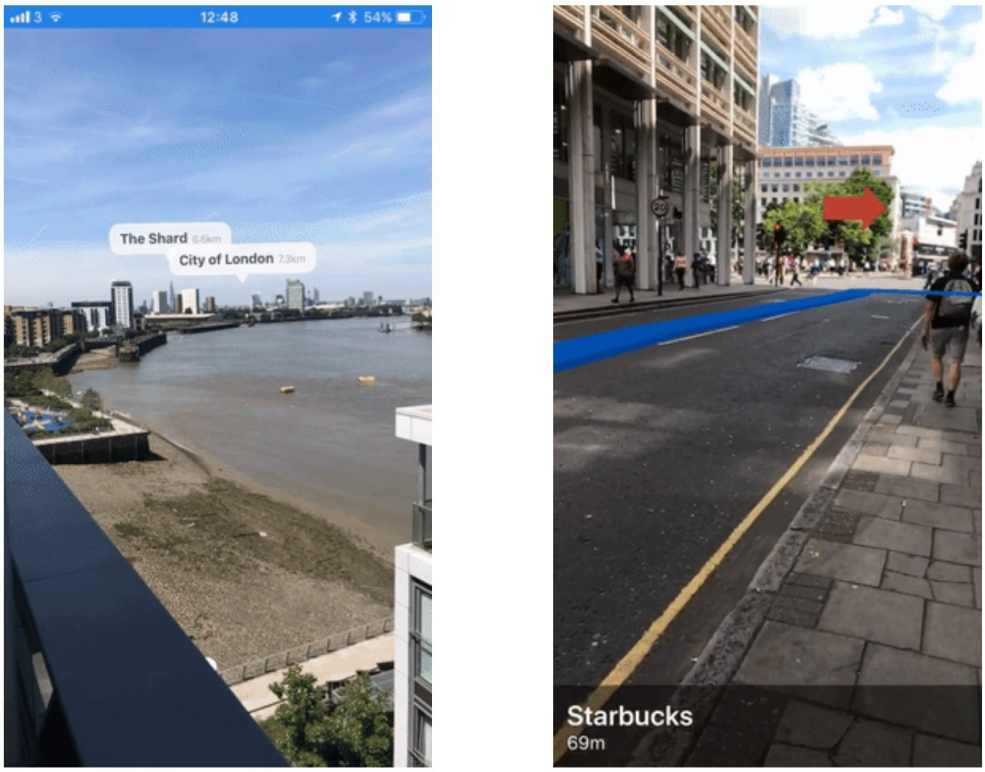


Figure 41.3. ARKit + CoreLocation application

Now that you have some basic ideas of AR, let's see how to build an AR app. We have already mentioned the term ARKit. It is the new framework introduced in iOS 11 for building AR apps on iOS devices. Similar to all other frameworks, it comes along with Xcode. As long as you use Xcode 11 (or up), you will be able to develop ARKit apps.

Before we dive into ARKit, please take note that ARKit app can only run on the following devices, equipped with A9 processor (or up):

1. iPhone 6s, 6s Plus, 7, 7 Plus, 8, 8 Plus, iPhone X/Xs/Xr, and iPhone 11/11 Pro/11 Pro Max
2. iPad Pro, iPad mini (5th generation), iPad Air (3rd generation), and iPad (5th generation)
3. iPhone SE

You can't test ARKit apps by using the built-in simulators. You have to use one of the compatible devices as listed above for testing. Therefore, try to prepare the device, otherwise, you can't test your app.

In this chapter, I'll give you a brief introduction to ARKit, which is the core framework for building AR apps on iOS. At the end of this chapter, you will walk away with a clear understanding of how ARKit works and how to accomplish the following using SpriteKit:

- Adding 2D objects into the real-world space
- Removing 2D objects from the space
- Interacting with virtual objects

Building Your First ARKit App

With the built-in template, Xcode has made it very easy for developers to create their first ARKit app. You don't even need to write a line of code. Let's have a try and you will understand what I mean in a minute.

Now fire up Xcode and choose to create a new project. In the project template, select the *Augmented Reality App* template.

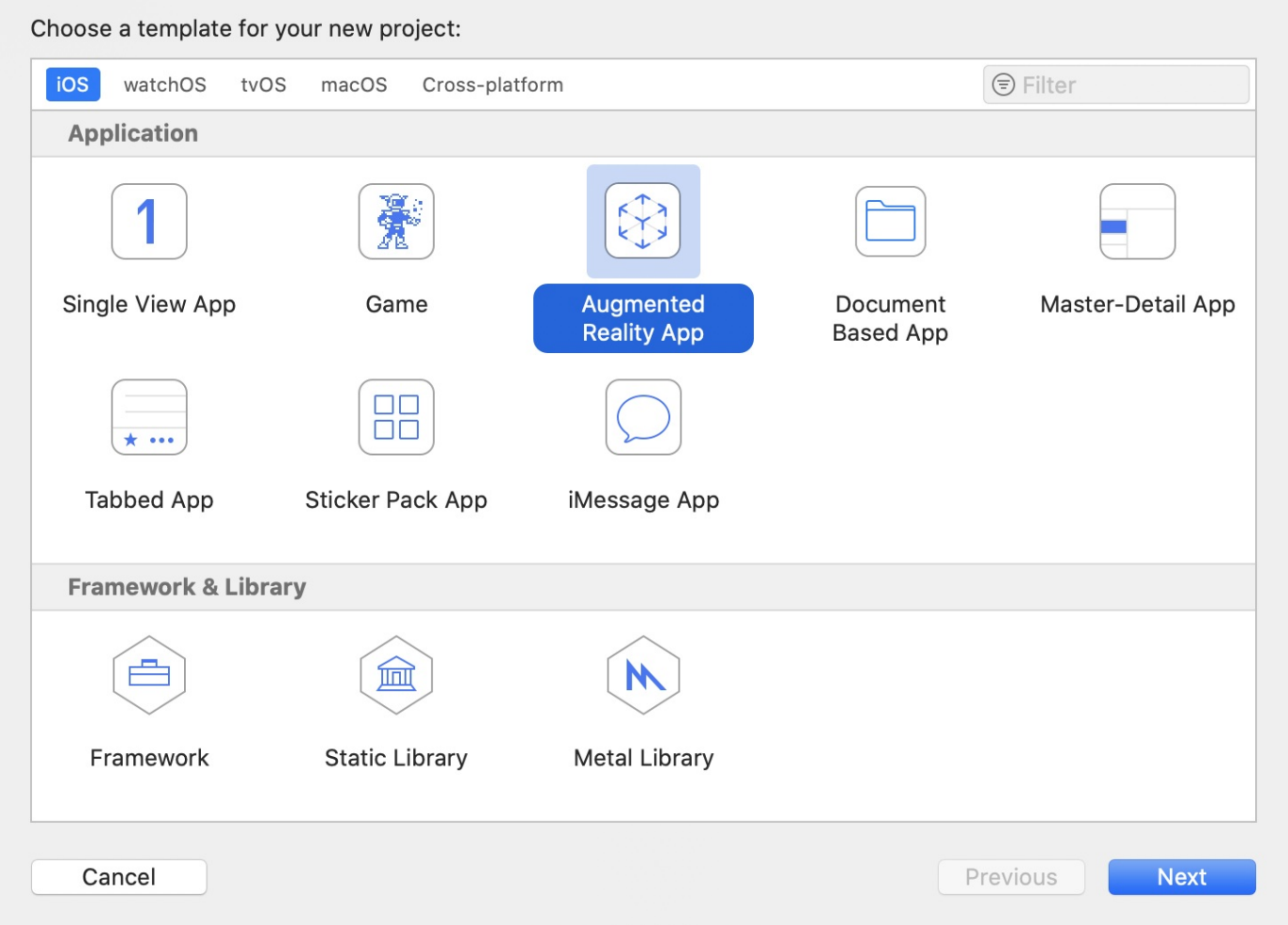


Figure 41.4. Choosing the Augmented Reality App template

In the next screen, fill in the project information. You can name the product "ARKitDemo", but please make sure you use an organization identifier different from mine. This is to ensure that Xcode generates an unique bundle identifier for you. The *Content Technology* option should be new to you. By default, it is set to SpriteKit.

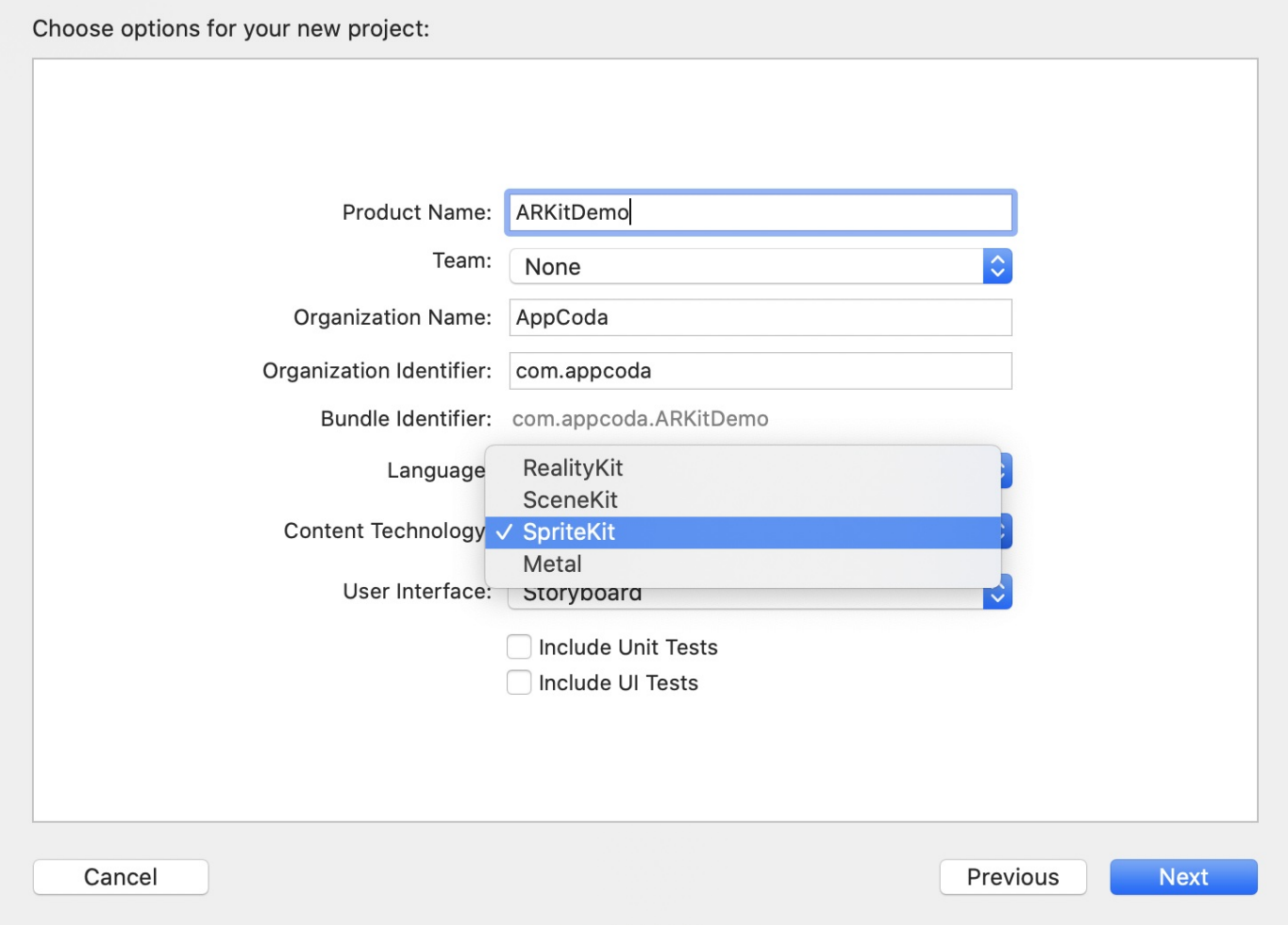


Figure 41.5. Using SpriteKit as the content technology

You probably have heard of SpriteKit, which is a framework for making 2D games, but wonder why is it here. Are we going to build a 2D game or an ARKit app?

One of the coolest things about ARKit is that it integrates well with Apple's existing graphics rendering engines. SpriteKit is one of the rendering engines. You can also use SceneKit for rendering 3D objects or Metal for hardware-accelerated 3D graphics. For our first ARKit app, let's use SpriteKit as the content technology.

Once you save the project, you are ready to deploy the app to your iPhone. You don't need to change a line of code. The AR app template already generates the code you need for building your first ARKit app. Connect your iPhone to your Mac and hit the Run button

to try out the ARKit app. It's a very simple app. Every time you tap on the screen, it displays an emoji character in augmented reality.

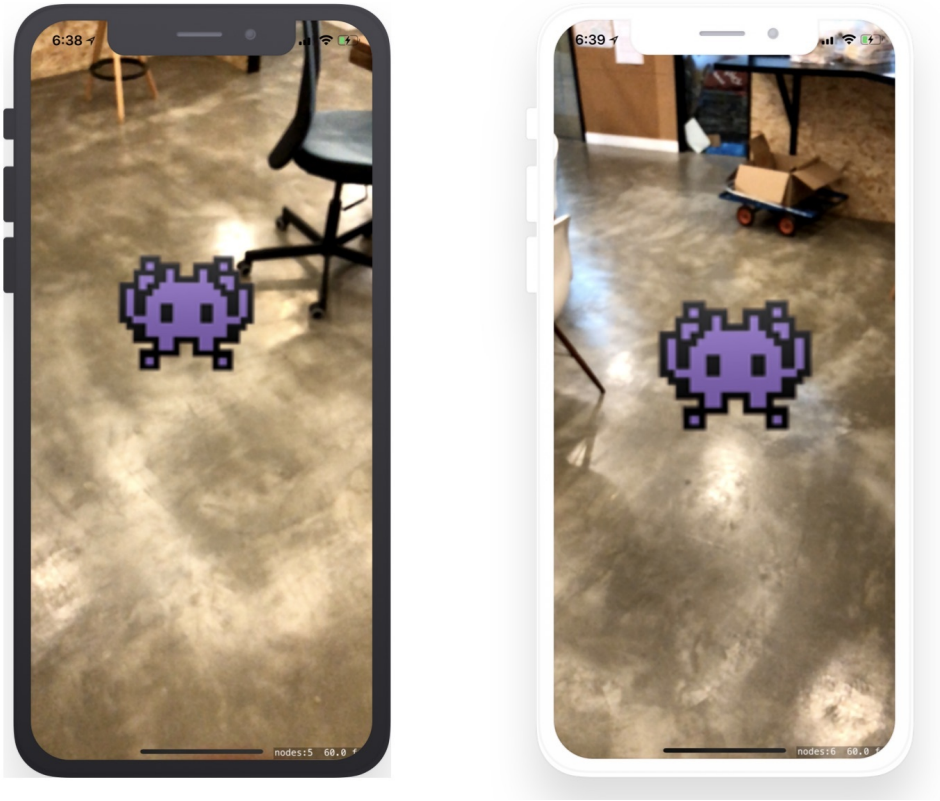


Figure 41.6. Your first ARKit app

Understanding ARKit and the Sample Project

Cool, right? Now let's go back to the implementation and see how the sample code works.

Augmented reality is an illusion that the virtual objects are part of the real world. To make this possible, ARKit uses a technique known as *Visual Inertial Odometry (VIO)*. When you point iPhone's camera in the real-world environment, ARKit analyzes both the camera sensor data and motion data to track the world around it. This allows developers to display virtual content in a real-world environment.

In doing so, ARKit maintains a virtual 3D coordinate system of the world. By placing virtual content on this coordinate space and combining the content with a live camera image, it gives an illusion that the virtual content is part of the real world (i.e. Augmented Reality).

ARKit Core Classes

To understand how ARKit works and how to use it, there are a number of core classes you should know.

- **ARSKView / ARSCNView** - Depending on the content technology you choose, you use one of these two classes to render the AR experiences that position the virtual content in 3D space within the device's camera view. If you use SpriteKit, you use `ARSKView` to create 2D objects in real world. For rendering 3D objects, you will need to use SceneKit and `ARSCNView` is the view for rendering 3D objects. Whether you use `ARSKView` or `ARSCNView`, the view automatically renders the live video feed from the camera as the scene background.
- **ARSession** - Every AR experience created by ARKit requires an `ARSession` object. The view has an `ARSession` object associated with it. This session object is responsible for managing motion data and camera image processing for the view's contents. It analyzes the captured images and synthesizes all these data to create the AR experience.
- **ARFrame** - When an ARSession is activated, it continuously captures video frames using the camera. Each frame is analyzed with the position-tracking information. All these data are stored in the form of `ARFrame`. You can access the most-recently captured frame image by using the `currentFrame` property of an ARSession.
- **ARWorldTrackingConfiguration** - Similar to other session objects (e.g. `URLSession`) you have worked with the iOS SDK, you have to provide an `ARSession` object with a session configuration. There are three different configurations built into ARKit. `ARWorldTrackingConfiguration` is one of the built-in configurations that tracks a device's position and motion relative to the real world by using the rear camera. It also supports plane detection and hit testing. The class tracks the

movement of the device with six-degrees of freedom (6DOF). That means the three rotation axes (roll, pitch, and yaw), and three translation axes (movement in x, y, and z). For devices with A9 processors (or up), they all support 6DOF. Earlier devices can only support three-degrees of freedom, which are the three rotation axes.

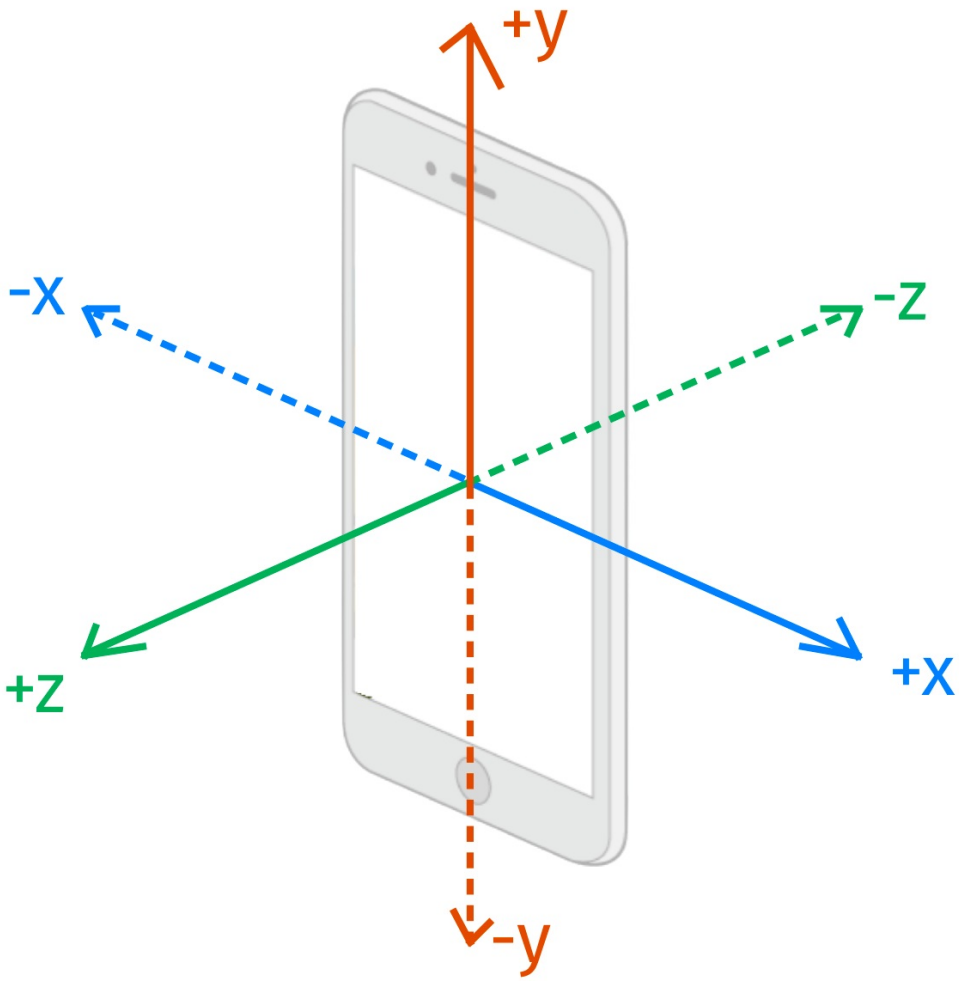


Figure 41.7. Three translation axes

In most cases, you do not need to tweak the tracking configuration. But I have to mention a property named `worldAlignment`. It controls how ARKit creates a scene coordinate system based on real-world device motion with the following possible options:

- **gravity** - By default, it is set to `gravity`. This AR coordinate system is mapped

to the real-world space that the y-axis matches the direction of gravity (see figure 41.7). For most AR experience, this setting should be used.
    - **gravityAndHeading** - it is pretty similar to `gravity` but the axes are fixed to north/south and east/west. For `gravity`, x and z directions are relative to the device's *original* heading. When `gravityAndHeading` is used, the X and Z axes are oriented to the compass heading. While the direction of the axes is related to the real-world directions, the *location* of the ARKit coordinate system's origin is still relative to the device.
    - **camera** - All axes are locked to match the orientation of the camera when `worldAlignment` is set to `camera`. For example, if you run the demo app with `worldAlignment` sets to this option, the emoji character will rotate when you turn your device sideway.

- **ARAnchor** - `ARAnchor` automatically tracks the positions and orientations of real or virtual objects relative to the camera. To place a virtual object in an AR environment, you use `ARAnchor`. You create an `ARAnchor` object and call its `add(anchor:)` method to add the object to the AR session.

How the Demo App Works

Now that you have some basic understanding of the core classes, let's dive into the demo project and see how it actually works.

First, let's look into `Main.storyboard`. This view controller was constructed by Xcode since we chose to use the ARKit app template at the very beginning. In the view controller, you should find an `ARSKView` because we selected to use SpriteKit as the content technology. This view renders the live video feed from the camera as the scene background.

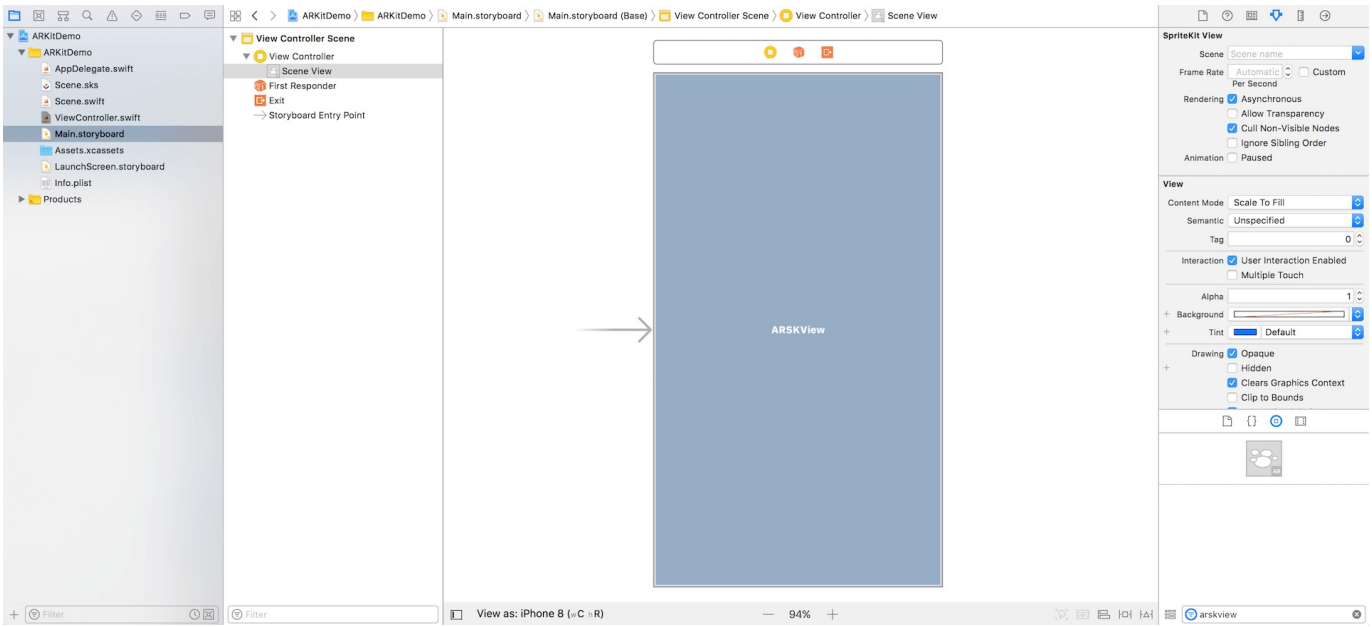


Figure 41.8. ARSKView in the view controller

Now, let's take a look at the `ViewController` class. In the `viewWillAppear` method, we instantiate `ARWorldTrackingConfiguration` and start creating the AR experience by using the configuration.

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    // Create a session configuration
    let configuration = ARWorldTrackingConfiguration()

    // Run the view's session
    sceneView.session.run(configuration)
}
```

Once you call the run method, the AR session runs asynchronously. So, how was the emoji icon added to the augmented reality environment?

As mentioned before, we use SpriteKit as the content technology. The power of ARKit is that it automatically matches the coordinate space of SpriteKit to the real world. Therefore, if you are familiar with the framework, it is pretty easy to place 2D content in

the real world. In the `Scene.swift` file, we create our own scene by inheriting from `SKScene`. To add a virtual object in the AR space, we override the `touchesBegan` method and add an `ARAnchor` object whenever the user taps on the screen:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let sceneView = self.view as? ARSKView else {
        return
    }

    // Create anchor using the camera's current position
    if let currentFrame = sceneView.session.currentFrame {

        // Create a transform with a translation of 0.2 meters in front of the cam
era
        var translation = matrix_identity_float4x4
        translation.columns.3.z = -0.2
        let transform = simd_mul(currentFrame.camera.transform, translation)

        // Add a new anchor to the session
        let anchor = ARAnchor(transform: transform)
        sceneView.session.add(anchor: anchor)
    }
}
```

As I have walked you through the core classes of ARKit, the code above is self-explanatory. However, you may be confused about the matrix operation especially if you totally forgot what you learned in college.

The goal is to place a 2D object 0.2 meters in front of the device's camera. Figure 41.9 illustrates where the emoji icon is going to position.

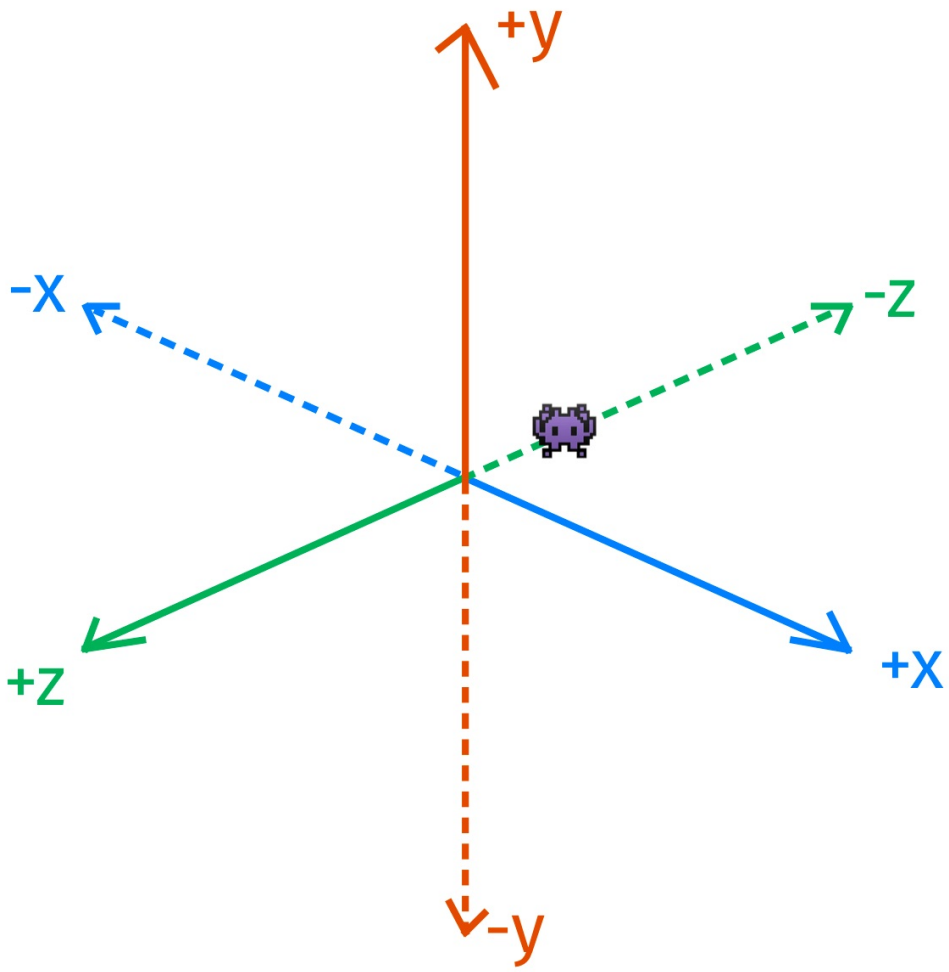


Figure 41.9. Positioning the emoji icon in front of the camera

In order to do that, you have to refresh your knowledge of matrices. The transformation of an object in 3D space is represented by a 4x4 matrix. To move a point in 3D space, it can be calculated using the following equation:

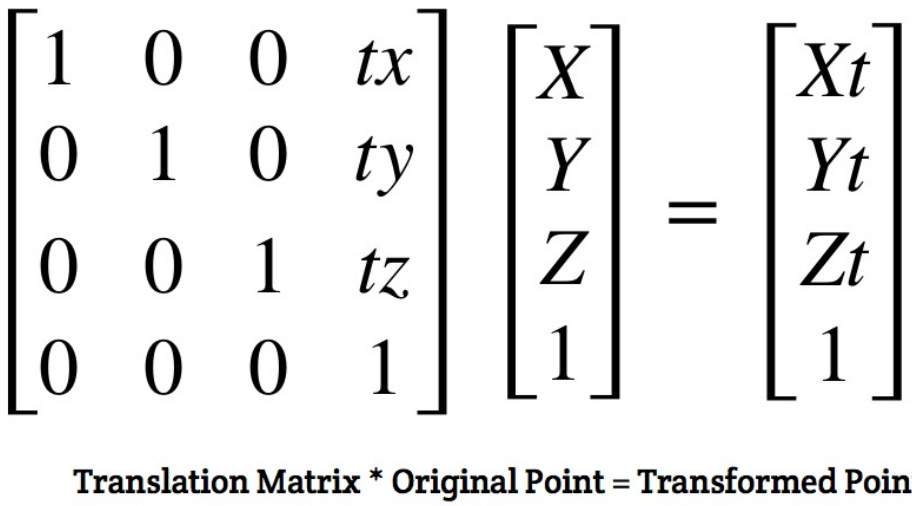


Figure 41.10. Translation Matrices

To create a translation matrix, we first start with an identity matrix (see figure 41.11). In the code, we use the constant `matrix_identity_float4x4`, which is the representation of an identity matrix in Swift.

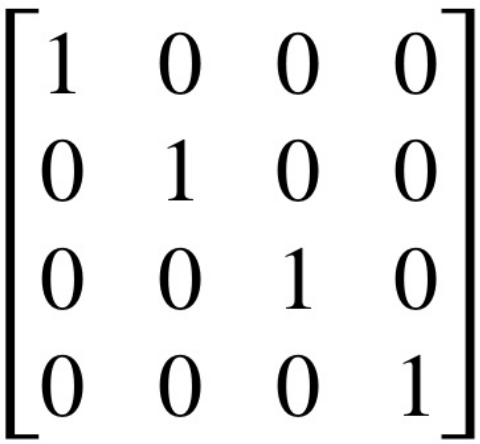


Figure 41.11. Identity matrix

In order to place the object 0.2m in front of the camera, we have to create a translation matrix like this:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 41.12. Translation matrix

In Swift, we write the code like this:

```
translation.columns.3.z = -0.2
```

Since the first column has an index of `0`, we change the `z` property of the column with index `3`.

With the translation matrix in place, the final step is to compute the transformed point by multiplying the original point (i.e. `currentFrame.camera.transform`) with the translation matrix. In code, we write it like this:

```
let transform = simd_mul(currentFrame.camera.transform, translation)
```

`simd_mul` is a built-in function for performing matrix multiplication. Once we calculated the target position (i.e. `transform`), we create an `ARAnchor` object and add it to the AR session.

```
let anchor = ARAnchor(transform: transform)
sceneView.session.add(anchor: anchor)
```

Now, let's go back to the `ViewController.swift` file. The `ViewController` class, which adopts the `ARSKViewDelegate` protocol, is the delegate of the scene view. You have to implement the protocol in order to provide the SpriteKit content. Whenever you add a new `ARAnchor` object, the following method of the `ARSKViewDelegate` protocol will be called:

```
optional func view(_ view: ARSKView, nodeFor anchor: ARAnchor) -> SKNode?
```

Therefore, we implement the method to provide the 2D content, which is a label node filling with an Emoji icon:

```
func view(_ view: ARSKView, nodeFor anchor: ARAnchor) -> SKNode? {
    // Create and configure a node for the anchor added to the view's session.
    let labelNode = SKLabelNode(text: "  ")
    labelNode.horizontalAlignmentMode = .center
    labelNode.verticalAlignmentMode = .center
    return labelNode;
}
```

Cool! This is pretty much how the ARKit demo app works.

Interacting with AR Objects

Now that you have a basic idea about how ARKit works, let's tweak the demo application a bit. When a user taps the emoji character, it will fade out and eventually disappear.

If you have some experience with SpriteKit, you know how to detect a touch. Update the `touchesBegan` method like this:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    guard let sceneView = self.view as? ARSKView else {
        return
    }

    // Check if the user touches a label node
    if let touchLocation = touches.first?.location(in: self) {
        if let node = nodes(at: touchLocation).first as? SKLabelNode {

            let fadeOut = SKAction.fadeOut(withDuration: 1.0)
            node.run(fadeOut) {
                node.removeFromParent()
            }

            return
        }
    }

    // Create anchor using the camera's current position
    if let currentFrame = sceneView.session.currentFrame {

        // Create a transform with a translation of 0.2 meters in front of the cam
era
        var translation = matrix_identity_float4x4
        translation.columns.3.z = -0.2
        let transform = simd_mul(currentFrame.camera.transform, translation)

        // Add a new anchor to the session
        let anchor = ARAnchor(transform: transform)
        sceneView.session.add(anchor: anchor)
    }
}
```

We have added an additional block of code to check if the user taps on the emoji character. The following line of code retrieves the location of the touch:

```
if let touchLocation = touches.first?.location(in: self) {
```

Once we get the touch location, we verify if it hits the node with the emoji character:

```
if let node = nodes(at: touchLocation).first as? SKLabelNode {
```

If it does tap the emoji character, we execute a "fade out" animation and finally remove the node from the view.

As you can see, the whole code snippet is related to SpriteKit. As long as you have the knowledge of SpriteKit, you would know how to interact with the objects in AR space.

Exercise #1

Instead of displaying an Emoji character, tweak the demo app to show an image. You are free to use your own image. Alternatively, you can find tons of free game characters on opengameart.org. Personally, I used the fat bird (https://opengameart.org/content/fat-bird-sprite-sheets-for-gamedev) for this exercise. Your resulting screen should look like the one shown in the figure below:

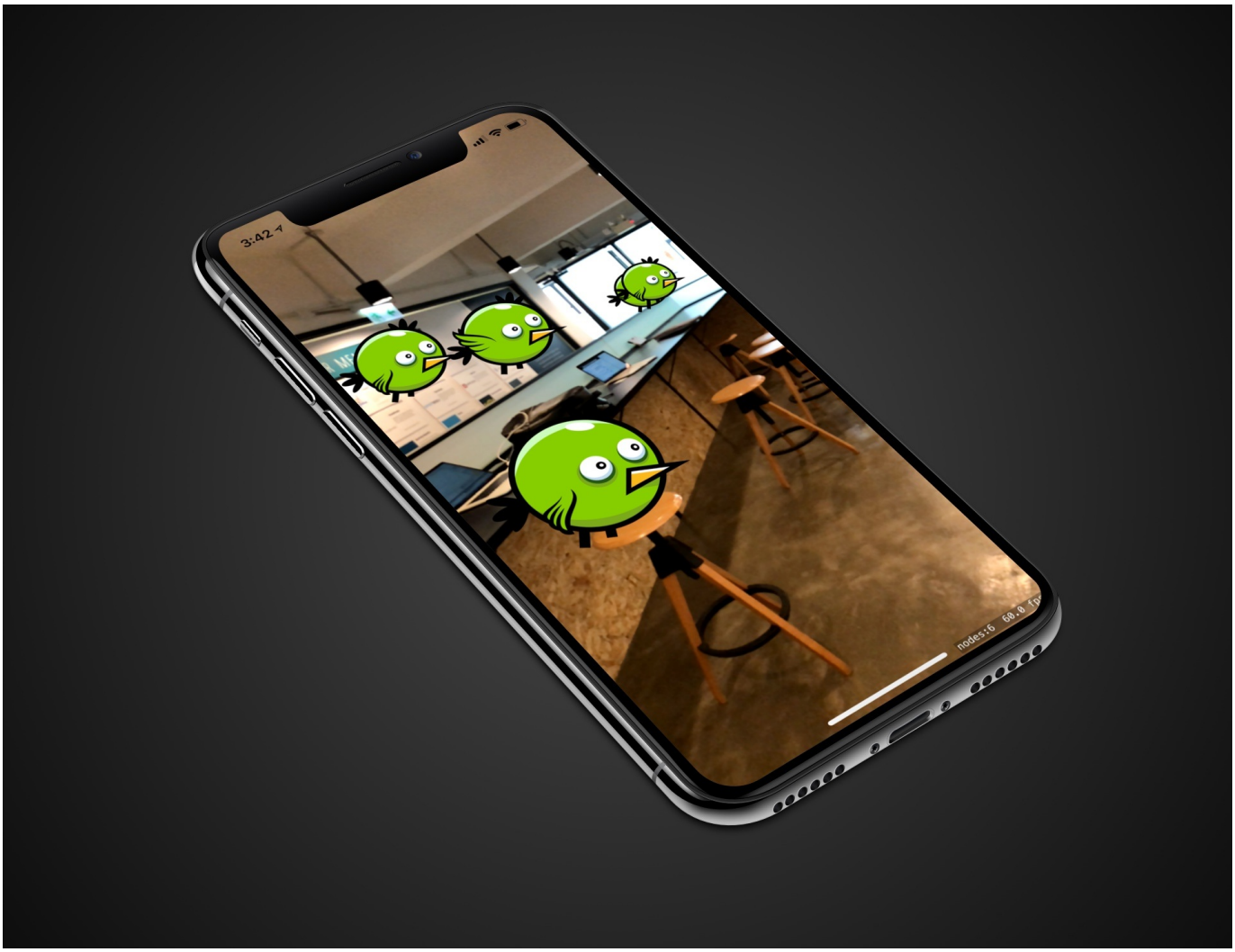


Figure 41.13. Displaying an image instead of an emoji character

If you are not very familiar with SpriteKit, here are a couple of tips:

1. We use `SKLabelNode` to create a label node. For images, you can use `SKSpriteNode` and `SKTexture` to load the image. Here is an example:

```
let texture = SKTexture(imageNamed: "fat-bird")
let spriteNode = SKSpriteNode(texture: texture)
```

2. To resize a sprite node, you can change its `size` property like this:

```
spriteNode.size = CGSize(width: spriteNode.size.width * 0.1, height: spriteNode.si
ze.height * 0.1)
```

Exercise #2

Let's continue to tweak the demo app to create a world having different types of birds. Assuming you've followed exercise #1, your app should now display a bird image when the user taps on the device's screen. Now you are required to implement the following enhancements:

1. First, add three more types of birds to the demo app. You can download the images of the birds using the link below:

   - https://opengameart.org/content/game-character-blue-flappy-bird-sprite-sheets
   - https://opengameart.org/content/flappy-grumpy-bird-sprite-sheets
   - https://opengameart.org/content/flappy-monster-sprite-sheets

   Tweak the demo app such that it randomly picks one of the bird images and shows it on screen whenever the user taps the device's screen.

2. If you look into the image packs you just downloaded, all come with a set of 8 images. By combining the set of images together, you can create a flying animation. As a hint, here is the sample code for creating the animation:

```
let flyingAction = SKAction.repeatForever(SKAction.animate(with: birdFrames, timeP
erFrame: 0.1))
```

This exercise is more difficult than the previous one. Take some time to work on it. Figure 41.14 shows a sample screenshot of the complete app.

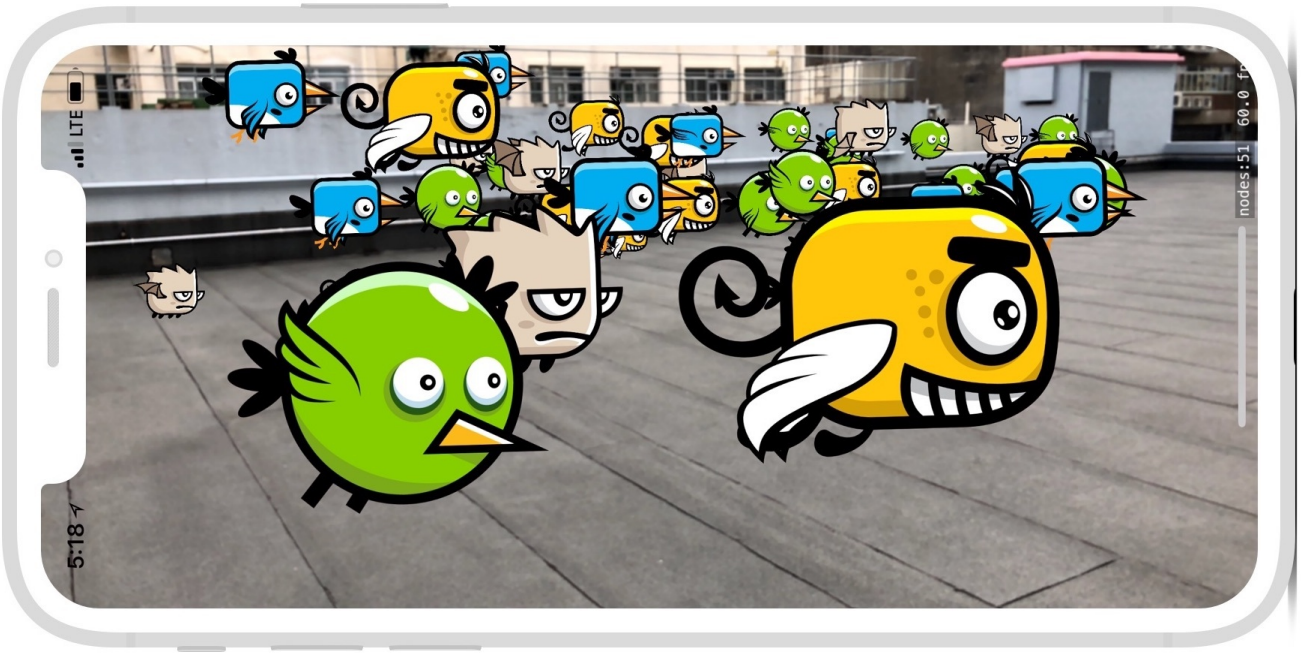
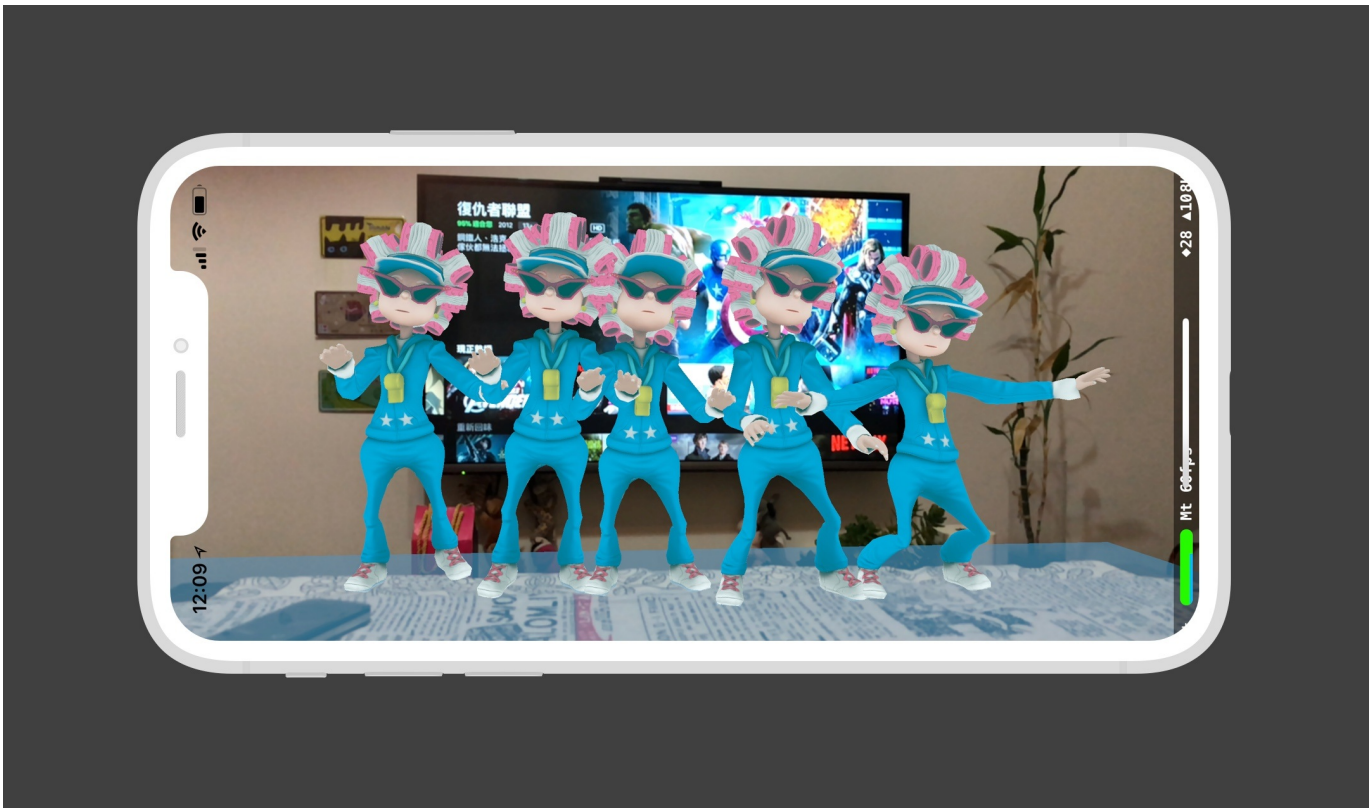


Figure 41.14. The app now supports multiple bird types and animations

For reference, you can download the complete Xcode project of this chapter and the solution of the exercise below. But before you check out the solution, make sure you try hard to figure out the solution. Here are the download links:

- Complete Xcode Project (http://www.appcoda.com/resources/swift5/ARKitDemo.zip)
- Solution to Exercise (http://www.appcoda.com/resources/swift5/ARKitDemoExercise.zip)

Chapter 42
Working with 3D Objects in Augmented Reality Using ARKit and SceneKit



In the previous chapter, you learned how to use SpriteKit to work with 2D objects in AR space. With 2D AR, you can overlay a flat image or a label in the real environment. However, do you notice one behaviour that seems weird to you? When you move your phone around, the emoji character (or the bird) always faces you. You can't look the bird from behind!

This is normal when you put a 2D object in a 3D space. The object will always face the viewer. If you want to move around the object and see how it looks from the side (or the back), you will need to work with 3D objects. In order to do that, you will need to implement the AR app using SceneKit instead of SpriteKit.

By pairing ARKit with the SceneKit framework, developers can add 3D objects to the real world. In this chapter, we will look into SceneKit and see how we can work with 3D objects. In brief, you will understand the following concepts and be able to create an ARKit app with SceneKit after going through the chapter:

- Understand how to pair SceneKit with ARKit
- Learn where to find free/premium 3D models
- Learn how to convert 3D models to SceneKit compatible format and import them into Xcode project
- Understand how to add 3D objects to AR environment
- Learn how to detect a horizontal plane and put virtual 3D objects on it

Cool! Let's get started.

The ARKit Demo App

Similar to what we have done in the earlier chapter, you can easily create a 3D AR app by using the built-in template. You don't even need to write a line of code. If you choose to create a new project using the Augmented Reality App template and select *SceneKit* as the content technology, Xcode will automatically generate a demo app for you.

Deploy and run the project on a real iPhone/iPad. You will see a jet aircraft floating in the AR space. You can move your phone around the virtual object. Since it's now a 3D object, you can see how the object looks from behind or the sides.

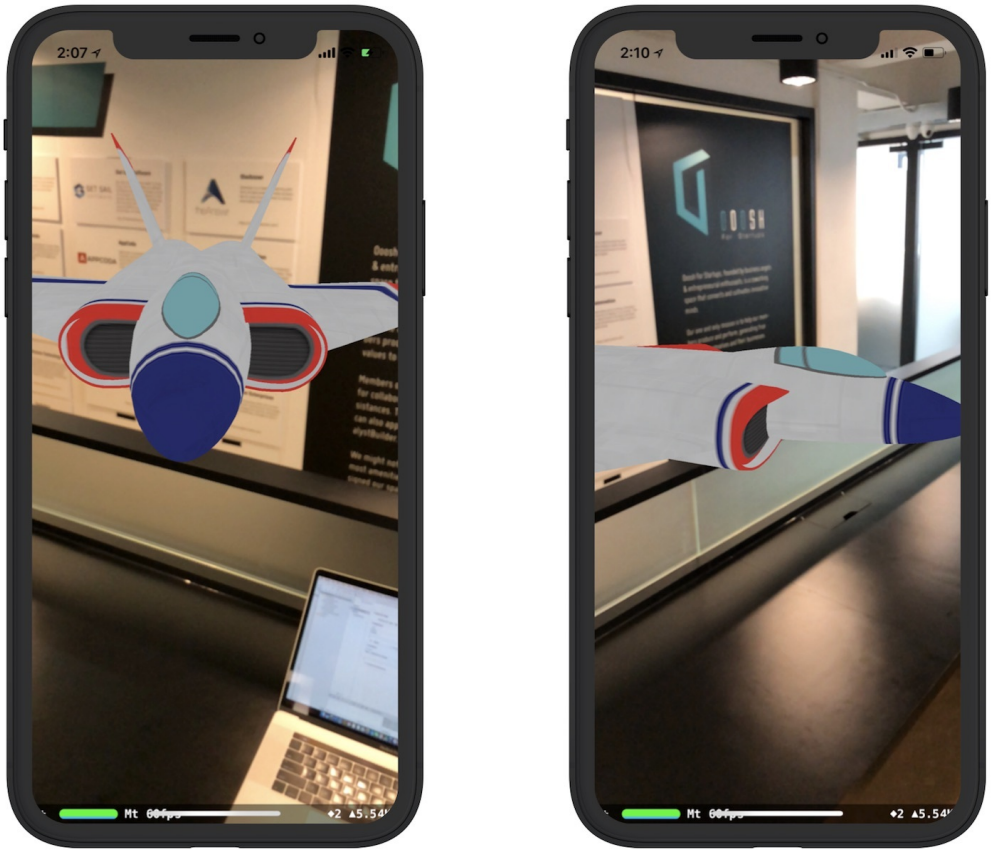


Figure 42.1. The built-in ARKit demo app rendering a jet aircraft

Understanding SceneKit and Scene Editor

While I am going to show you how to create an ARKit app from scratch, let's dive a little bit deeper and look into the code of the demo project.

We will start with the `ViewController.swift` file. If you have read the previous chapter, the code should be very familiar to you. One thing that you may wonder is how the app renders the 3D object.

SceneKit, which was first released along with iOS 8, is the content technology behind this AR demo for rendering 3D graphics. The framework allows iOS developers to easily integrate 3D graphics into your apps without knowing the low-level APIs such as Metal and OpenGL. By pairing with ARKit, SceneKit further lets you work with 3D objects in the AR environment.

All SceneKit classes begin with *SCN* (e.g. `SCNView`). For its AR counterpart, it is further prefixed with *AR* (e.g. `ARSCNView`). At the beginning of `ViewController.swift`, it has an outlet variable that connects with the `ARSCNView` object in the storyboard:

```
@IBOutlet var sceneView: ARSCNView!
```

Similar to `ARSKView`, the `ARSCNView` object is responsible for rendering the AR experiences that blend virtual 3D objects with the real world environment. To place 3D virtual objects in the real world, you can create an `ARAnchor` object and add it to the AR session. This is similar to what you have done with SpriteKit.

SceneKit, however, offers you another way to place virtual objects by using a scene graph. If you look into the `viewDidLoad()` method of `ViewController`, you will find that the demo app loads a scene file to create a `SCNScene` object. This object is then assigned to the AR view's scene.

```
// Create a new scene
let scene = SCNScene(named: "art.scnassets/ship.scn")!

// Set the scene to the view
sceneView.scene = scene
```

This is where the magic happens. ARKit automatically matches the SceneKit space to the real world and places whatever virtual objects found in the scene file. If you open the `ship.scn` file of the scene assets, you will find a 3D-model of a jet aircraft, which was located in front of the three-axis. This is the exact model rendered in the AR app.

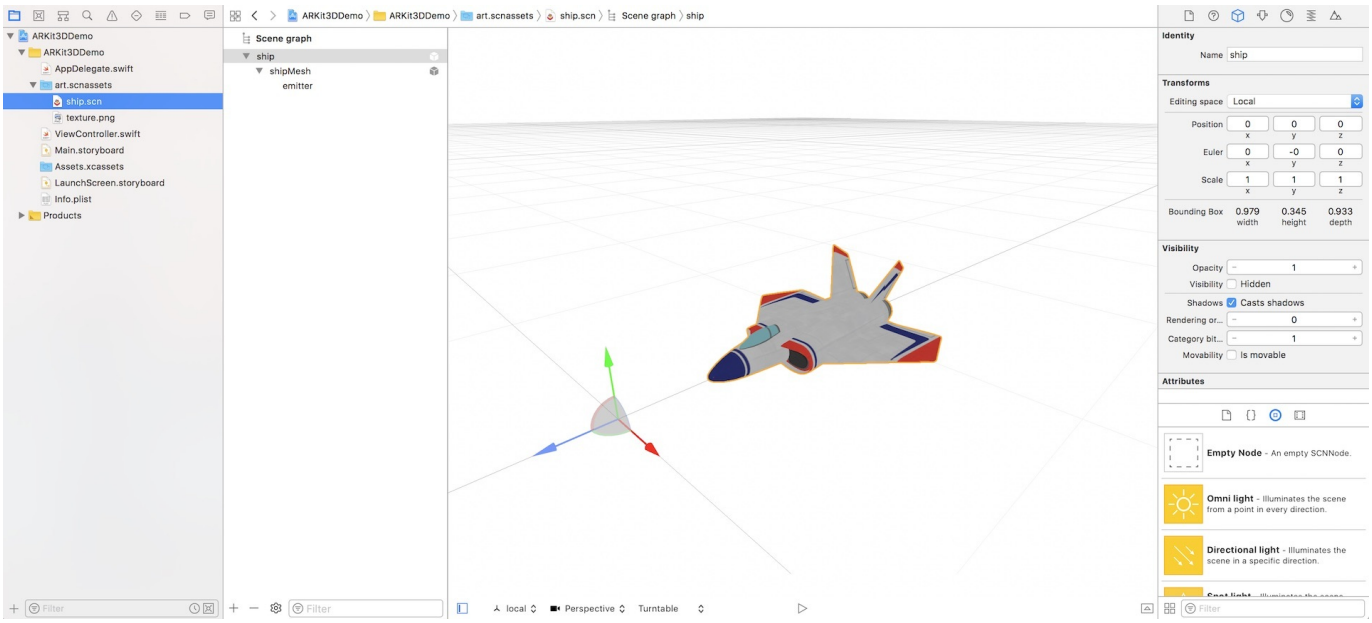


Figure 42.2. The ship.scn file

You can click the *Scene Graph View* button to reveal the scene graph. As you can see, a scene is actually comprised of multiple nodes. And, the hierarchical tree of nodes forms what-so-called *Scene Graph*.

In the scene graph above, the `ship` node contains both `shipMesh` and `emitter` nodes. Under the Node inspector, you can reveal the name (i.e. identity) and the attributes of the node. For example, the position of the `ship` node is set to (0, 0, 0). This is why the jet aircraft renders right in front of your device's camera when you launched the demo app.

Let's have a quick test. Change the value of the *z* axis from `0` to `-1` and test the app again. This will place the aircraft 1 meter from your device's camera.

There is nothing fancy here. In the previous chapter, we programmatically set the position of a sprite node. Now you can do it by using the scene editor. Of course, if you prefer, you can change the position of a node using code. Later this chapter, I will show you the code.

Let's continue to explore the scene editor. In the lower right corner, you should find the Object library. It is quite similar to the Object library of Interface Builder. Instead of showing you the UIKit components, it provides developers with common components

(e.g. Box, Sphere) of SceneKit.

Now, let's drag the *3D Text* object to the scene and place it near the aircraft. You can use this object to render 3D text in the AR environment. To set the content, choose the Attributes inspector and set the *Text* field to `Welcome to ARKit`. Then go back to the Node inspector and change the values of the Transforms section to the following:

- **Position** - Set *x* to `0`, *y* to `0.3`, and *z* to `-1`. This will place the 3D text above the aircraft.
- **Scale** - Set the value of *x*, *y*, and *z* to `0.01`. This default size of the text is too big. By setting the values of scale to 0.01, this will scale down its size.

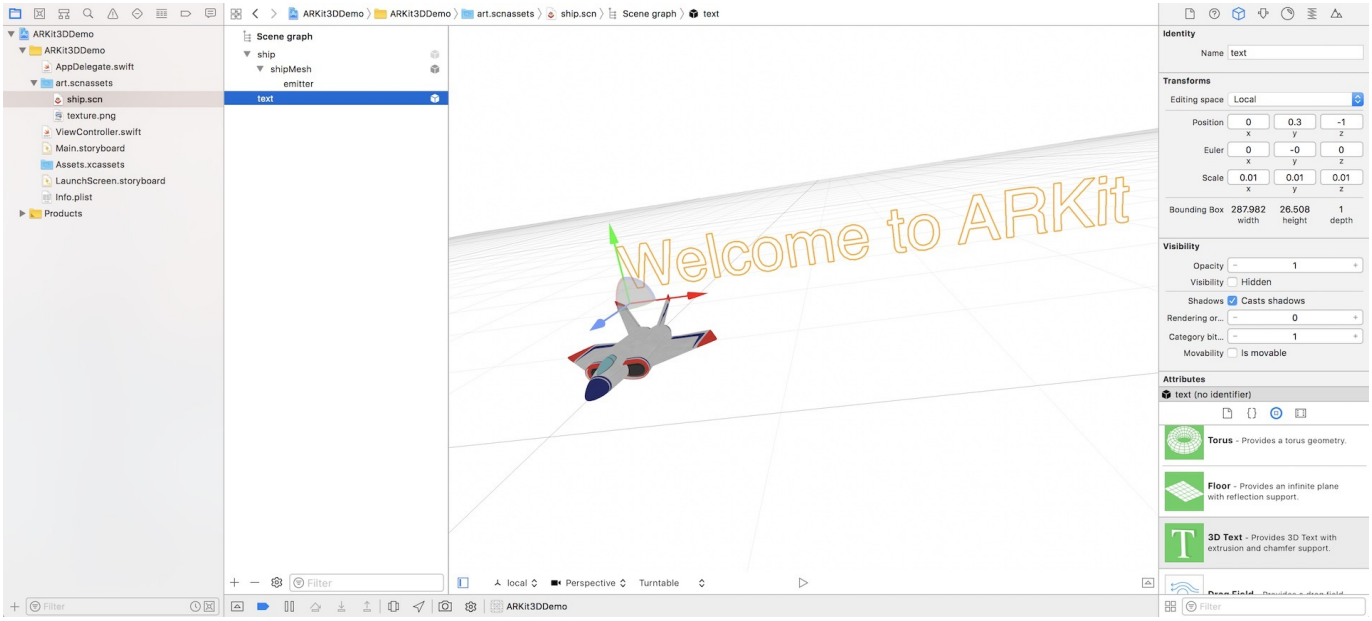


Figure 42.3. Adding 3D text using the scene editor

It's time to test the app again. Compile the project and deploy the app to your iPhone. When the app is launched, you should see the jet aircraft and the 3D text.

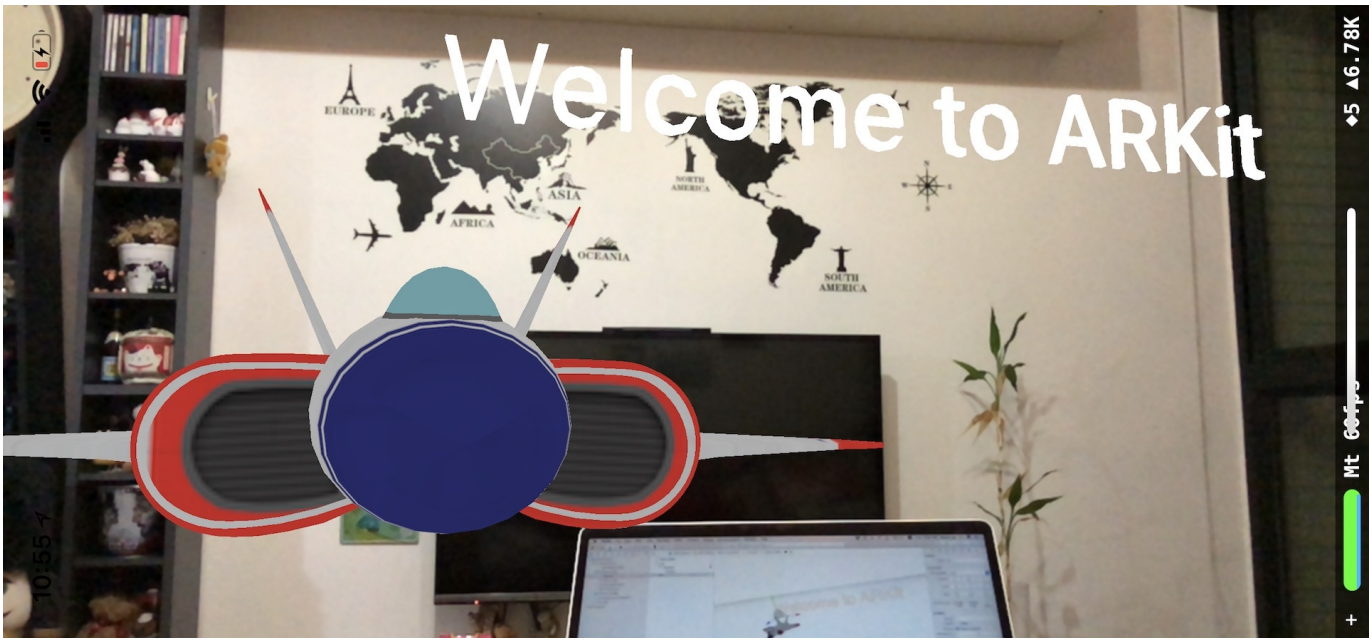


Figure 42.4. 3D text in the AR environment

Cool, right? The scene editor allows you to edit your scene without writing a line of code. Whichever components you put in the scene file, ARKit will blend the virtual objects and put them in the real world.

This is pretty much how the ARKit demo app works. I recommend you to play around with the scene editor. Try to add some other objects and edit the properties. This will further help you understand the concept.

Building Your ARKit App from Scratch

I introduced you the basics of SceneKit by walking you through the demo app. Wouldn't it be great if you can learn how to build your own ARKit app from scratch without using the built-in template? This is what I want to show you in the rest of the chapter. Specifically, you will learn the following stuff:

- How to create an ARKit app using the Single View Application template
- How to import a 3D model into Xcode projects
- How to detect a plane using ARKit
- How to place multiple virtual 3D objects on the detected plane

Okay, let's get started.

Where do You Find the 3D Objects?

Before diving into the code, the very first question you may have is where to find the 3D models. If you know 3D graphics design, chances are that you already have created some 3D characters or objects. What if you don't know anything about 3D object creation? You probably like to get some 3D models from these online resources:

- SketchFab (https://sketchfab.com)
- TurboSquid (https://www.turbosquid.com)
- Google Poly (https://poly.google.com)

Not all models are free for download, but there is no shortage of free models. Of course, if budget is not a big concern, you can also purchase premium models from the above websites.

> When you include a scene file in DAE or Alembic format in your Xcode project, Xcode automatically converts the file to SceneKit's compressed scene format for use in the built app. The compressed file retains its original `.dae` or `.abc` extension.
>
> - Apple's documentation (https://developer.apple.com/documentation/scenekit/scnscenesource)

The 3D objects are usually available in OBJ, FBX and DAE format. In order to load a 3D object into the ARKit app, Xcode needs to read your 3D object file in a SceneKit supported format. Therefore, if you download a 3D object file in OBJ/FBX format, you will need to convert it into one of the SceneKit-compatible formats.

As an example, go to https://poly.google.com/view/8WyS_yhFbX1 and download the 3D model of a robot. I will show how to use Blender to convert the file format. Later, we will also use this model in our ARKit app. When you hit the Download button, choose *OBJ file* to retrieve the file in OBJ format.

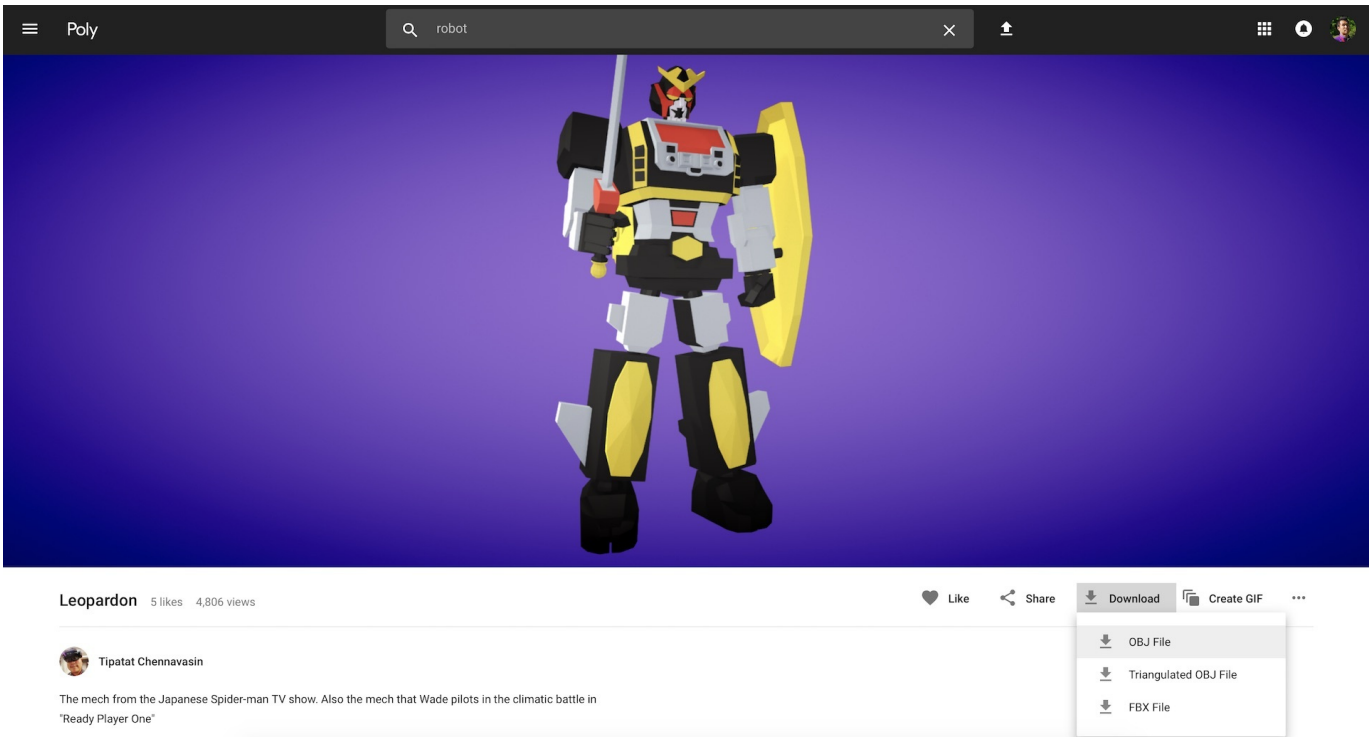


Figure 42.5. A free 3D model from Google Poly created by Leopardon

After decompressing the zip archive, you should find two files:

1. **model.obj** - this is the model of the robot.
2. **materials.mtl** - this file describes the texture used by the model.

To convert these files into SceneKit supported format, we will use an open source 3D graphics creation software called *Blender*. Now fire up Safari and point it to https://www.blender.org. The software is free for download and available for Mac, Windows, and Linux.

Once you install Blender, fire it up and you'll see a default Blender file. Go up to the Blender menu. Click *File* > *Import* > *Wavefront (.obj)*. Navigate to the folder containing the model files and choose to import the `model.obj` file.

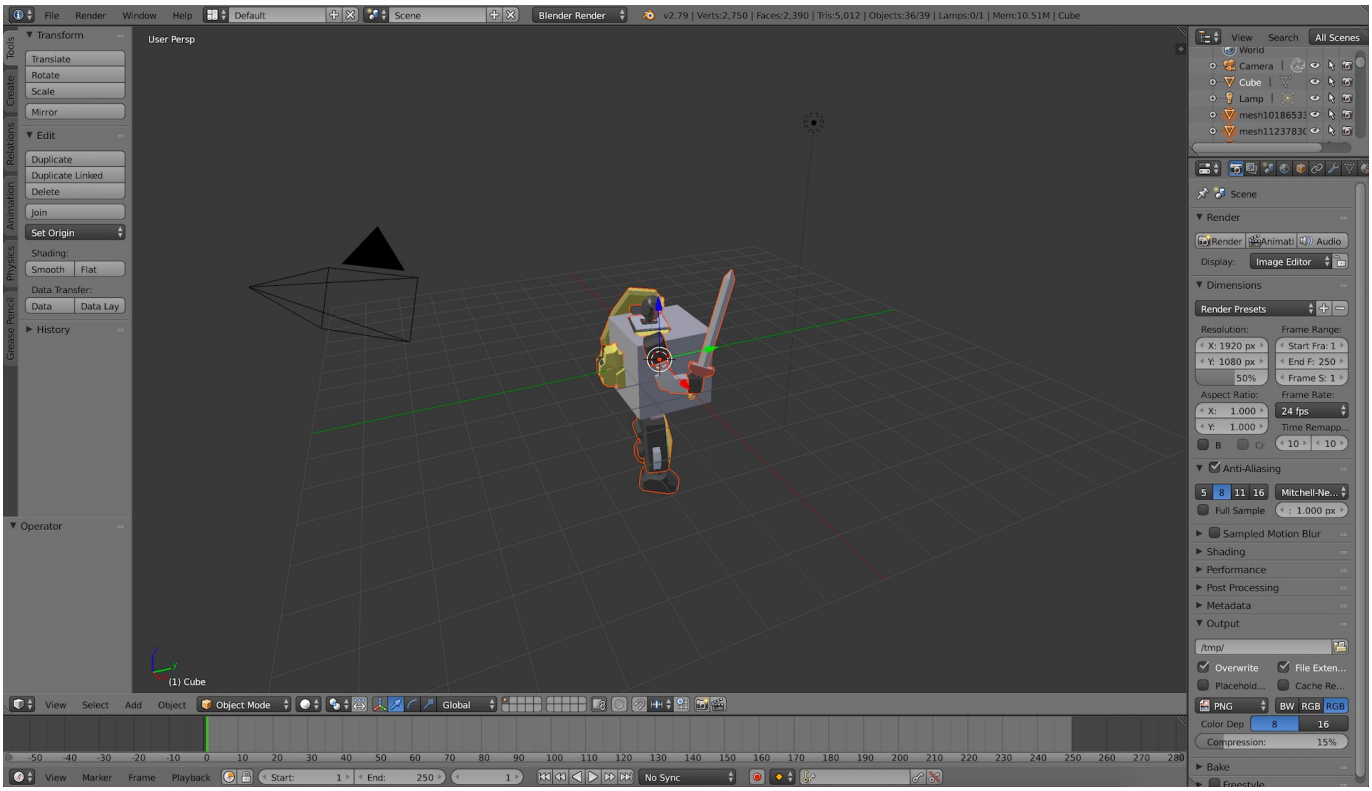


Figure 42.6. Importing the model in Blender

You should notice that the body of the robot is bounded by a cube. Blender automatically adds a cube whenever you import a model. For this project, we do not need the cube. So, right-click *Cube* under the *All Scenes* section and choose *Delete* to remove it.

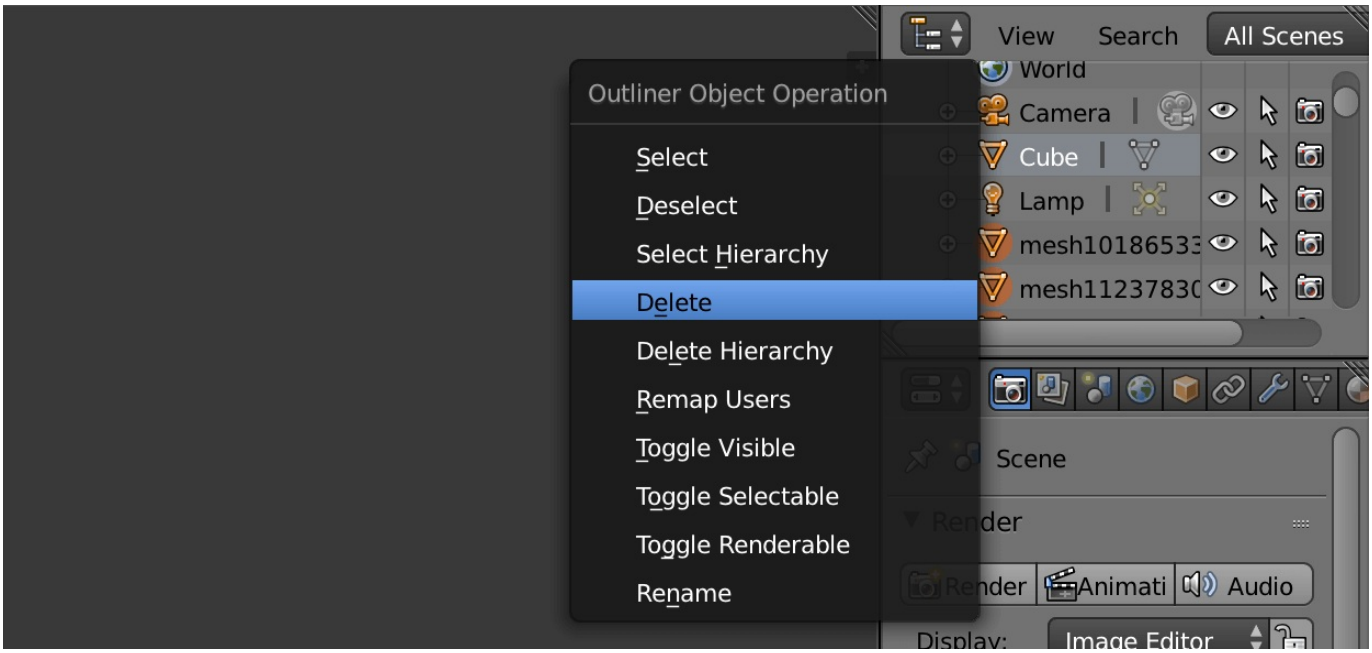


Figure 42.7. Deleting the cube

Now you're ready to convert the model to DAE format. Select *File* > *Export* > *Collada (Default) (.dae)* to export the model and save the file as *robot.dae*.

This is how you use Blender to convert a 3D model to SceneKit supported format. To preview the .dae file, simply open Finder and let it render the model for you.

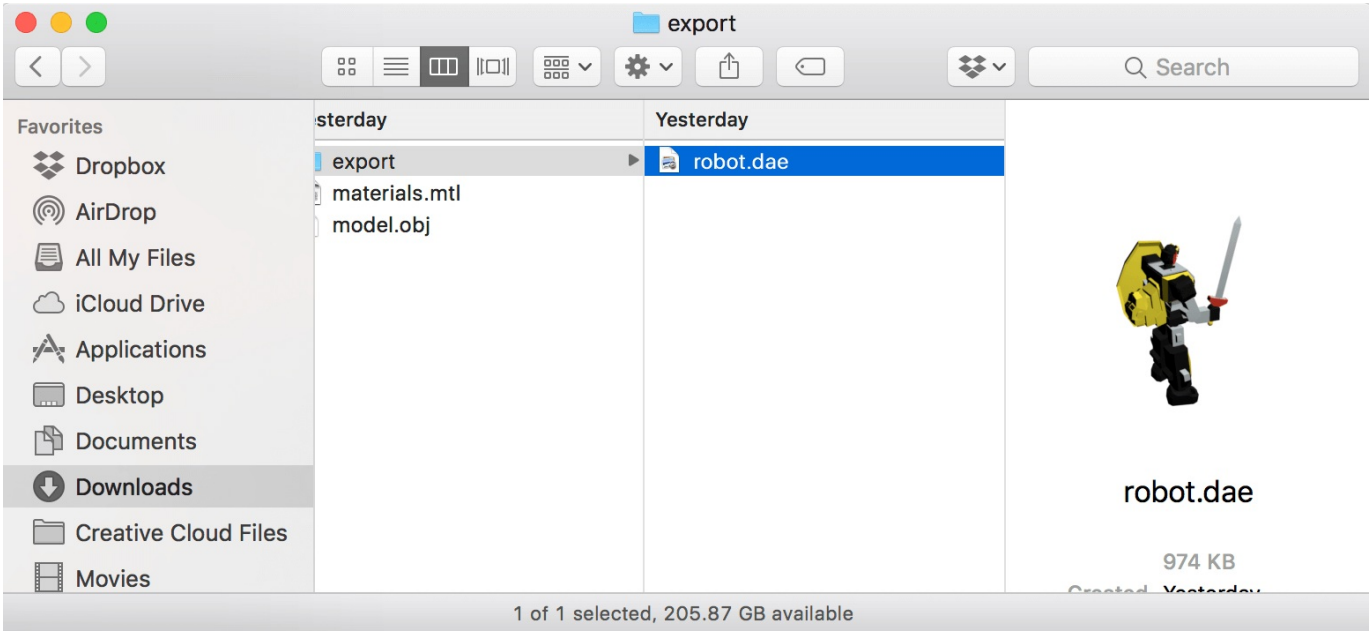


Figure 42.8. Previewing the model using Finder

Creating an ARKit App Using the Single View Application Template

Now that you have prepared the 3D model, let's begin to create our ARKit app. Open Xcode to create a new project. This time, make sure you choose to use the Single View Application template. I want to show you how to create the app from scratch.

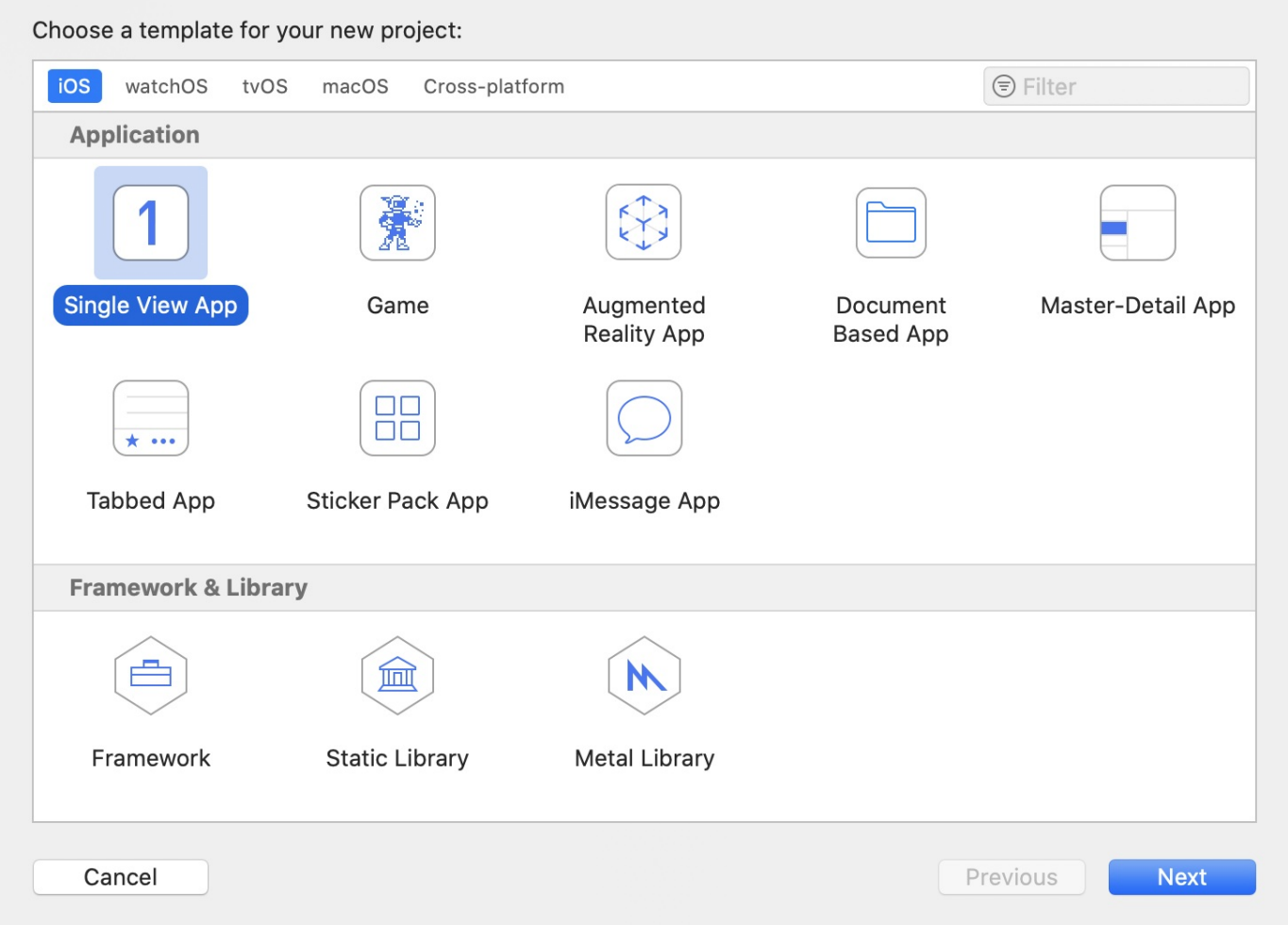


Figure 42.9. Use the Single View App template

I name the project `ARKitRobotDemo` but you are free to choose whatever name you prefer. Please make sure you choose *Storyboard* for the User Interface option. Now go to `Main.storyboard` and delete the *View* object from the view controller. In the Object library, look for the *ARKit SceneKit View* object and drag it to the view controller.

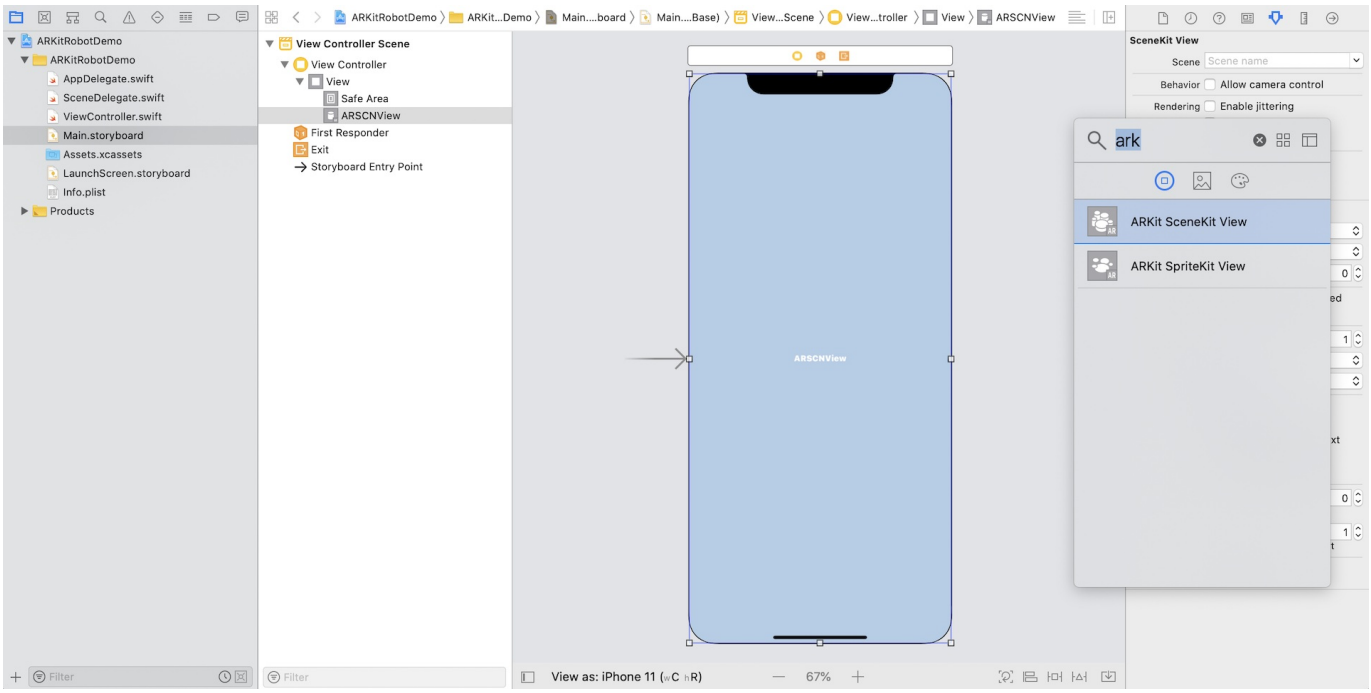


Figure 42.10. Drag the ARKit SceneKit View to the view controller

Let's go back to the code. Open `ViewController.swift` and import both *SceneKit* and *ARKit*. These are the two frameworks we need:

```
import SceneKit
import ARKit
```

Then create an outlet variable for connecting with the view we just added:

```
@IBOutlet var sceneView: ARSCNView!
```

Open the storyboard again and establish a connection between the `sceneView` variable and the *ARKit SceneKit View* object.

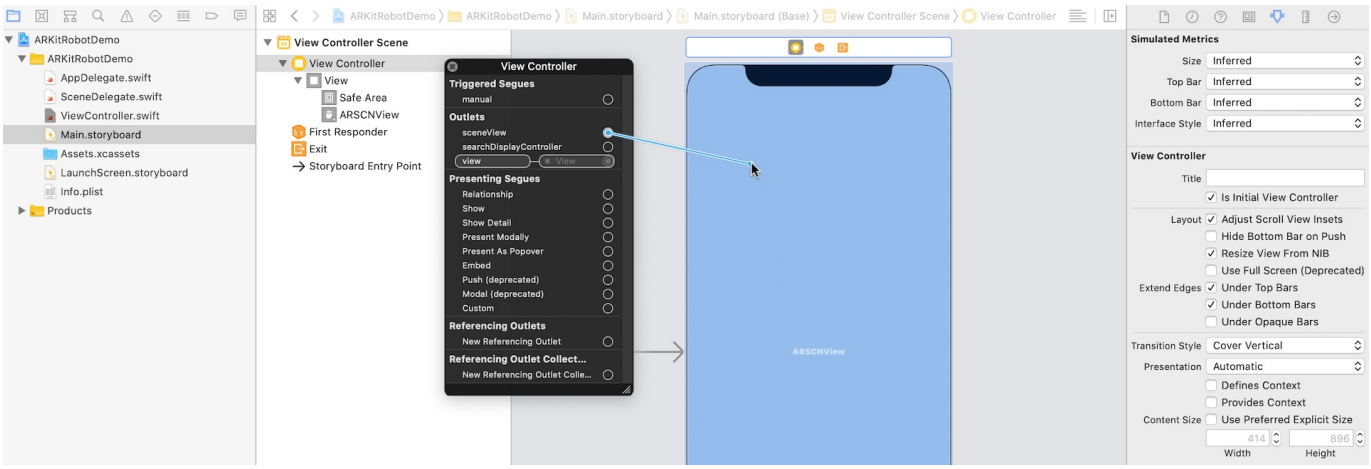


Figure 42.11. Connecting the outlet variable with the view

Next, we are going to import the .dae model created earlier into the Xcode project. All the scene assets are stored in a SceneKit asset catalog. To create the asset catalog, right-click `ARKitRobotDemo` in the project navigator. Choose *New file...*, scroll down to the Resource section and choose *SceneKit Catalog*. When prompted, name the file `art.scnassets` and save it.

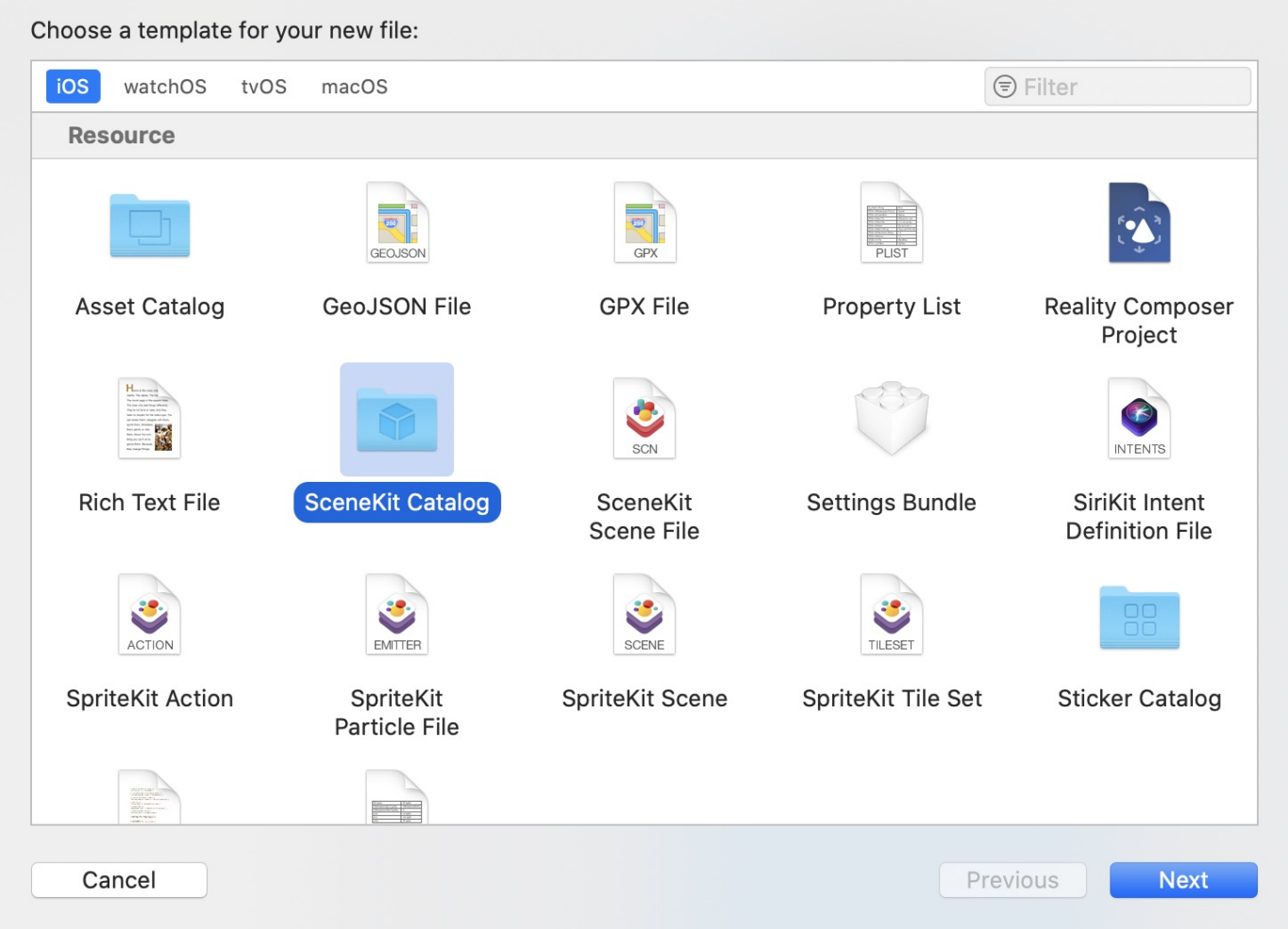


Figure 42.12. Choose the SceneKit Catalog template

Now go back to Finder and locate the `robot.dae` file. Drag it to `art.scnassets` to add the file.

Do you still remember the file extension of the SceneKit file used in the demo ARKit project? It is in .scn format. You may wonder if we have to convert the .dae file to .scn format.

The answer is no.

You can preview and edit the DAE file without converting it to .scn file because Xcode automatically converts it to SceneKit's compressed scene format behind the scene. The file extension still remains the same but the file's content has actually been converted.

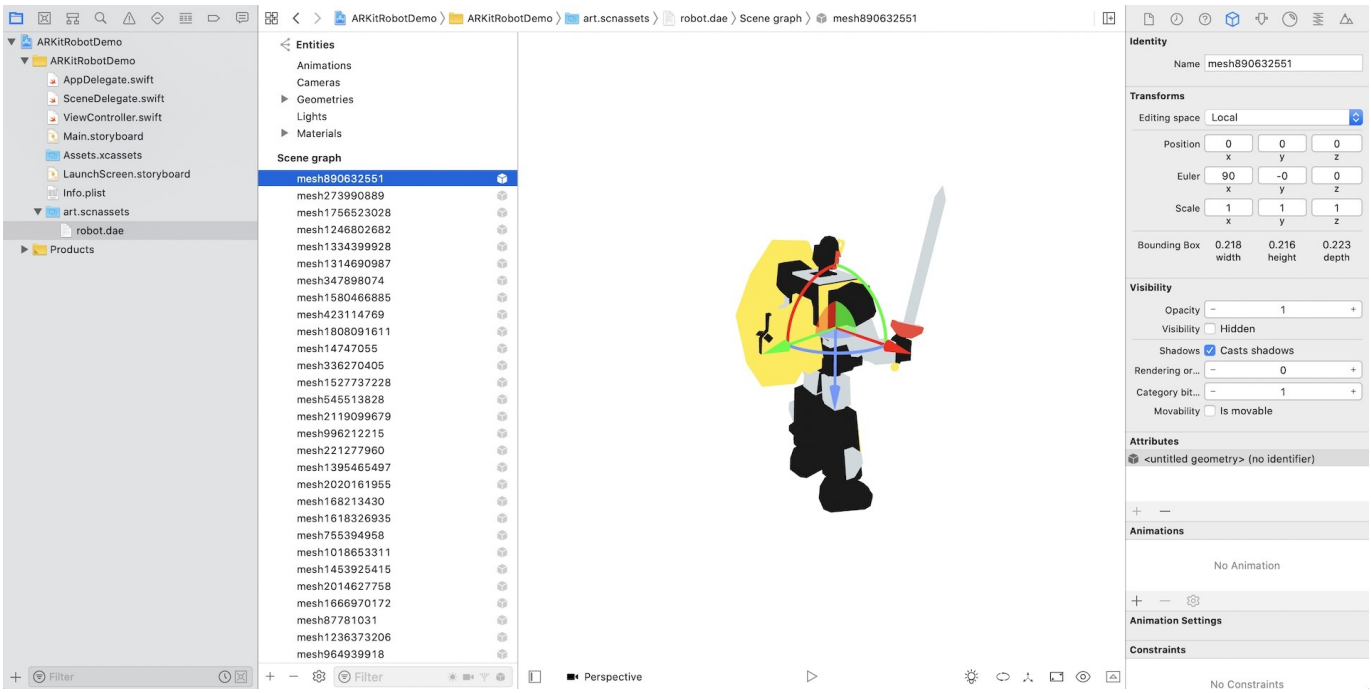


Figure 42.13. Viewing the DAE file in the scene editor

Now it's time to write some code to prepare the AR environment and render the scene file. Open the `ViewController.swift` file and update the `viewDidLoad()` method like this:

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

    // Show statistics such as fps and timing information
    sceneView.showsStatistics = true

    // Create a new scene
    let scene = SCNScene(named: "art.scnassets/robot.dae")!

    // Set the scene to the view
    sceneView.scene = scene
}
```

We instantiate a `SCNScene` object by loading the `robot.dae` file and then assign the scene to the ARKit's scene view. In order to display statistics such as fps, we also set the `showsStatistics` property of the scene view to `true`.

Next, insert the following methods in the `ViewController` class:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    // Create a session configuration
    let configuration = ARWorldTrackingConfiguration()

    // Run the view's session
    sceneView.session.run(configuration)
}

override func viewWillDisappear(_ animated: Bool) {
    super.viewWillDisappear(animated)

    // Pause the view's session
    sceneView.session.pause()
}
```

The code above is not new to you. It is the same as the one we discussed in the previous chapter. We instantiate an `ARWorldTrackingConfiguration` to track the real world and create an immersive AR experience. When the view is about to appear, we start the AR session. We pause the session when the view is going to disappear.

Now open the `Info.plist` file. Since the app needs to access the device's camera, we have to insert a new key in the file. Right-click the blank area and choose *Add row* to insert a new key named *Privacy - Camera Usage Description*. Set the value to *This application will use the camera for Augmented Reality*.

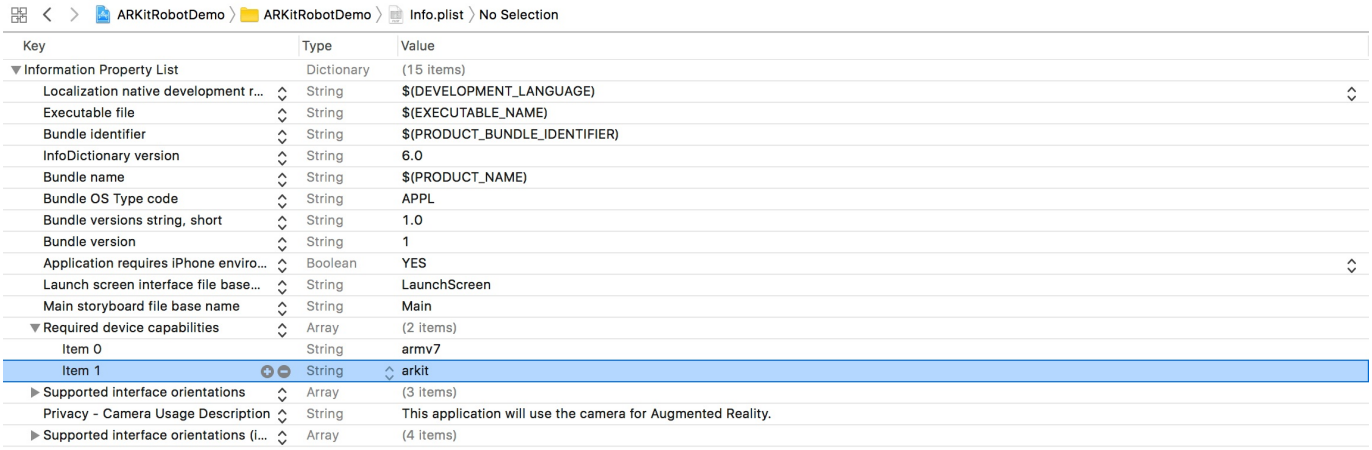


Figure 42.14. Editing the Info.plist file

Lastly, insert an additional item for the *Required device capabilities* key. Set the value of the new item to *arkit*. This tells iOS that this app can only be run on an ARKit-supported device.

Great! It is now ready to test your ARKit app. Deploy and run it on a real iPhone or iPad. You should see the robot augmented in the real world.

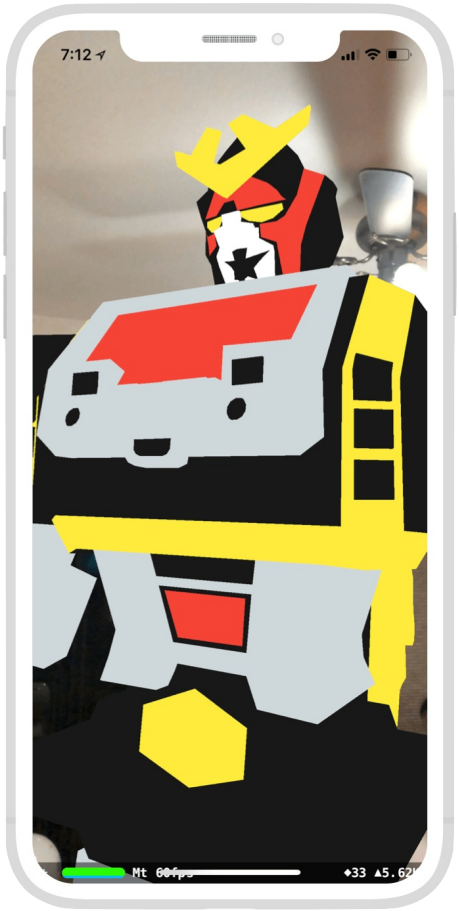


Figure 42.15. Testing the ARKit app

For reference, you can download the complete Xcode project at http://www.appcoda.com/resources/swift5/ARKitRobotDemo.zip.

Working with Plane Detection

Now that you should understand how to create an ARKit app without using the template, let's move onto another topic and create something even better. ARKit supports plane detection allowing you to detect planes and visualize them in your ARKit app. Not only can you visualize the plane, it empowers you to place virtual objects on the detected plane.

Not only can ARKit detects horizontal plane, it also empowers developers to detect vertical planes like walls and doors. For this demo, we will only use the horizontal plane detection.

Apple's engineers have made plane detection easily accessible. All you need to do is set the `planeDetection` properties of `ARWorldTrackingConfiguration` to `.horizontal`. Insert the following line in the `viewWillAppear` method and put it after the instantiation of the `ARWorldTrackingConfiguration` object:

```
configuration.planeDetection = .horizontal
```

With this line of code, your ARKit app is ready to detect horizontal planes. Whenever a plane is detected, the following method of the `ARSCNViewDelegate` protocol is called:

```
optional func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anc
hor: ARAnchor)
```

We will adopt the protocol by an extension like this:

```
extension ViewController: ARSCNViewDelegate {

    func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor:
ARAnchor) {
        print("Surface detected!")

    }

}
```

Meanwhile, to keep things simple, we simply print a message to the console when a plane is detected.

In the `viewDidLoad` method, insert the following two lines of code:

```
sceneView.delegate = self
sceneView.debugOptions = [ ARSCNDebugOptions.showFeaturePoints ]
```

The first line of code is very straightforward that we set the delegate to itself. The second line is optional. By enabling the debugging option to show feature points, however, ARKit will render the feature points as yellow dots. You will understand what I mean after running the app.

Now deploy and run the app on your iPhone. After the app is initialized, point the camera to any horizontal surfaces (e.g. floor). If the plane is detected, you will see the message "Surface detected" in the console.

By the way, as you move the camera around, you should notice some yellow dots, which are the feature points. These points represent the notable features detected in the camera image.

Figure 42.16. The ARKit app shows you feature points

Visualizing the Detected Plane

Meanwhile, the app doesn't highlight the detected plane on the screen. We just display a console message when a surface is detected. Wouldn't it be great if we can visualize the detected plane on the screen like the one shown in figure 42.17?

Figure 42.17. Visualizing the detected plane

As mentioned earlier, the following method is called every time when a plane is detected:

```
optional func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anc
hor: ARAnchor)
```

I didn't discuss the method in details. But it actually passes us two pieces of information when the method is invoked:

- **node** - this is the newly added SceneKit node, created by ARKit. By utilizing this node, we can provide visual content to highlight the detected plane.
- **anchor** - the AR anchor corresponding to the node. Here it refers to the detected plane. This anchor will also provide us extra information about the plane such as the

plane size and position.

Now update the method like this to draw a plane on the detected flat surface:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAn
chor) {

    if let planeAnchor = anchor as? ARPlaneAnchor {

        // Create a virtual plane to visualize the detected plane
        let plane = SCNPlane(width: CGFloat(planeAnchor.extent.x), height: CGFloat
(planeAnchor.extent.z))

        // Set the color of the virtual plane
        plane.materials.first?.diffuse.contents = UIColor(red: 90/255, green: 200/
255, blue: 250/255, alpha: 0.50)

        // Create the SceneKit plane node
        let planeNode = SCNNode(geometry: plane)
        planeNode.position = SCNVector3(planeAnchor.center.x, 0.0, planeAnchor.cen
ter.z)

        // Since the plane in SceneKit is vertical, we have to rotate it by 90 deg
rees
        // The value should be in the form of radian.
        planeNode.eulerAngles.x = -Float.pi / 2.0

        node.addChildNode(planeNode)
    }
}
```

Whenever ARKit detects a plane, it automatically adds an `ARPlaneAnchor` object. Therefore, we first check if the parameter `anchor` has the type `ARPlaneAnchor`.

To visualize the detected plane, we draw a plane over it. This is why we create a `SCNPlane` object with the size of the detected plane. The AR plane anchor provides information about the estimated position and shape of the surface. You can get the width and length of the detected plane from the `extent` property.

For the next line of code, we simply set the color of the plane.

In order to add this plane, we create a `SCNNode` object and set its position to the plane's position. By default, all planes in SceneKit is vertical. To change its orientation, we update the `eulerAngles` property of the node to rotate the plane by 90 degrees.

Lastly, we add this plane as a child node.

Now if you run the app again, it will be able to visualize the detected plane.

Updating the Planes

As you walk around your room to play around with plane detection, you may end up with results similar to that shown in figure 42.18.

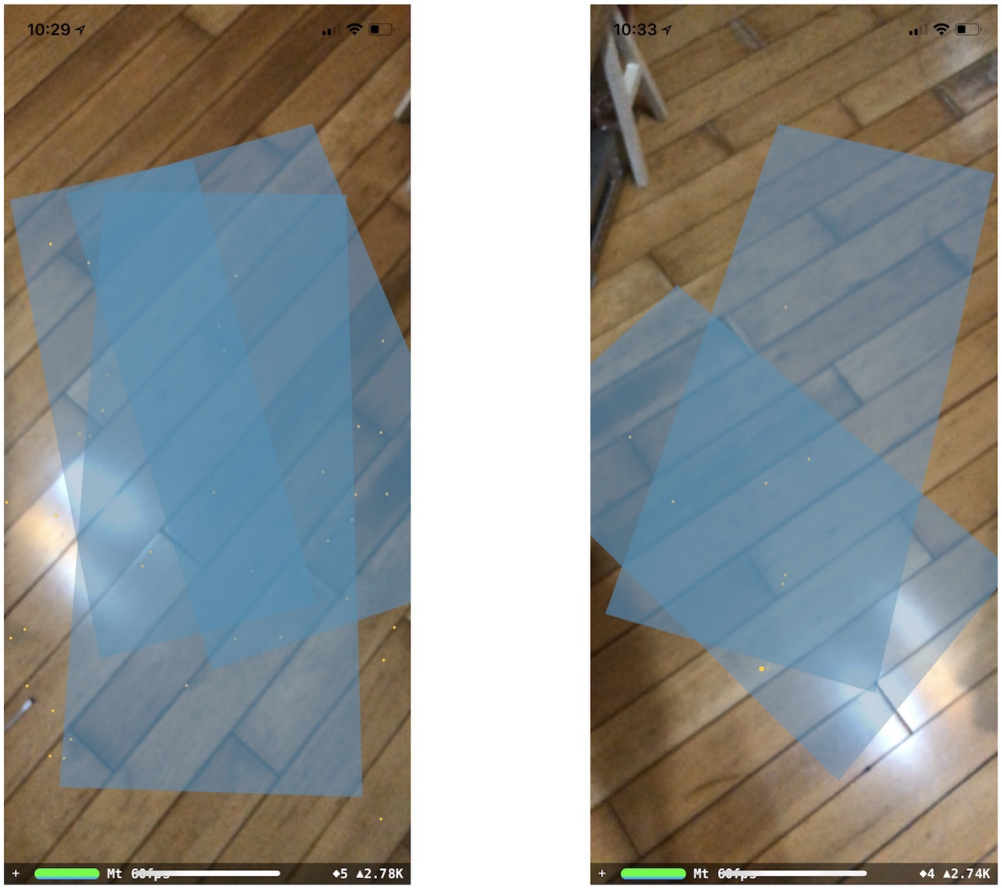


Figure 42.18. Duplicated planes

The app now renders a virtual plane whenever a flat surface is detected. This is why you may find one virtual plane overlaps with another. In fact, ARKit keeps updating the detected plane as you move the device's camera around. No matter how the updated plane changes (whether it's bigger or smaller), it calls the following delegate method to inform you about the update:

```
optional func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNode, for
anchor: ARAnchor)
```

So, to render the updated plane on the screen, we must implement the method and update the virtual plane accordingly. In order to perform the update, we will need to keep track the list of virtual planes created. Thus, let's organize our code a bit for this purpose.

Now create a new Swift file named `PlaneNode.swift`. Update the file content with the following code:

```
import Foundation
import SceneKit
import ARKit

class PlaneNode: SCNNode {
    private var anchor: ARPlaneAnchor!
    private var plane: SCNPlane!

    init(anchor: ARPlaneAnchor) {
        super.init()

        self.anchor = anchor

        // Create a virtual plane to visualize the detected plane
        self.plane = SCNPlane(width: CGFloat(anchor.extent.x), height: CGFloat(anc
hor.extent.z))

        // Set the color of the virtual plane
        self.plane.materials.first?.diffuse.contents = UIColor(red: 90/255, green:
200/255, blue: 250/255, alpha: 0.50)

        // Create the SceneKit plane node
        self.geometry = plane
        self.position = SCNVector3(anchor.center.x, 0.0, anchor.center.z)

        // Since the plane in SceneKit is vertical, we have to rotate it by 90 deg
rees
        // The value should be in the form of radian.
        self.eulerAngles.x = -Float.pi / 2.0
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }

}
```

`PlaneNode` is a subclass of `SCNNode`, which is used to store two properties including the anchor of the detected plane and the virtual plane drawn. If you look closely at the `init(anchor:)` method, the code is exactly the same as we have implemented. With the given plane anchor, we create a `SCNPlane` object for rendering the virtual plane.

Next, we will edit the `ViewController` class to make use of this newly created class. As we need to keep track of the list of virtual planes, declare the following dictionary variable in `ViewController`:

```
private var planes: [ UUID: PlaneNode ] = [:]
```

We use a dictionary to store the list of planes. `ARAnchor` has a property named `identifier` that stores a unique identifier of the anchor. The key of the `planes` dictionary is the identifier of the detected plane anchor.

Now update the `renderer(_:didAdd:for:)` method like this:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAn
chor) {

    if let planeAnchor = anchor as? ARPlaneAnchor {

        let planeNode = PlaneNode(anchor: planeAnchor)

        planes[anchor.identifier] = planeNode
        node.addChildNode(planeNode)
    }
}
```

As most of the code is relocated to the `PlaneNode` class, we can simply create a `PlaneNode` object using the detected plane anchor. Similarly, we add the plane node as a child node. Additionally, we store this virtual plane in the `planes` variable.

If you test the app again, everything works like before. It will be able to show you a virtual plane when a flat surface is detected, but the virtual plane is still not expandable.

To update the virtual plane, let's create another method in the `PlaneNode` class:

```
func update(anchor: ARPlaneAnchor) {
    self.anchor = anchor

    // Update the plane's size
    plane.width = CGFloat(anchor.extent.x)
    plane.height = CGFloat(anchor.extent.z)

    // Update the plane's position
    self.position = SCNVector3(anchor.center.x, 0.0, anchor.center.z)
}
```

The method is simple. It takes in the new anchor and updates the virtual plane accordingly.

Now we are ready to implement the `renderer(_:didUpdate:for:)` method. Insert the following code in the `ViewController` extension:

```
func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNode, for anchor: A
RAnchor) {
    guard let planeAnchor = anchor as?  ARPlaneAnchor, let plane = planes[planeAnc
hor.identifier] else {
        return
    }

    plane.update(anchor: planeAnchor)
}
```

We first verify if the updated anchor is found in our list. If yes, we call the `update(anchor:)` method to update the size & position of the virtual plane.

That's it! Deploy the app onto your iPhone again. This time, the virtual plane keeps updating itself as you move the camera around a flat horizontal surface.

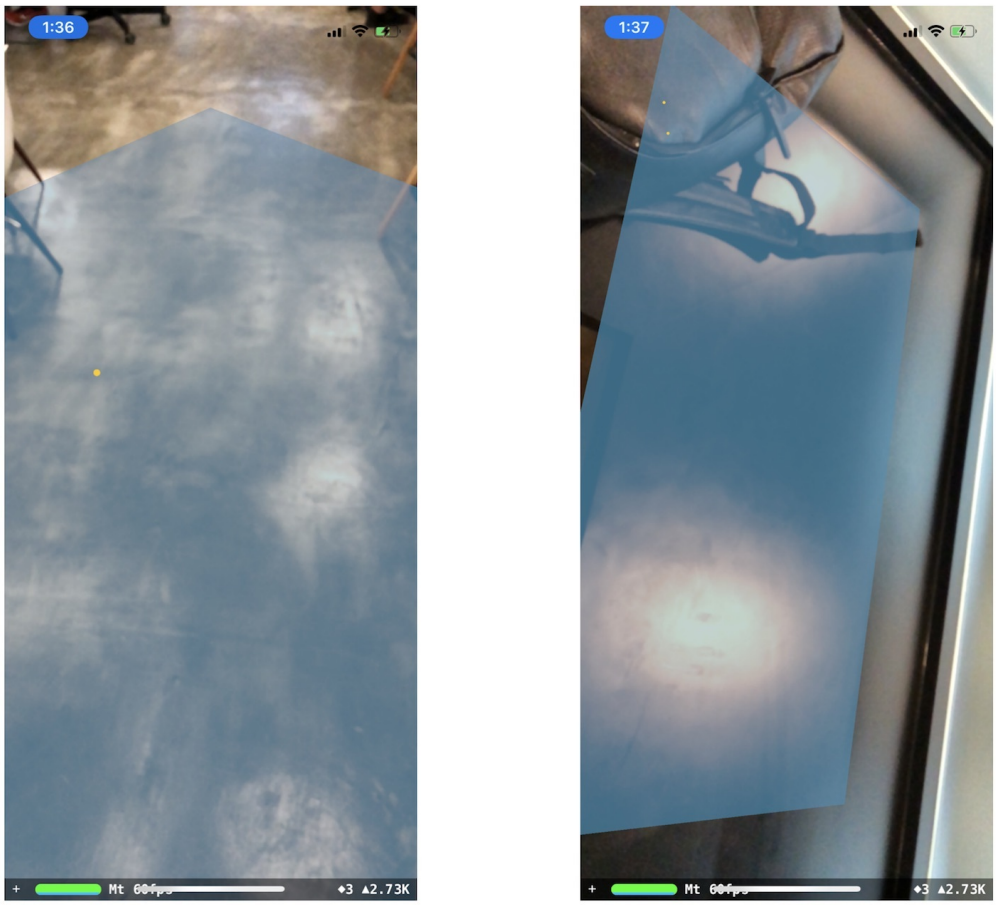


Figure 42.19. Now the virtual plane can update itself

Adding 3D Objects on a Plane

Now that you have learned how to place a 3D object in the real world and detect a flat surface, let's combine these two things together and create something awesome. We will enhance the ARKit app to let users place multiple robots on a flat surface.

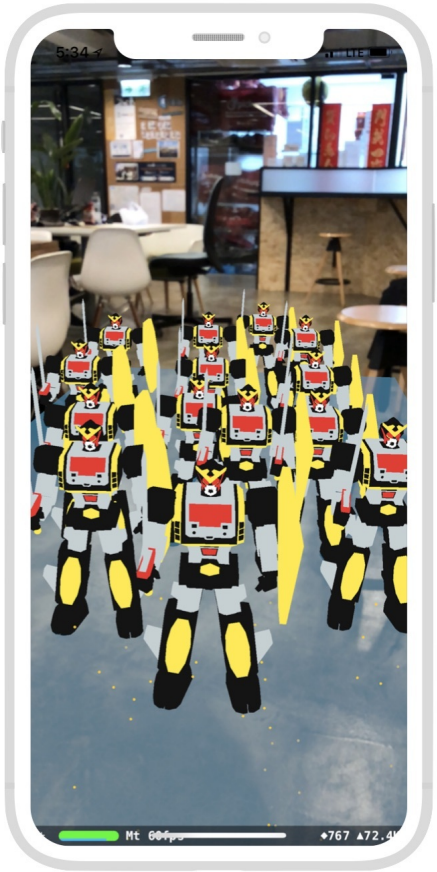


Figure 42.20. A team of robots

First, open `ViewController.swift` and change the following line of code in the `viewDidLoad()` method.

From:

```
let scene = SCNScene(named: "art.scnassets/robot.dae")!
```

To:

```
let scene = SCNScene()
```

We no longer load the robot scene right after the app launch. Instead, we want to let users tap on the detected plane to place the robot. So, an empty scene is more suitable in this case.

Next, insert the following line of code in the same method:

```
let tapGestureRecognizer = UITapGestureRecognizer(target: self, action: #selector(
addRobot(recognizer:)))
sceneView.addGestureRecognizer(tapGestureRecognizer)
```

We configure a tap gesture recognizer to detect the user's touches. When a tap is detected, we will call the `addRobot(recognizer:)` method to place the virtual robot. For this method, we implement like this:

```
@objc func addRobot(recognizer: UITapGestureRecognizer) {
    let tapLocation = recognizer.location(in: sceneView)
    let hitResults = sceneView.hitTest(tapLocation, types: .existingPlaneUsingExte
nt)

    guard let hitResult = hitResults.first else {
        return
    }

    guard let scene = SCNScene(named: "art.scnassets/robot.dae") else {
        return
    }

    let node = SCNNode()
    for childNode in scene.rootNode.childNodes {
        node.addChildNode(childNode)
    }

    node.position = SCNVector3(hitResult.worldTransform.columns.3.x, hitResult.wor
ldTransform.columns.3.y, hitResult.worldTransform.columns.3.z)
    node.scale = SCNVector3(0.1, 0.1, 0.1)

    sceneView.scene.rootNode.addChildNode(node)

}
```

In the code above, we first get the touch's location and then check if the touch hits the detected plane. If the user taps any area outside the plane, we just ignore it. When the touch is confirmed, we load the scene file containing the robot model, loop through all the nodes and add them to the main node.

Why looping through multiple nodes here? Take a look at figure 41.13 or open `robot.dae` again. You should see multiple nodes in the scene graph. For some 3D models like the robot we are working on, they may have more than one node. In this case, we need to render all the nodes in order to display the complete model on the screen. Furthermore, by adding these child nodes to the main node, it allows us to scale or position the model easily. The second last line of the code is to resize the robot to 10% of the original size.

Lastly, we put the node to the root node of the scene view for rendering.

Run the app, move around to detect a plane and then tap on the plane to place a robot.

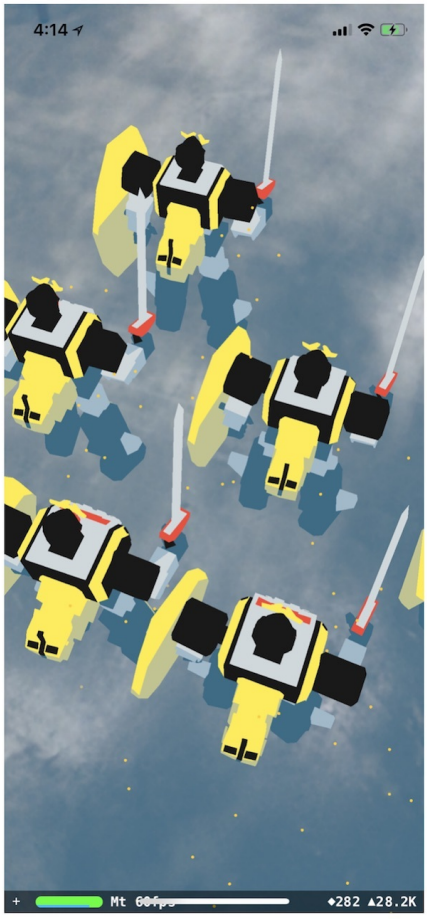


Figure 42.21. Adding the robots on the detected plane

There is something weird you may notice. The robots immerse into the surface rather than stand upright on the floor. Open the `robot.dae` file and examine the model again. The lower part of the model is below the x-axis. This explains why part of the robot body is rendered below the detected plane. It also explains why its back faces you when the robot appears on the screen.

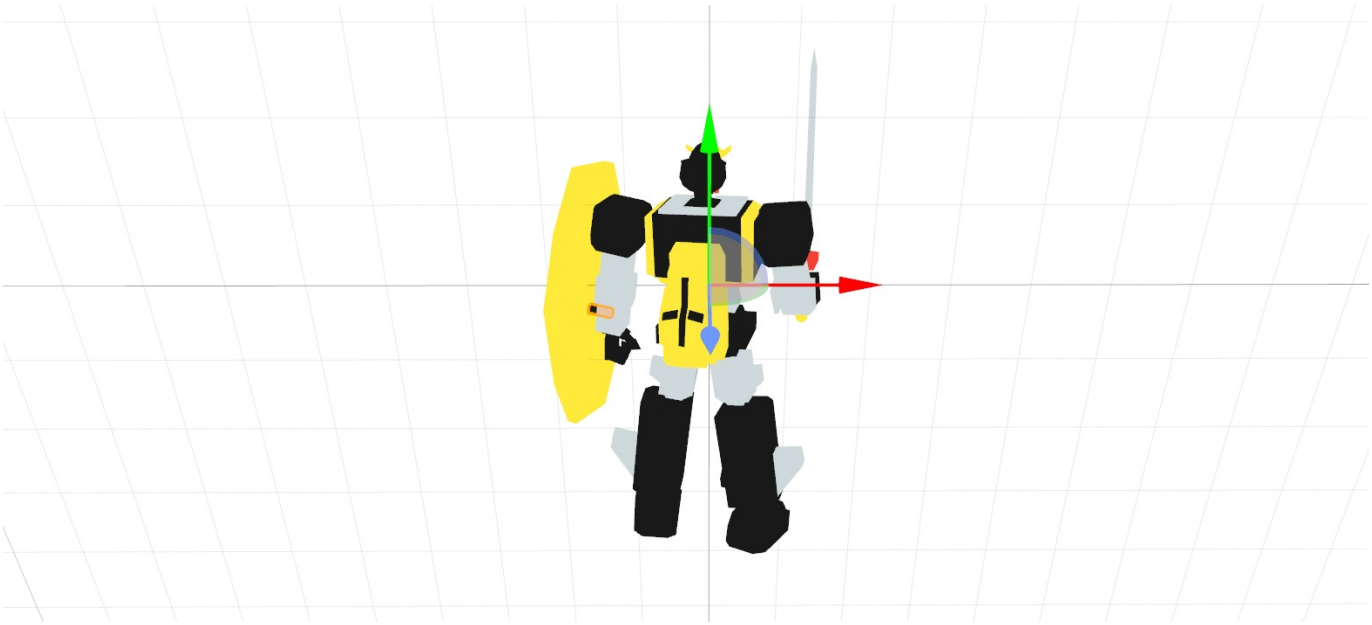


Figure 42.22. The 3D model of the robot

To improve the rendering, update the position of the node in `addRobot(recognizer:)` method like this:

```
node.position = SCNVector3(hitResult.worldTransform.columns.3.x, hitResult.worldTr
ansform.columns.3.y + 0.35, hitResult.worldTransform.columns.3.z)
```

Also, insert a line of code to rotate the model by 180 degrees (around y-axis):

```
node.rotation = SCNVector4(0, 1, 0, Float.pi)
```

Test the app again and you will see a much better result.

Figure 42.23. The robots now face towards you.

Exercise

The 3D model doesn't need to be static. You can import animated 3D models into Xcode project and render them using ARKit. Mixamo from Adobe is an online character animation service that provides many animated characters for free download. You can even upload your own 3D character and use Mixamo to create character animations.

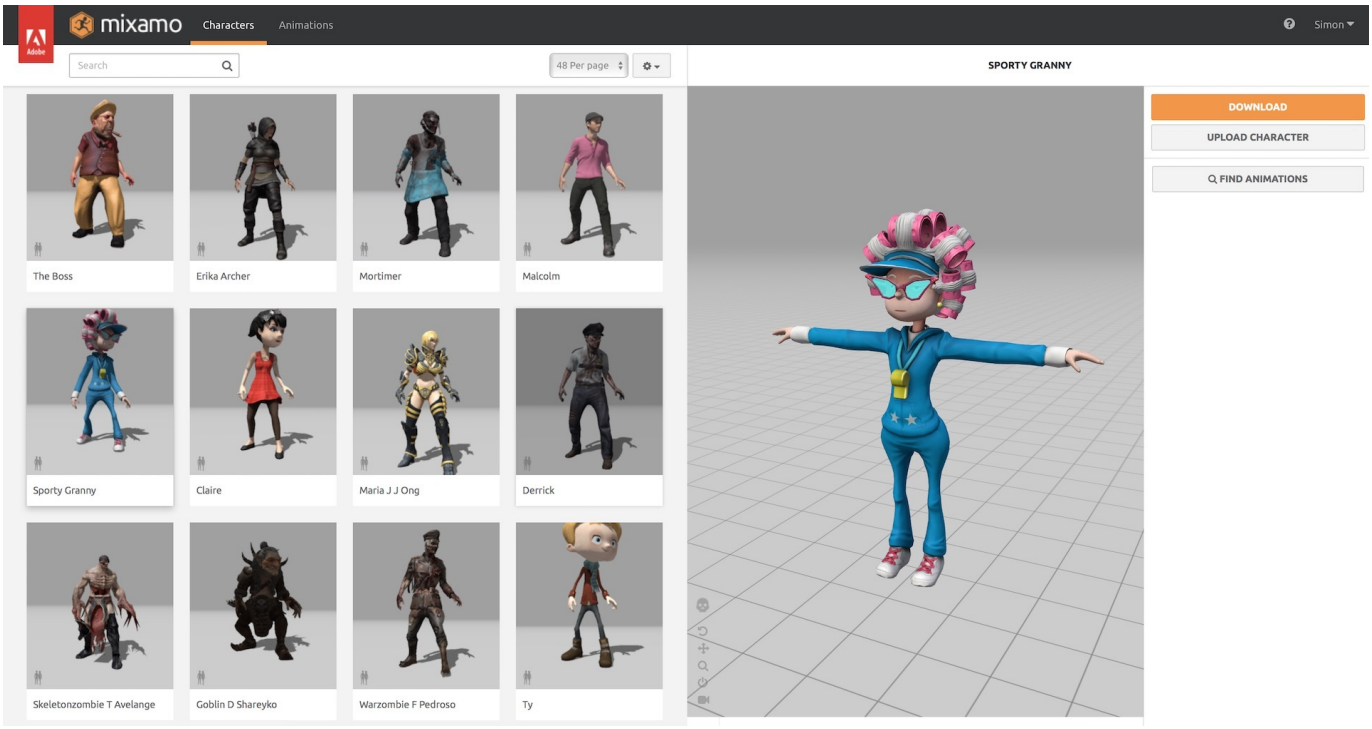


Figure 42.24. Mixamo

Your task of this exercise is to go to mixamo.com and create an animated character. Start by selecting the character you like and then click *Find animations*. Choose one of the animations (e.g. Samba Dancing) to create a 3D animated character. Once you are satisfied with your character, hit Download and choose *Collada (.dae)* as the format to download the animated character.

You can then import the .dae file, together with the textures folder, into the Xcode project. Finally, modify the code to render the animated character in augmented reality.

Figure 42.25. Rendering animated 3D characters

Summary

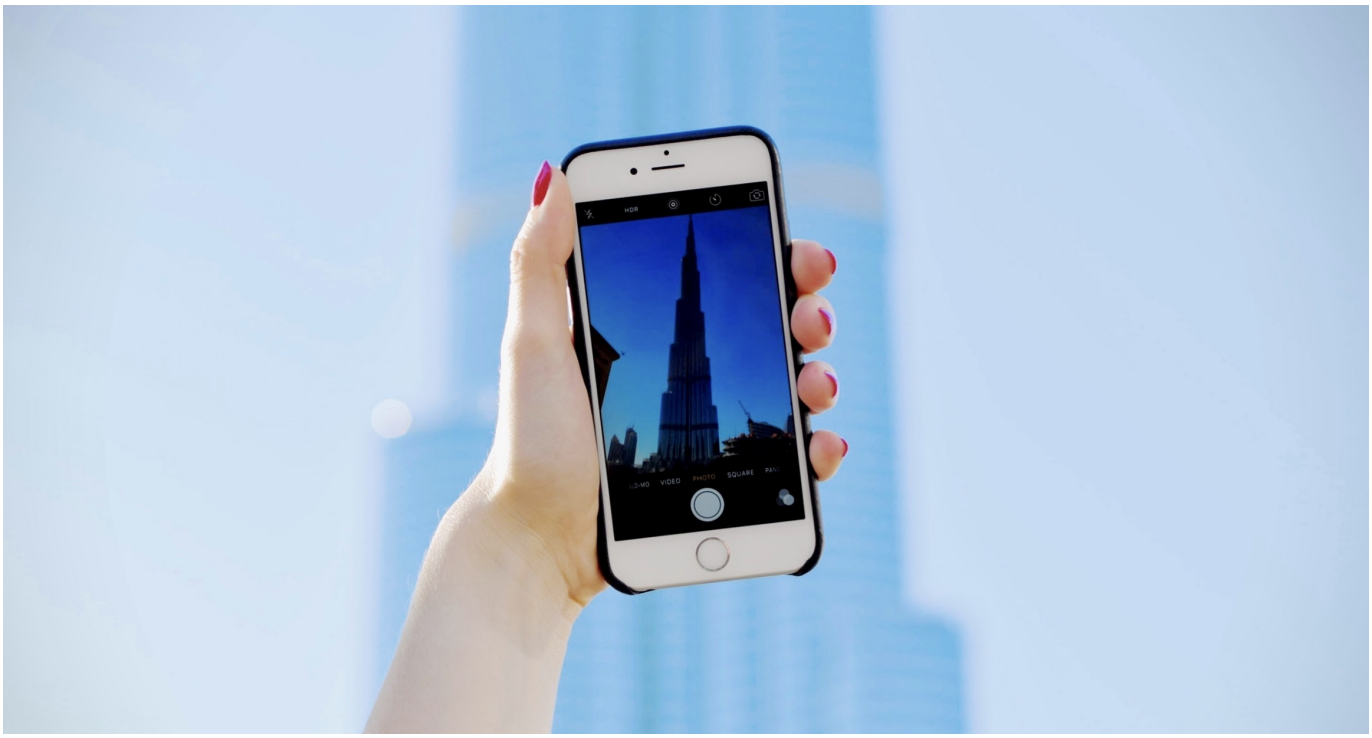
This is a huge chapter. You should now know how to find free 3D models, convert them into SceneKit supported format and add 3D objects in the real world using ARKit. Plane detection has been one of the greatest features of ARKit. The tracking is fast and robust, although the detection is less satisfactory for shinning surfaces. The whole idea of augmented reality is to seamlessly blend virtual objects into the real world. Plane detection simply allows you to detect flat surfaces like tables, floors and place objects on them. This opens up tons of opportunities and lets you to build realistic AR apps.

For reference, you can download the complete Xcode project and the sample solution to the exercise using the links below:

- Complete Xcode project
  http://www.appcoda.com/resources/swift5/ARKitRobotDemoPlaneDetection.zip
- Sample solution to the exercise

  http://www.appcoda.com/resources/swift5/ARKitRobotDemoExercise.zip

Chapter 43
Using Create ML to Train Your Own Machine Learning Model for Image Recognition



Earlier, you learned how to integrate a Core ML model in your iOS app. In that demo application, we utilized a pre-trained ML model which was created by other developers. What if you can't find a ML model that fits your need?

You will have to train your own model. The big question is how?

There are no shortage of ML tools for training your own model. Tensorflow and Caffe are a couple of the examples. However, all these tools require lots of code and don't have a friendly visual interface. Starting from Xcode 10, Apple introduced a new tool called **Create ML** that allows developers (and even non-developers) to train their own ML model.

In Xcode 10, Create ML was built right into Playgrounds. With the release of Xcode 11, Create ML is now an independent full featured app. To train your own ML model, all you need is import your training data into the Create ML tool and you are good to go. You will understand what I mean when we dive into the demo in later section.

Last year, Create ML only focused on these three main areas of machine learning models:

1. Images
2. Text
3. Tabular data

In Xcode 11, Apple allows developers to train ML model for Sound and Activity.

Say, for images, you can create your own image classifier for image recognition. In this case, you take a photo or an image as input and the ML model outputs the label of the image. You can also create your own ML model for text classification. For example, you can train a model to classify if a user's comment is positive or negative.

In this chapter, we will focus on training a ML model for image recognition. For other types of ML models, we will look into them in later chapters.

The Workflow of Creating a ML Model

When you need to create your own ML model, you usually start with a problem. Here are a couple of the examples:

- I can't find any ML models to identify all type of cuisines.
- All existing ML models I found can only identity 1,000 real world objects. However, some objects such as HomePod and Apple Watch are not available in the model.

Generally speaking, the main reason why you need to build your own ML model is there is no existing model that fits your need.

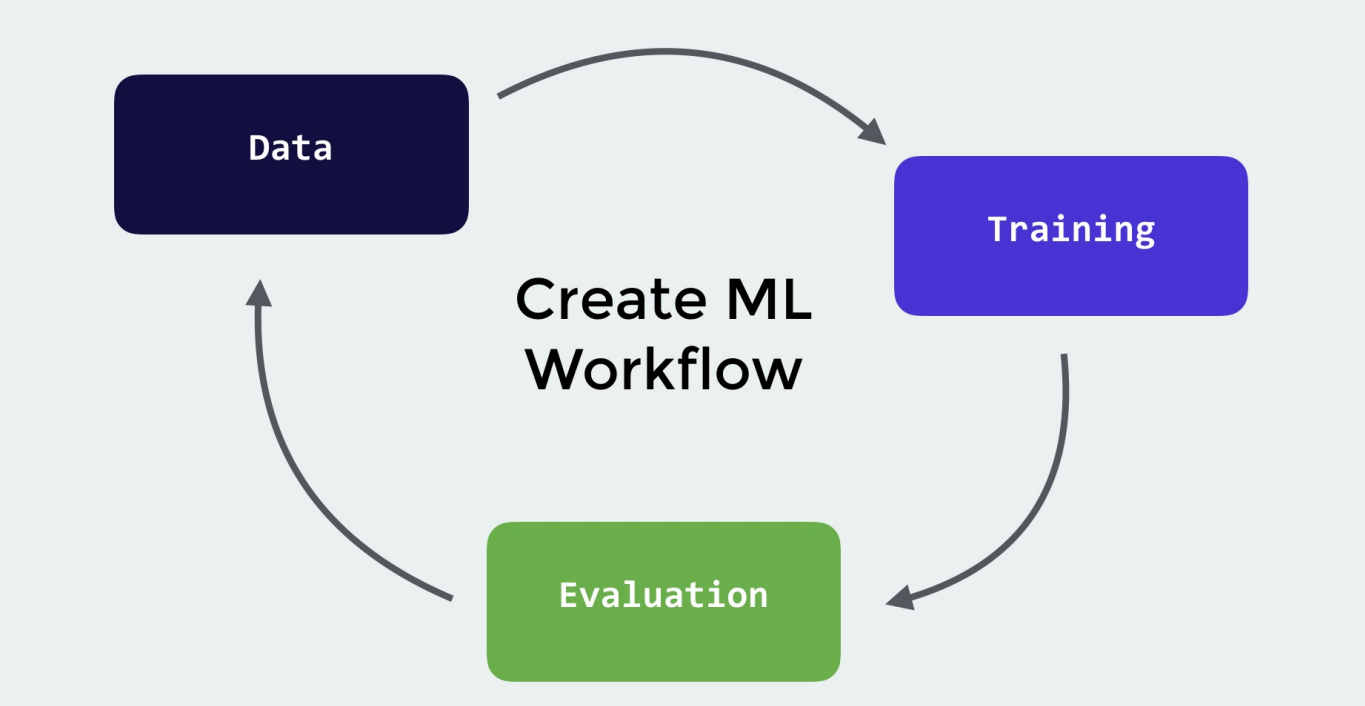


Figure 43.1. The workflow of creating a ML model

In order to create the model, you start by collecting the data. Say, you want to train a model for classifying bread according to types. You take photos of different bread and then train the model.

Next, you use some test data to evaluate the model and see if you are happy with the result. If not, you go back to collect more data, fine tune them and train the model again until the results are satisfactory.

Finally, you export the ML model and integrate it into your iOS app. This pretty much sums up the workflow of creating a ML model.

Requirements of Using Create ML

As mentioned at the very beginning, we use a new tool called Create ML to create and train ML models. This tool is built into Xcode 11 and requires macOS Catalina (or macOS 10.15) to run. If you are running Xcode 11 on older version of macOS, please make sure you upgrade your macOS to 10.15 in order to follow the rest of the chapter.

Creating an Image Classifier Model

In this chapter, I will show you how to train an image classifier using Create ML. This image classifier model is very simple. It is designed to recognize "Dim Sum", a style of Chinese cuisine. We are not going to train a model that identifies all sorts of dishes. Of course, you can do that, but for demo purpose, we will only focus on recognizing a couple of "Dim Sum" dishes such as Steamed Shrimp Dumpling and Steamed Meat Ball. That said, you will have an idea about how to train your own image classifier model with Create ML.

Preparing the data

The first step of training a ML model is data collection. In this case, we need to collect some photos of the "Dim Sum" dishes. For me, I dined in a Chinese restaurant, ordered some dim sum, and took some photos of them. Alternatively, you can try to collect "Dim Sum" photos on the web (please be aware of the copyright issue). To follow the demo, please download the image pack from https://www.appcoda.com/resources/swift42/CreateMLImages.zip.

After you unpack the archive, you should find two folders: *training* and *testing*. In order to use Create ML for creating your own ML model, you need to prepare two sets of data: *one for training* and *the other one for testing*. Here, the *training* folder contains all the images for training the model, while the *testing* folder contains the images for evaluating the trained model.

If you look into the *training* folder, you will find a set of sub-folders. The name of the sub-folder is the label of a dim sum dish (e.g. Steamed Shrimp Dumpling). In that sub-folder, it contains around 10-20 images of that particular dim sum dish. This is how you prepare the training data. You create a sub-folder, put all the images of a dim sum dish in the folder, and set the folder's name to the image label.

Figure 43.2. A sample training data of Steamed Meat Ball

The structure of the *testing* folder is similar to that of the *training* folder. You assign the test images into different sub-folders. The name of sub-folders is the expected label of the test images.

You may wonder how many images should you prepare for training? In general, the more samples you provide the better is the accuracy.

> Create ML leverages the machine learning infrastructure built in to Apple products like Photos and Siri. This means your image classification and natural language models are smaller and take much less time to train.
>
> - Apple's documentation (https://developer.apple.com/documentation/create_ml)

As Apple recommended, you should have at least 10 images for each object. It is also suggested to provide photos of the object taken from various angles and with different background. This would improve the accuracy of the trained ML model.

Training the Model Using CreateML

With the training data, the next step is fire up Xcode and use Create ML to train the model. Create ML is built right into Xcode. To launch the app, go up to the Xcode menu and select *Xcode* > *Open Developer Tool* > *Create ML*. Upon launch, choose *File* > *New Project* to create a new project and you will be asked to choose a ML model template.

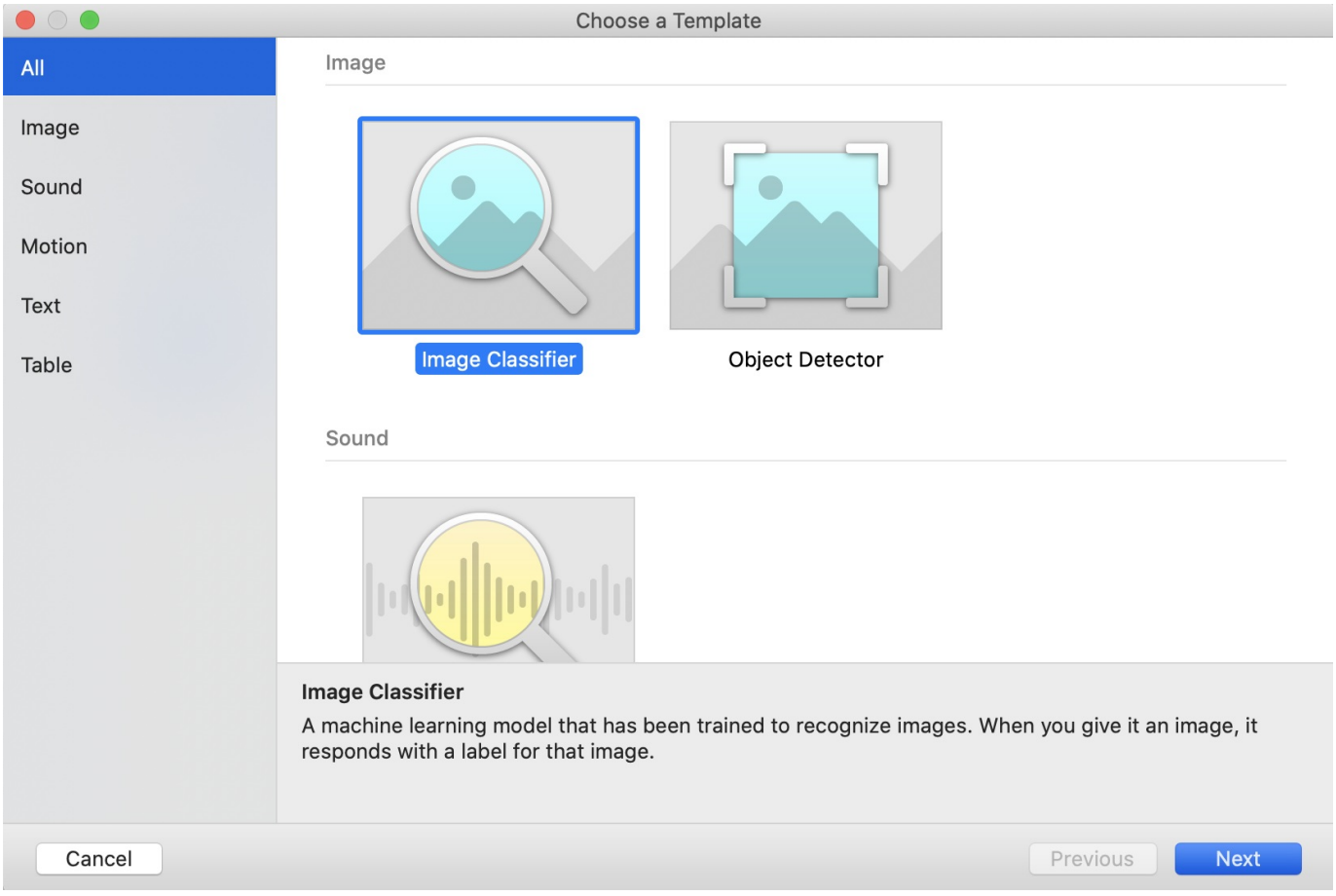


Figure 43.3. Choose a ML model template

Since we are going to train a ML model for classifying images, let's choose the *Image Classifier* template. In the next screen, name the project *DimSumClassifier*. Optionally, you can change the author and description of the model. Next, save it to your preferred folder. You will then see the UI of the image classifier builder.

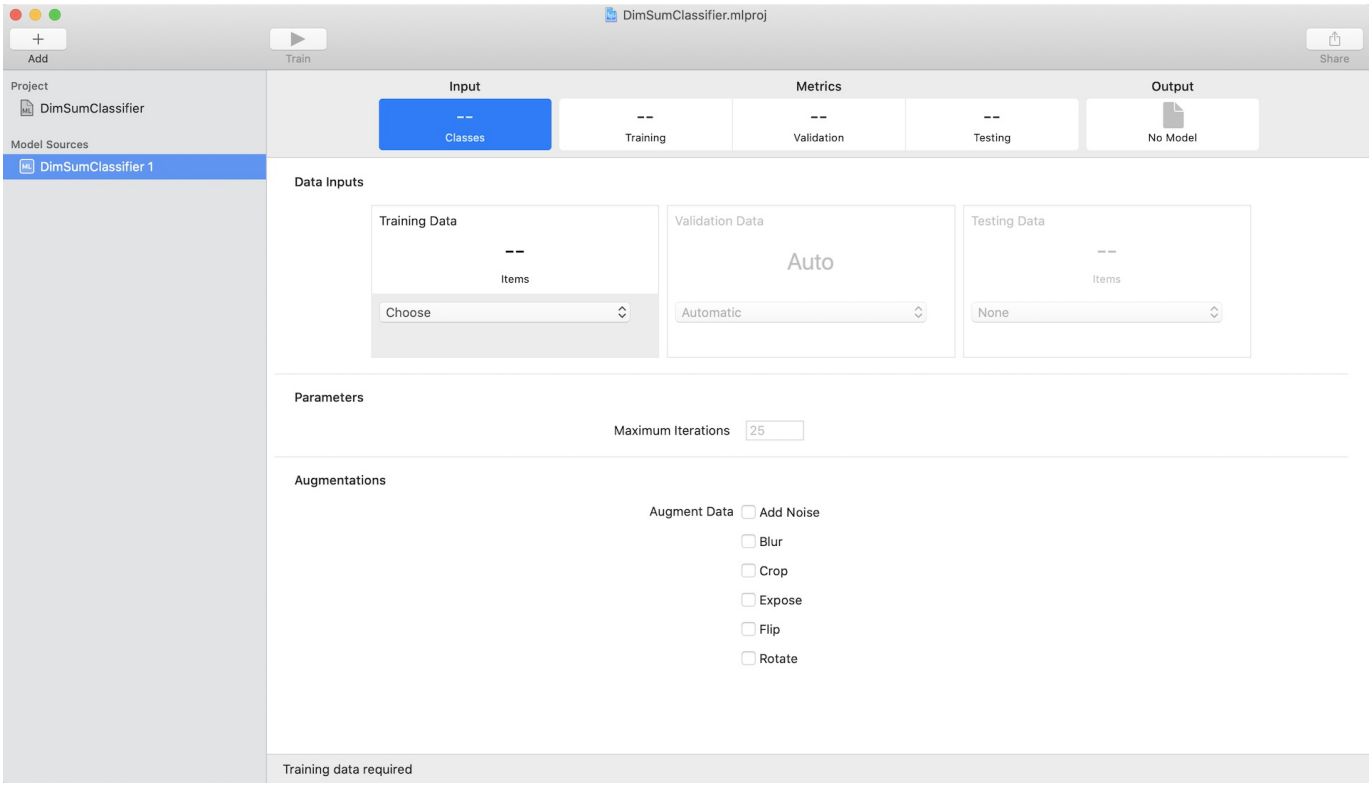


Figure 43.4. The image classifier builder

Normally, you don't need to change the setting of the image classifier builder such as maximum iterations and augmentations. To begin training your ML model, all you need to do is specify the *training* folder, which should point it to the *training* folder of the image pack you downloaded earlier. For the Validation Data, you can keep it to *automatic* and let the CreateML tool choose the validation data randomly. For the Testing Data, please set it to the *testing* folder of the image pack. If you've made everything correct, you should see a screen similar to figure 43.5. The tool automatically recognizes the training and testing data. The image pack includes 107 images with 5 classes (i.e. 5 different types of dim sum) for training.

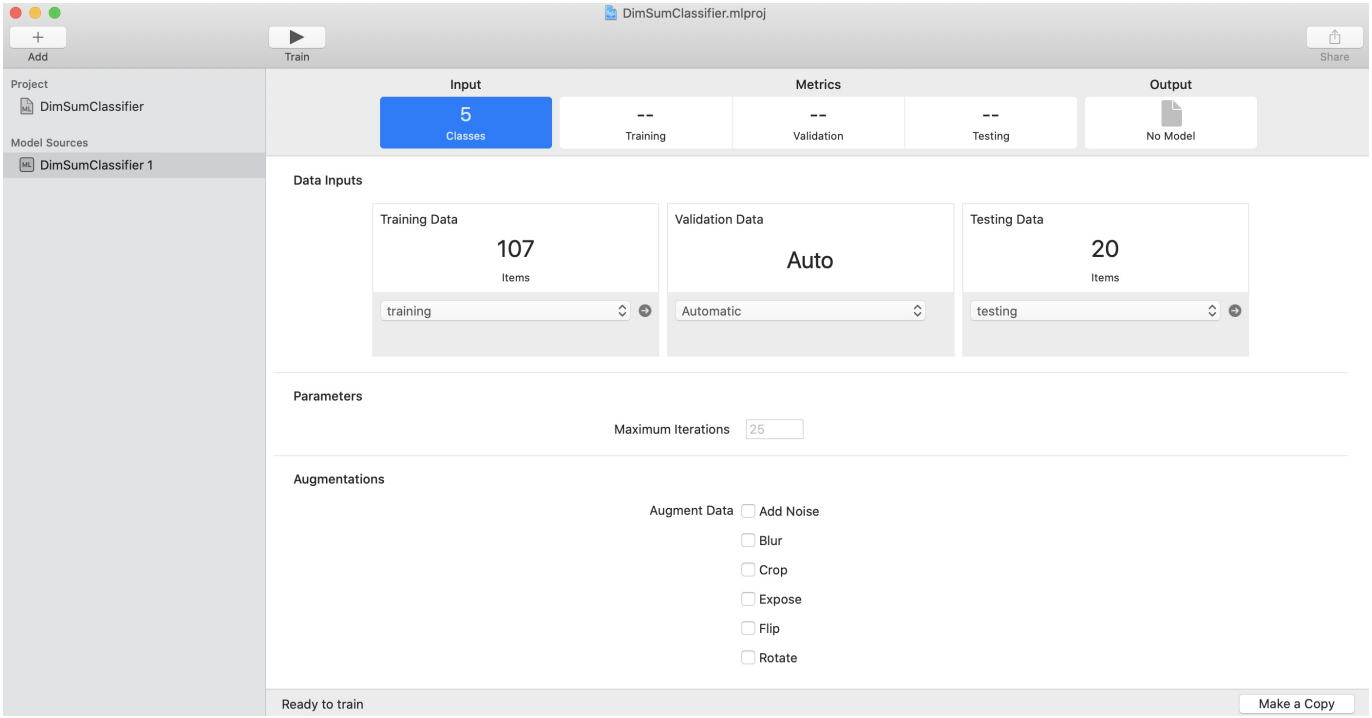


Figure 43.5. Drop the training folder to start training

Next, what you need to do is to hit the *Play* button to start the training. Once the training completes, the tool should show you the training result. Training indicates the percentage of training data Xcode was successfully able to train. It should normally display 100%. If it's less than 100%, this means Create ML couldn't extract an identifiable feature from the image.

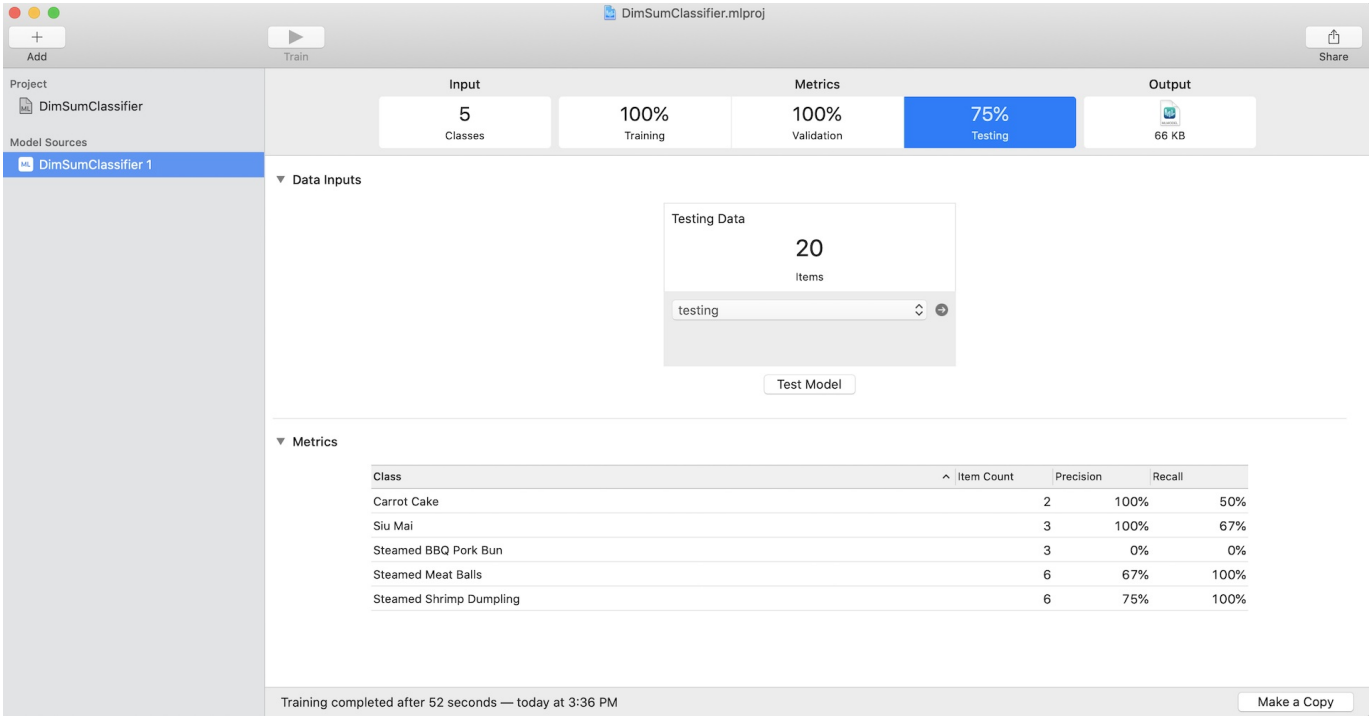


Figure 43.6. Training in progress

During the training, Create ML randomly selects around 10% of the training data for validation purpose. In the demo above, we achieve 100% accuracy.

How well does the trained ML model perform? After training, we need to test the accuracy of the ML model. That's why we need to provide the testing data set. In this case, the testing data folder contains the dim sum images that the model has never seen before. During testing, the ML model makes prediction on the test images. Create ML compares each prediction and compute the overall testing accuracy. In the demo, the overall testing accuracy is 75%.

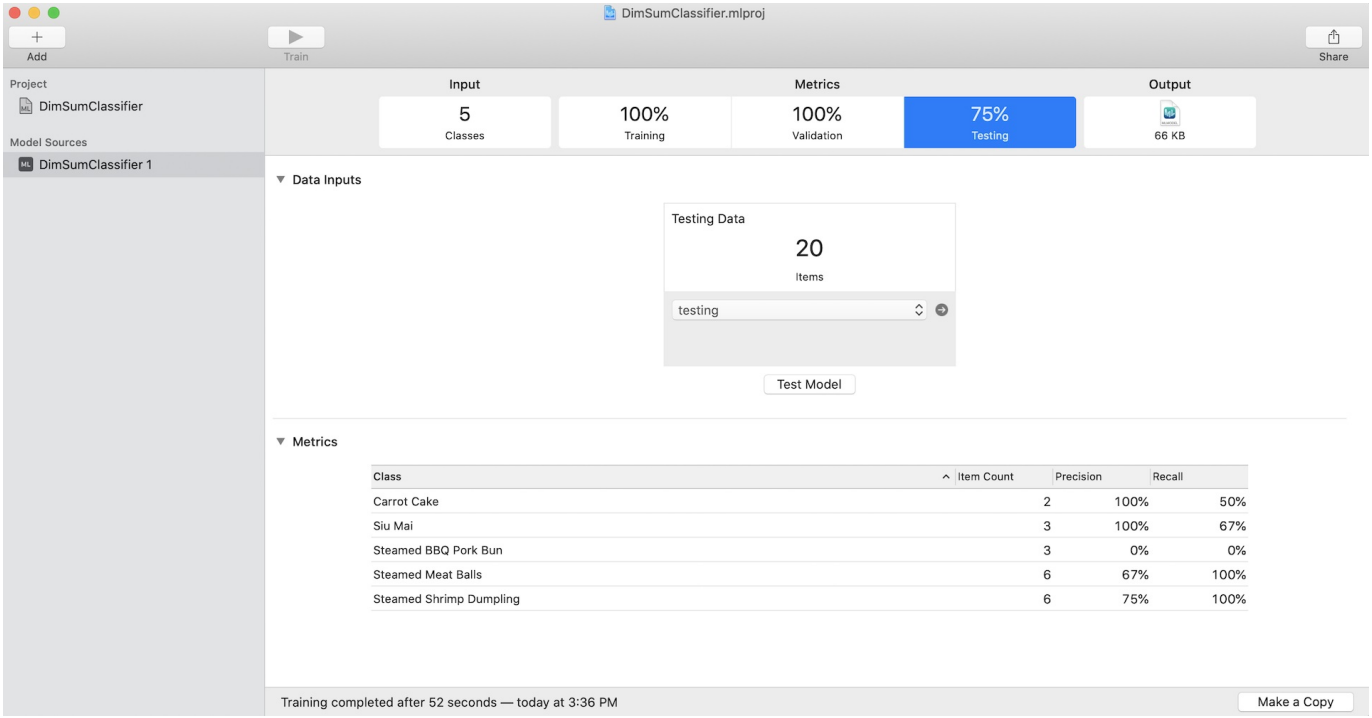


Figure 43.7. Testing Accuracy of the ML model

It's not good enough. What you can do is re-train the model with more training data. And you may vary the parameters of the training (e.g. iteration) to improve the accuracy.

On the other hand, if you think the result is good enough for your application, you can export the ML model for further integration. Simply drag the model icon from the *Output* to a folder in Finder. That's it!

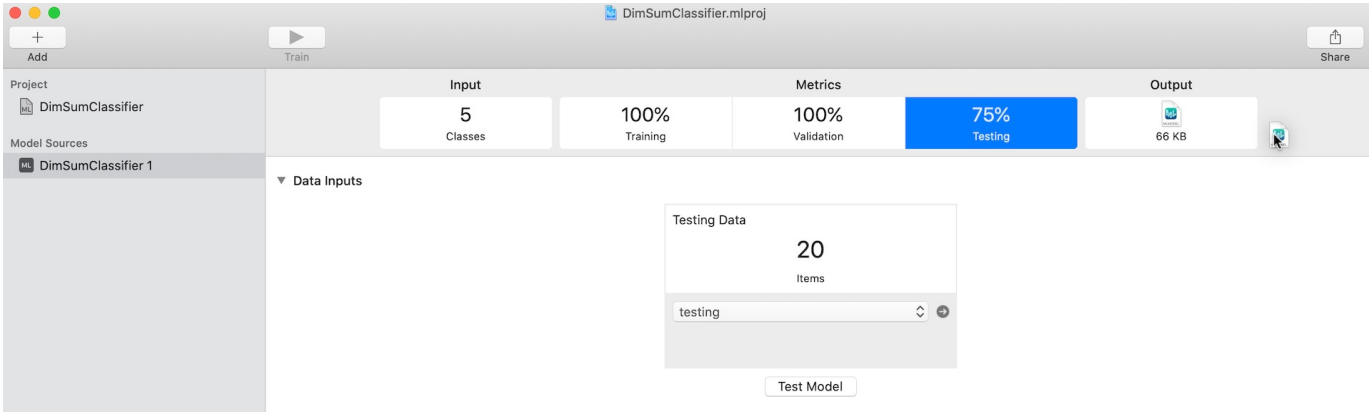


Figure 43.8. Save your ML model

One thing I want to highlight is the *Output* tab, you can perform a quick test to see how the model performs on some unseen photos. Simply drag and drop a test image to the empty pane to see the result.

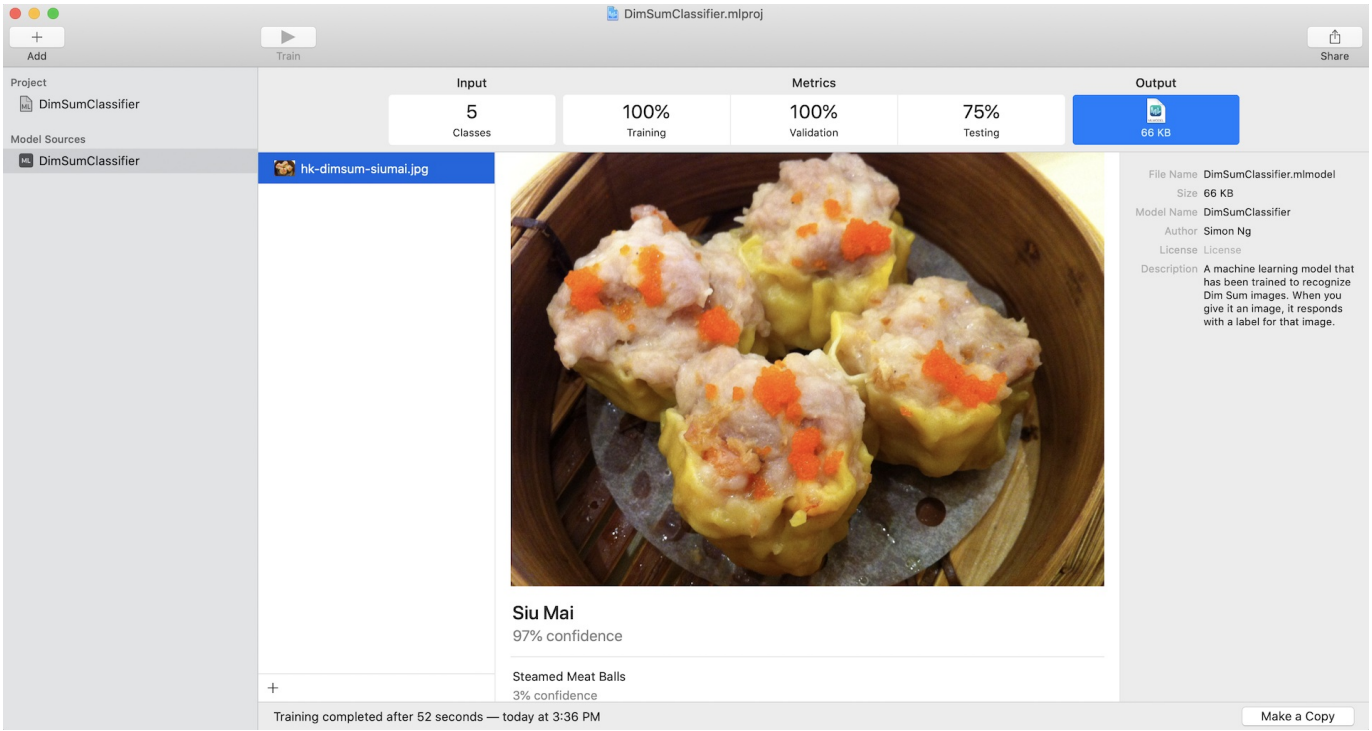


Figure 43.9. Testing the model accuracy by using drag & drop

You are now ready to use the ML model file and integrate it in your iOS app. Say, for the demo app you built in chapter 40, you can replace `MobileNetV2.mlmodel` with `DimSumClassifier.mlmodel`. This will allow the Image Recognition app to identify dim sum. Give it a try and see what you get!

Training the Model Programmatically

Before we move onto the next chapter to show you how to train a text classifier, let's take a look at an alternative way to create the image classifier model. Earlier, we used the Create ML tool to train the ML model. You can drag and drop the training and test data through a graphical user interface.

While the visual builder provides an intuitive way to train your model, it is probably not the best way for developers. You may want to do things programmatically. Instead of dragging and dropping the training data, you may want to read the data directly from a specific folder.

To create a ML model programmatically, open a new Playground project using a Blank template of macOS. And, insert the following code:

```
import CreateML
import Foundation

let trainingDir = URL(fileURLWithPath: "/Users/simon/Downloads/DimSum/training")
let testingDir = URL(fileURLWithPath: "/Users/simon/Downloads/DimSum/testing")

// Train the model
let model = try MLImageClassifier(trainingData: .labeledDirectories(at: trainingDi
r))

// Evaluate the trained model with test data
let evaluation = model.evaluation(on: .labeledDirectories(at: testingDir))

// Export the model
try model.write(to: URL(fileURLWithPath: "/Users/simon/Downloads/DimSumImageClassi
fier"))
```

Depending on the directory of your training and test data, you will need to modify the file path of the above code to your own path. For my demo, I put all the training data in the *Downloads* folder.

The first two lines of code set the file path of the training and test data. We then create the `MLImageClassifier` object and use our training data to train the model. Once the model is created, we call up the `evaluation` method of the model to evaluate the accuracy of the ML model. Lastly, we export the model to a specific folder.

That's how you train the ML model without using the visual builder. When you execute the code, you should see the status messages and can reveal the validation accuracy in the console. When finished, the `DimSumImageClassifier.mlmodel` file will be saved in the specified folder.

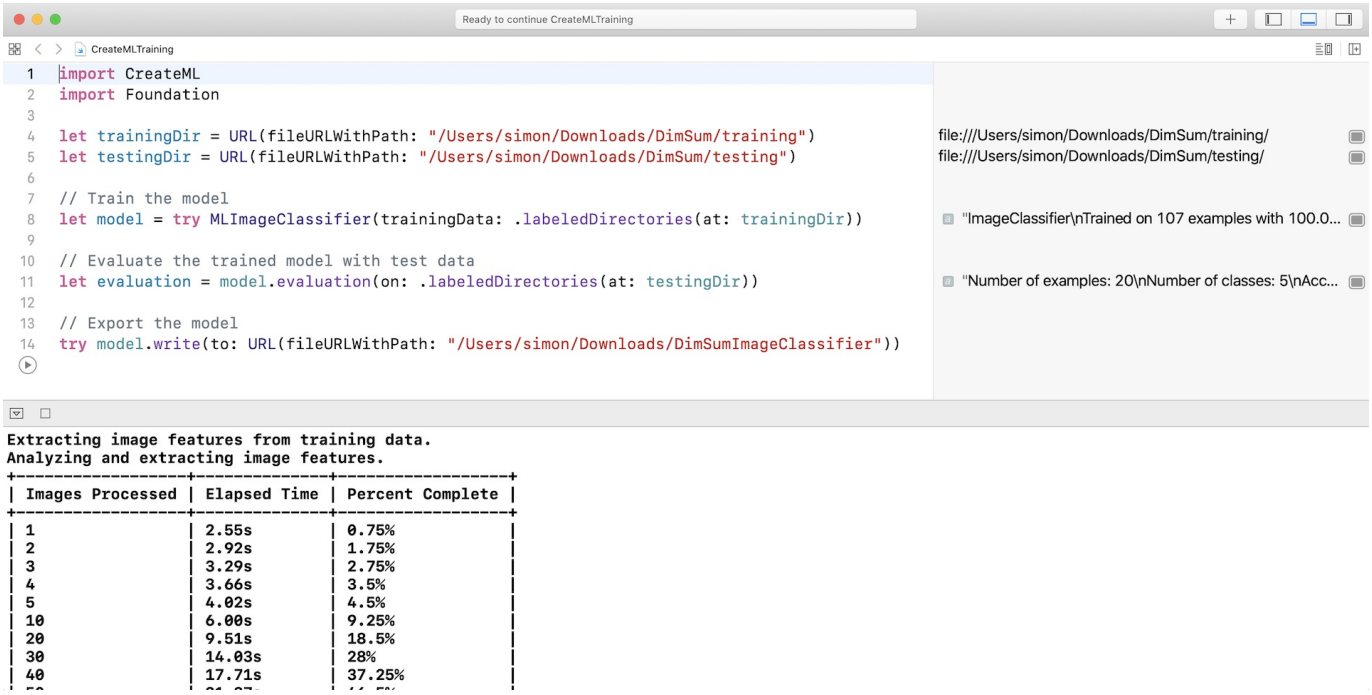


Figure 43.10. Training the ML model using code

As a developer, I prefer the programmatical approach for training the ML model. This allows you to automate the whole training process. You can just update the training/test data and then re-run the program to create the updated ML model.

Conclusion

In this chapter, you saw how to create your own machine learning models using Apple's newest framework Create ML. With just a few lines of code, you can create advanced, state-of-the-art machine learning algorithms to process your data and give you the results you want.

To learn more about Create ML, you can watch Apple's video on the Create ML app:

- https://developer.apple.com/videos/play/wwdc2019/430/

Chapter 44
Building a Sentiment Classifier Using Create ML to Classify User Reviews

In the previous chapter, I have walked you through the basics of Create ML and showed you how to train an image classifier. As mentioned, Create ML doesn't limit ML model training to images. You can use it to train your own text classifier model to classify natural language text.

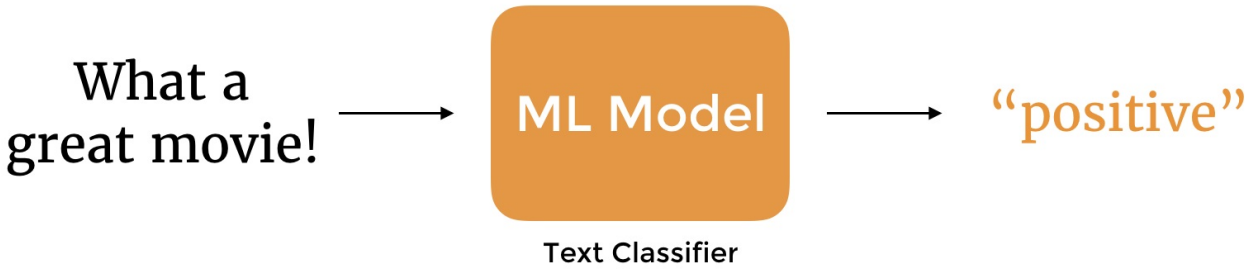


Figure 44.1. How the text classifer works

What we are going to do in this chapter is to create a sentiment model for classifying product and movie reviews. The trained model takes in opinions in natural language text, analyze, and classifies it into positive/negative reviews. Furthermore, I will show you how to compile and test the model in Playgrounds.

Before you move on, please make sure you check out the previous two chapters. I assume you already equip yourself with the basics of Create ML and Core ML.

Data Preparation

The workflow of creating a text classifier is very similar to that of an image classifier. You begin with data collection. Since we are going to create a sentiment classifier, we have to prepare tons of examples of product/movie/restaurant reviews (in natural language) to train the ML model. For each of the reviews, we label it as *positive* or *negative*. This is how we train the machine to understand and differentiate positive/negative reviews.

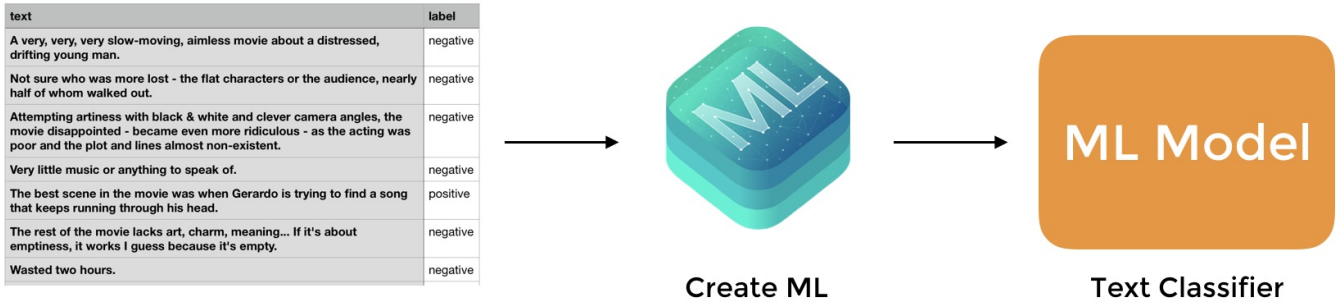


Figure 44.2. Training the ML model by showing it lots of examples

But where can you find the examples of the product/movie/restaurant reviews?

One way is to write the sample reviews by yourself and label each of the reviews like this:

```
"I like this movie. It's really good!", positive
"This coffee machine sucks!", negative
```

If your customers give you regular feedback on your products, you can also use the reviews as the training data. Alternatively, there are a plenty of websites like amazon.com, yelp.com, and imdb.com that you can refer to for retrieving sample user reviews. Some of the websites offer APIs (like Yelp) for you to request the reviews. The other common approach is to extract and collect the user reviews through web scraping. I will not go into the details of web scraping in this book, but you can refer to the following reference if you are interested in learning more:

- How to scrape websites with Python and BeautifulSoup (https://medium.freecodecamp.org/how-to-scrape-websites-with-python-and-beautifulsoup-5946935d93fe)
- How to scrape Amazon Reviews using Python (https://www.scrapehero.com/how-to-scrape-amazon-product-reviews/)

In this demo, we will use the sample data, provided by Dimitrios Kotzias, from the UCI Machine Learning Repository:

- Sentiment Labelled Sentences Data Set

(https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences)

This data set contains a total of 3,000 sample reviews from amazon.com, yelp.com, and imdb.com. The creator already labelled all the reviews with either 1 (for positive) or 0 (for negative). Here is the sample content:

```
But I recommend waiting for their future efforts, let this one go.      0
Service was very prompt.    1
I could care less... The interior is just beautiful.    1
```

This is pretty cool, so instead of preparing our own data, we will use this data set to train the sentiment classifier. However, before opening Playgrounds to train the model, we will have to alter the data a bit in order to conform to the requirement of Create ML.

Create ML has a built-in support for tabular data. If you look into the data set closely, it is actually a table with two columns. The first column is the sentence (or the user review), while the second column indicates whether the corresponding sentence is positive or negative. The new framework introduces a new class called `MLDataTable` to import a table of training data. It supports two types of data format including JSON and CSV. In this case, we will convert the data file to CSV format like this:

```
text,label
"But I recommend waiting for their future efforts, let this one go.",negative
"Service was very prompt.",positive
"I could care less... The interior is just beautiful.",positive
.
.
.
```

The first line of the file is the column labels. Here we name the first column *text* and the second column *label*. It is required to give the columns of data a name. Later, when you import the data using `MLDataTable`, the resulting data table will have two columns, named "text" and "label". Both names serve as a key to access the specific column of the data.

There are various ways to perform the conversion. You can manually modify the file's content and convert it to the desired CSV format. Or you can open TextEdit and use its "Find & Replace" function to replace the tab character with a comma. As a practice, I suggest you think of your own approach to handle the text conversion.

For me, I prefer to use `sed`, a stream editor for performing basic text transformation, to create the CSV file from the data files. On macOS, you can run `sed` in Terminal using the command like this:

```
echo "text,label" > sample_reviews.csv

sed "s/\"/'/g; s/    0/\",negative/g; s/        1/\",positive/g; s/^/\"/g" imdb_la
belled.txt >> sample_reviews.csv

sed "s/\"/'/g; s/      0/\",negative/g; s/      1/\",positive/g; s/^/\"/g" amazon_
cells_labelled.txt >> sample_reviews.csv

sed "s/\"/'/g; s/      0/\",negative/g; s/      1/\",positive/g; s/^/\"/g" yelp_la
belled.txt >> sample_reviews.csv
```

The first command uses `echo` to write the column names and create the `sample_reviews.csv` file. The next three commands are very similar, except that the text transformation applies to different files.

In the commands above, I use `sed` to execute 4 replacement patterns at once and then output the transformed content to `sample_reviews.csv`:

1. Replace all double quotes with single quotes.
2. Replace "0" with ",negative". As a side note, to type the *tab* character in Terminal, press `control` + `v` and then press `tab`.
3. Replace "1" with ",positive".
4. Insert a double quote at the beginning of each sentence.

Once you execute the commands, `sed` will convert the files accordingly. Here is the excerpt of the resulting CSV file:

```
text,label
"A very, very, very slow-moving, aimless movie about a distressed, drifting young man.",negative
"Not sure who was more lost - the flat characters or the audience, nearly half of whom walked out.",negative
"Attempting artiness with black & white and clever camera angles, the movie disappointed - became even more ridiculous - as the acting was poor and the plot and lines almost non-existent.",negative
"Very little music or anything to speak of.",negative
"The best scene in the movie was when Gerardo is trying to find a song that keeps running through his head.",positive
.
.
.
.
```

In case you have problems converting the data files, you can download the final CSV file from http://www.appcoda.com/resources/swift5/createml-sample-reviews.zip.

Create and Train the Text Classifier Using MLTextClassifier

Now that you've prepared the training data, let's begin to train the ML model. In this demo, I will show you how to train the model programmatically using MLTextClassifier.

Similar to what we've done before, create a new Playground using the *Blank* template. Please make sure you choose the blank template under macOS. Name the Playgrounds file to whatever you like. Here I set its name to *SentimentTextClassifier*.

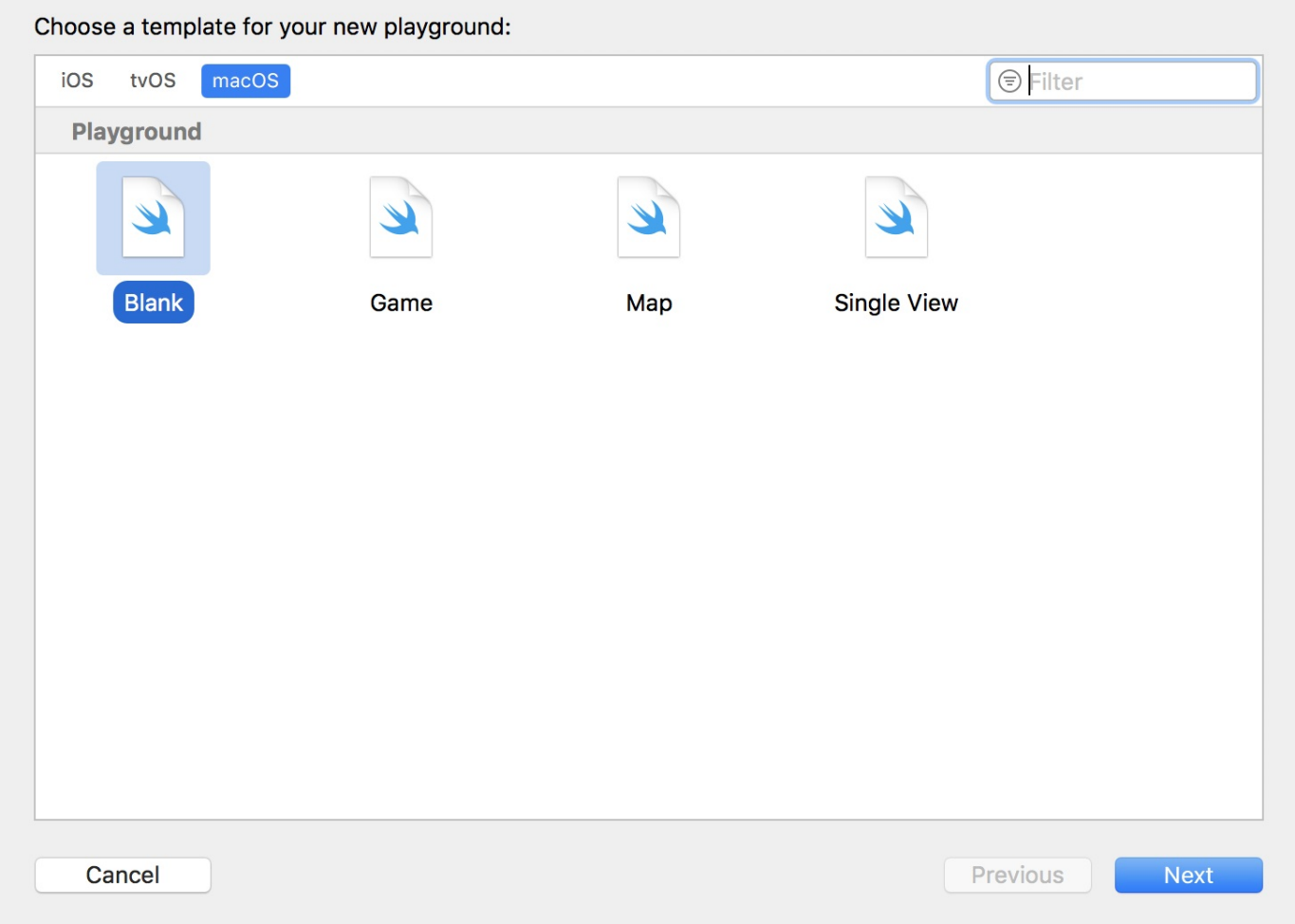


Figure 44.3. Creating a new Playground using the Blank template

Once created, replace the content with the following code:

```
import CreateML
import Foundation

let dataPath = URL(fileURLWithPath: "/Users/simon/Downloads/sample_sentences/sampl
e_reviews.csv")

// Load the data set
let dataSet = try MLDataTable(contentsOf: dataPath)
```

We first import the `CreateML` framework and then load the `sample_reviews.csv` file into a `MLDataTable` object. Your file path should be different from mine. Therefore, please make sure you replace the path with your own path.

So far, we only prepared the training data. You may wonder why we do not need to prepare the test data for the text classifier?

In fact, we need to have the test data for evaluation purpose. However, instead of arranging another data set for testing, we will derive the test data from the data set of `sample_reviews.csv`. To do that, insert the following line of code in the Playgrounds project:

```
let (trainingData, testingData) = dataSet.randomSplit(by: 0.8, seed: 5)
```

The `randomSplit` method divides the current data set into two sets of data. In the code above, we specify the value of the `by` parameter to `0.8`. This means 80% of the original data will be assigned as the training data. The rest of it (i.e. 20%) is for testing.

Now that both training and test data are all set, it is ready to train the text classifier for classifying the user reviews into sentiment. Continue to insert the following code:

```
let textClassifier = try MLTextClassifier(trainingData: trainingData, textColumn:
"text", labelColumn: "label")
```

To train a text classifier, we use `MLTextClassifier` and specify the training data. In addition to that, we have to tell `MLTextClassifier` the name of the text column and the label column. Recall that we set the column of the user reviews to "text" and that of the sentiment to "label", this is why we pass these two values in the method call.

This is the line of code you need to create and train a ML model for classifying natural language text. If you execute the code, the model training will begin right away. Once finished, you can reveal the accuracy of the model by accessing the `classificationError` properties of the model's `trainingMetrics` and `validationMetrics` properties like this:

```
// Find out the accuracy of the ML model in percentage
let trainingAccuracy = (1.0 - textClassifier.trainingMetrics.classificationError)
* 100
let validationAccuracy = (1.0 - textClassifier.validationMetrics.classificationErr
or) * 100
```

It's easy to understand what training accuracy is, but you may wonder what validation accuracy means. If you print out the value of `trainingAccuracy`, the ML model got a 99.96% training accuracy. That's pretty awesome!

During training, Create ML put aside a small percentage (~10%) of the training data for validating the model during the training phase. In other words, 90% of the training data is used for training the model. The rest of it is assigned for validating and fine-tuning the model. So, the validation accuracy indicates the performance of the model on the validation data set. If you run the code, the model achieves an 81% validation accuracy.

How does the model perform on some "unseen" data set?

This comes to the next phase of ML model training. We will provide another set of data known as test data set to evaluate the trained model. Insert the following lines of code in the Playgrounds project:

```
let evaluationMetrics = textClassifier.evaluation(on: testingData)
let evaluationAccuracy = (1.0 - evaluationMetrics.classificationError) * 100
```

To start the evaluation, you invoke the `evaluation` method of the text classifier with your test data. Then you can find out the evaluation accuracy. If you are satisfied with the evaluation accuracy, you can then export the model and save it as `SentimentClassifier.mlmodel`.

Here is the required code for saving the ML model:

```
if evaluationAccuracy >= 80.0 {
    let modelInfo = MLModelMetadata(author: "Simon Ng", shortDescription: "A train
ed model to classify review sentiment", license: "MIT", version: "1.0", additional
: nil)
    try textClassifier.write(to: URL(fileURLWithPath: "/Users/simon/Downloads/Sent
imentClassifier.mlmodel"), metadata: modelInfo)
}
```

Running the code will result in the following output in the console:

```
Finished parsing file /Users/simon/Downloads/sample_reviews.csv
Parsing completed. Parsed 100 lines in 0.032053 secs.
Finished parsing file /Users/simon/Downloads/sample_reviews.csv
Parsing completed. Parsed 3000 lines in 0.009207 secs.
Tokenizing data and extracting features
20% complete
40% complete
60% complete
80% complete
100% complete
Starting MaxEnt training with 2397 samples
Iteration 1 training accuracy 0.494785
Iteration 2 training accuracy 0.908219
Iteration 3 training accuracy 0.922820
Iteration 4 training accuracy 0.961202
Iteration 5 training accuracy 0.992908
Iteration 6 training accuracy 0.998748
Iteration 7 training accuracy 0.999166
Iteration 8 training accuracy 0.999583
Finished MaxEnt training in 0.07 seconds
Trained model successfully saved at /Users/simon/Downloads/SentimentClassifier.mlm
odel.
```

Testing Out the Model in Playgrounds

Now that you've created your ML model for classifying sentiment, you can integrate the model file into your iOS apps to test out the model. But sometimes you want to test it in Playgrounds.

In order to do that, you have to first compile the ML model file using the command below:

```
xcrun coremlc compile SentimentClassifier.mlmodel .
```

By running the command in Terminal, it will compile the model and result the `SentimentClassifier.mlmodelc` bundle, which is actually a folder. To use the compiled model, create another Blank Playground project and add the `SentimentClassifier.mlmodelc` bundle to the `Resources` folder of your Playground.

Next, replace all the generated code with the following code snippet:

```
import NaturalLanguage

let sampleReviews = ["I probably haven't been hooked to a TV show like I am to Bre
aking Bad before. This beautiful piece of art is incredibly well written and direc
ted, furthermore the actors are doing a tremendous job! ",
                     "I don't believe there has ever been anything like 'Game of Th
rones' on TV. The sheer amount of quality and talent in this series is staggering.
 The actors (and I mean really ALL the actors), the costumes, the visual effects,
the make-up: everybody working on this show seems to have wanted to make Televisio
n-history. And the writing is just phenomenal.",
                     "We have found this machine to be extremely inconsistent.",
                     "I was very disappointed in the structure of the book. I didn'
t get any useful information and will be returning it. If I've waited too late to
return it because I am working all the time to become financially independent, I w
ill sell it at my next garage sale."
]

if let modelURL = Bundle.main.url(forResource: "SentimentClassifier", withExtensio
n: "mlmodelc") {

    let model = try! NLModel(contentsOf: modelURL)

    for review in sampleReviews {
        print(model.predictedLabel(for: review) ?? "")
    }
}
```

Here, we initialize several sample reviews for testing purpose. To load the bundle, you can call `Bundle.main.url` with the `SentimentClassifier.mlmodelc` file. The `NaturalLanguage` framework provides a class named `NLModel` for developers to integrate custom text classifier. In the code above, we initialize an `NLModel` object with the sentiment classifier, and then call `predictedLabel(for:)` to classify the sample reviews.

Running the code in Playground will give you the following result in the console.

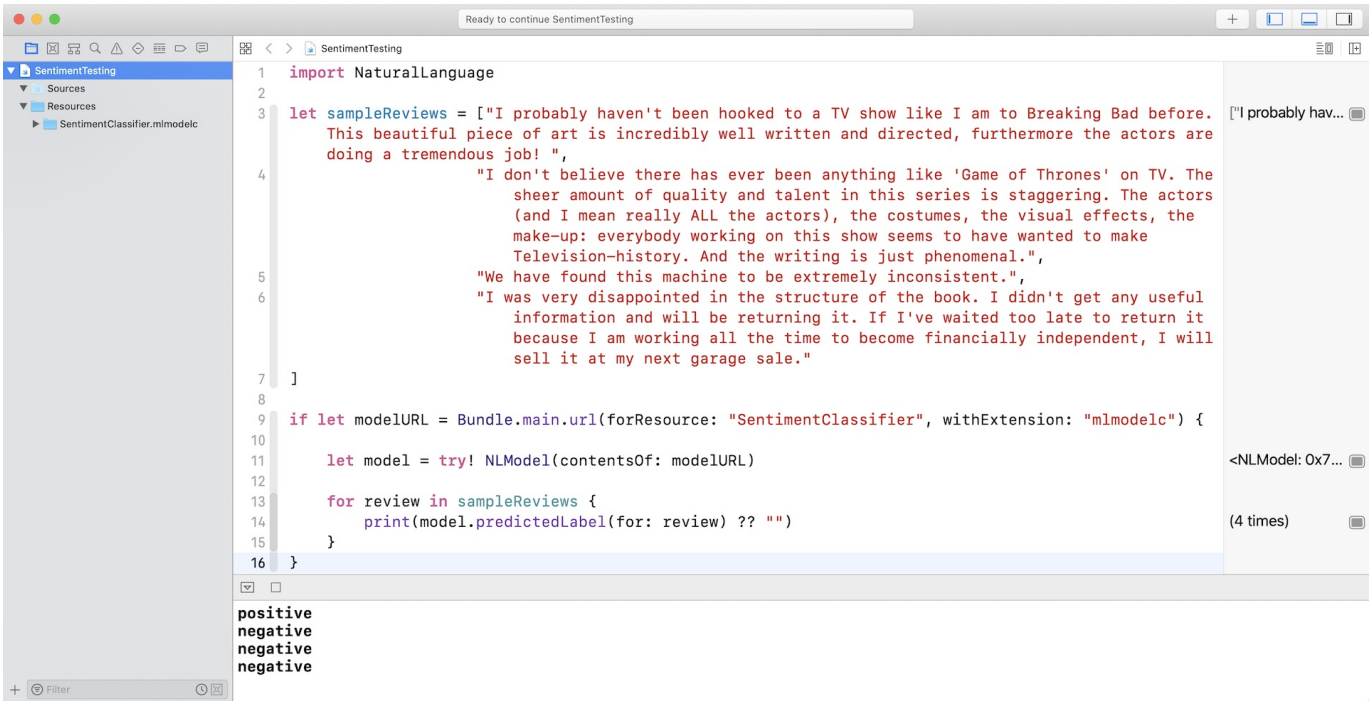


Figure 44.4. The sentiment result is displayed in the console

Training the Model Using Create ML

Earlier, I showed you how to train the model programmatically. If you prefer to use the Create ML app, you can achieve the same result. All you need to do is to launch the Create ML app and create a new project using the *Text Classifier* template.

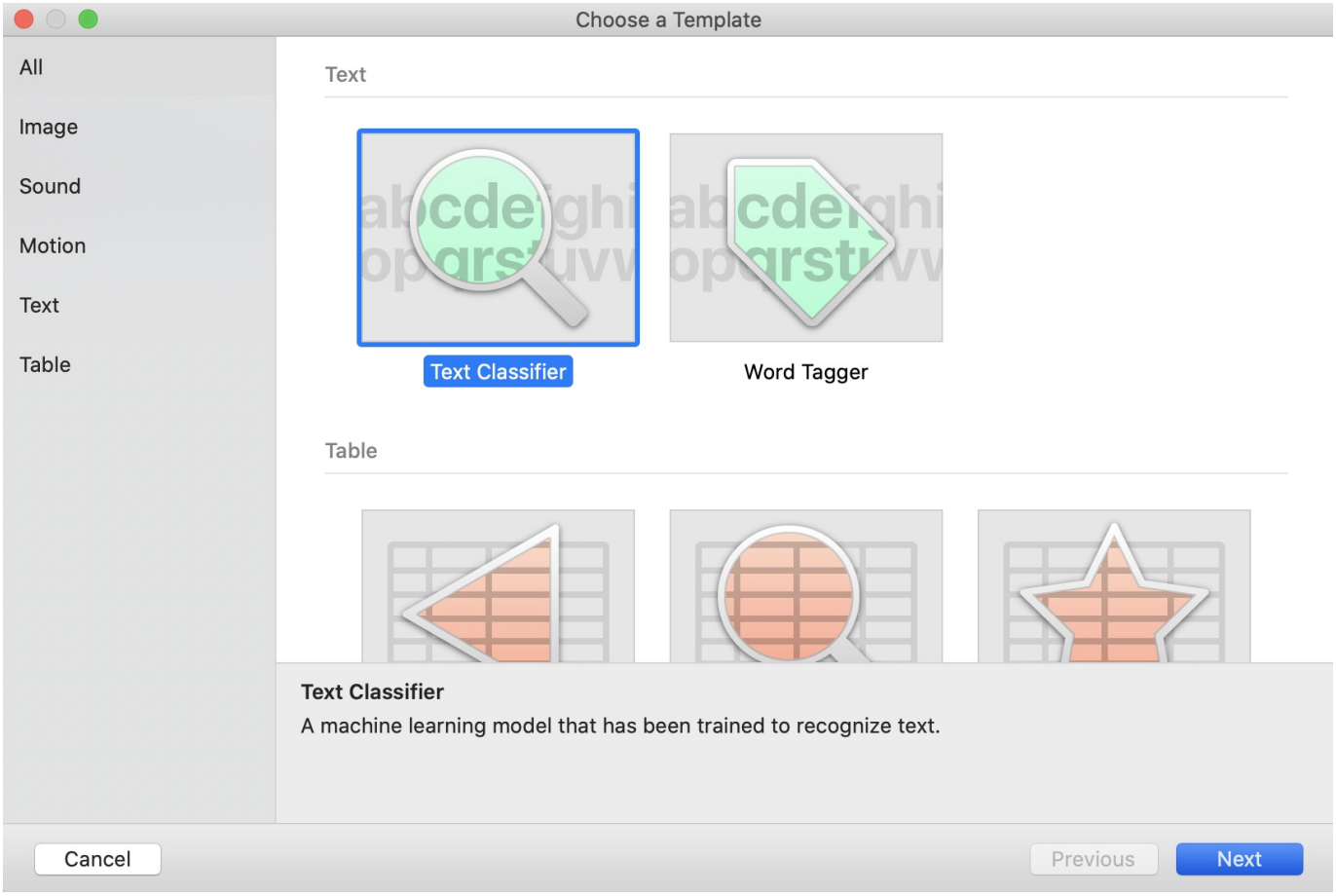


Figure 44.5. The Text Classifier template

You can then assign the training and test data to the project, similar to what we have done in the previous chapter.

Your Exercise

Now that you've created a trained ML model for sentiment analysis and tested it in Playgrounds, wouldn't it be great to integrate it in an iOS app? This is an exercise I really want you to work on.

The app is very simple. It allows users to input a paragraph of text in a text view. When the user hit the *return* key, the app will analyze the text and classify the message. If the message is positive, it displays the emoji . Conversely, it shows the thumb down emoji if the user's message is negative.

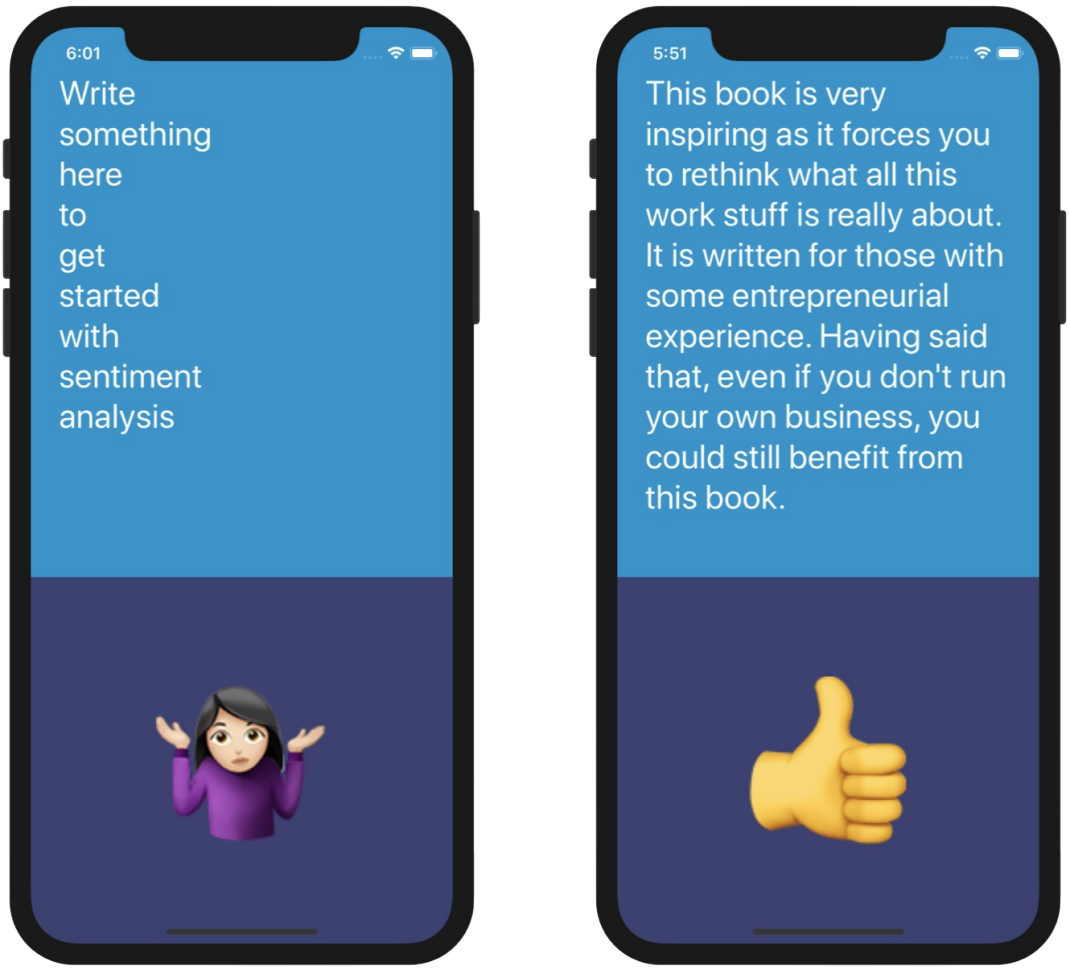


Figure 44.6. Sample screens of the Sentiment Analysis app

To integrate the trained ML model into an app, all you need to do is add the `SentimentClassifier.mlmodel` file into your iOS project. The code for using the ML model is exactly the same as that we used in the Playground project. Don't skip this exercise and take some time to work on it.

Conclusion

As you can see, training a text classifier is very similar to training an image classifier, that we have done before. Create ML has provided developers an easy way to create our own model. You don't need to write a lot of code, but just a few lines. The tool empowers developers to build features that couldn't be built before.

Just consider the demo we built in this chapter, it's impossible to use pattern matching to find out an accurate sentiment of a user review. Now, all you need to do is collect the data and train your own model in Playgrounds. You will then have a ML model for you to build a sentiment classification feature in your iOS app. This is pretty amazing.

For reference, you can download the Playground project from http://www.appcoda.com/resources/swift5/SentimentTextClassifier.playground.zip.

For the solution of the exercise, you can download it from http://www.appcoda.com/resources/swift5/SentimentClassifierDemo.zip

Chapter 45
Working with Image Tracking in ARKit

Starting from iOS 12, developers can use the back-facing camera to detect and track known images by using an API called `ARImageTrackingConfiguration`. What does it mean to you? If you've watched the Harry Potter movie before, you probably remember the wizarding newspaper with animated images in print. With the image tracking configuration, we can turn this into a reality through an ARKit app.

In this chapter, you will learn how to use `ARImageTrackingConfiguration` to perform image tracking and build an AR newspaper app similar to that appeared in the Harry Potter movie. It's not easy to describe how the app works in words. So, please head over to this link (https://link.appcoda.com/arnewspaper) and see the demo app in action. When the

app recognizes a specific photo, it overlays the image with a video. As you can see in the video, what's amazing is that ARKit tracks the position change of the image and keep anchoring the video content.

Looking interesting, right? Let's get started.

Creating a New Project Using ARKit template

We will build this project from scratch. Assuming you've opened Xcode, you can start by creating a new project. Choose the *Augmented Reality* template and click Next. For the project information, name the project *ARNewspaper* and make sure you set the *Content Technology* to *SceneKit*.

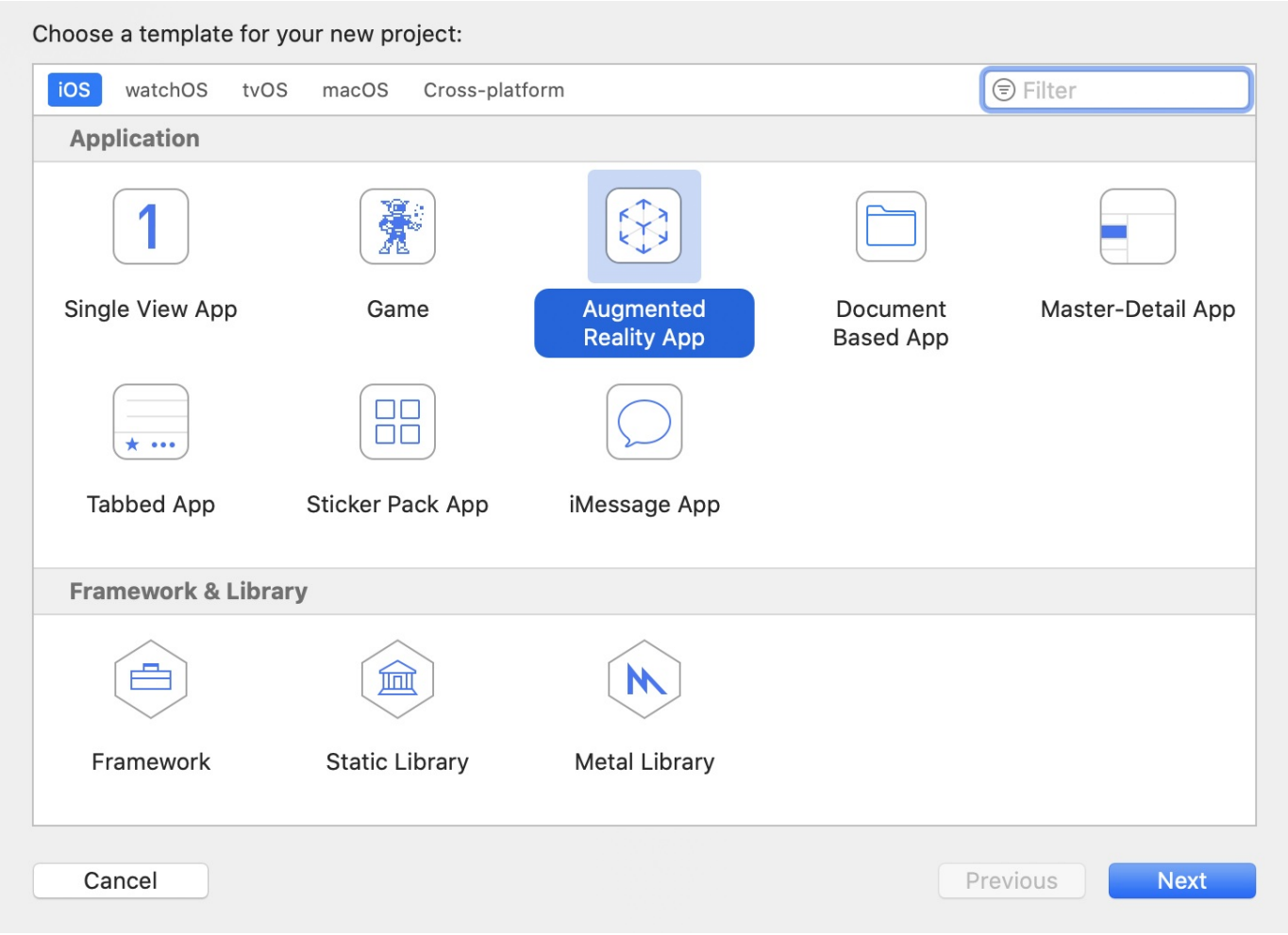


Figure 45.1. Choosing the Augmented Reality template

Once you saved the project, Xcode will generate the ARKit project for you. There are a couple of things we don't need in the generated code. By default, the template generates a 3D spaceship, which is loaded when the app launches. Obviously, we do not need this model. In the `ViewController.swift` file, change the following line of code in `viewDidLoad()` from:

```
let scene = SCNScene(named: "art.scnassets/ship.scn")!
```

To:

```
let scene = SCNScene()
```

For the `viewWillAppear` method, it's loaded with `ARWorldTrackingConfiguration()`:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    // Create a session configuration
    let configuration = ARWorldTrackingConfiguration()

    // Run the view's session
    sceneView.session.run(configuration)
}
```

We don't need it too because we will use another tracking configuration to handle the image tracking. Meanwhile, empty the method like this:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
}
```

Preparing the Resources

Now that we have created the project skeleton, let's move onto the next step and import the required resources. First, download the resource pack from https://www.appcoda.com/resources/swift5/ARNewspaperResources.zip. Since this app is going to play a video when a particular image is recognized, you have to prepare the following resources:

1. An image to be recognized. It can be any image (JPG/PNG) in your photo library. Or you can use the one (i.e. `snowboarding.jpg`) which comes with our resource pack.
2. A video to be played after the image is detected. Again, it can be any video in mp4 format. In the resource pack, you should find a video file named `snowboarding.mp4`.
3. A document containing the image. This is completely optional. But to mimic the Harry Potter newspaper, the resource pack includes a PDF file named `arkit-newspaper.pdf`, as shown in figure 45.2. You can print it out in A4 size paper for later testing. If you don't have a printer, this is okay. You can open the file on an iPad or a Mac. We will discuss more about how you can run the test in the later section.

Figure 45.2. The newspaper for testing

Next, we will import the video file (i.e. `snowboarding.mp4`) into Xcode. In the project navigator of Xcode, right click *ARNewspaper* and create a new group named `Resources`. Drag the video file from Finder into Xcode and place it in the `Resources` folder. When Xcode prompts you to choose options for adding the file, please make sure you tick the checkbox of `ARNewspaper` in the *Add to targets* option. If you've successfully imported the video file, you should be able to preview it in Xcode.

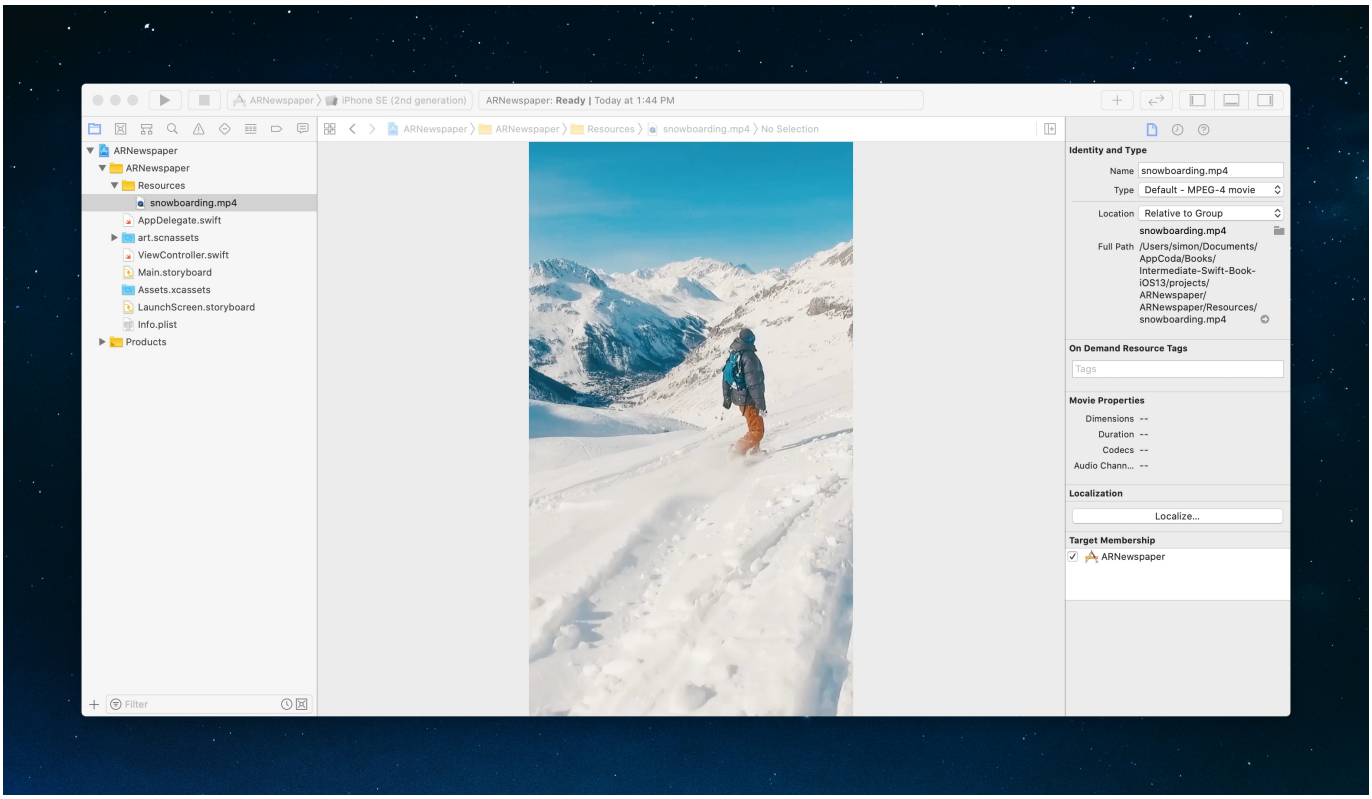


Figure 45.3. Importing the video file (snowboarding.mp4)

Setting the Reference Image for Image Tracking

We haven't imported the image yet. As you know, we used to add the images to the asset catalog. But this time, it needs a bit of extra work for image tracking. Open `Assets.xcassets` and right-click to choose *New AR Resources Group*. This creates a new resource group to specify the reference images for image tracking. You can drag the `snowboarding.jpg` file (or your own image) to the resource group.

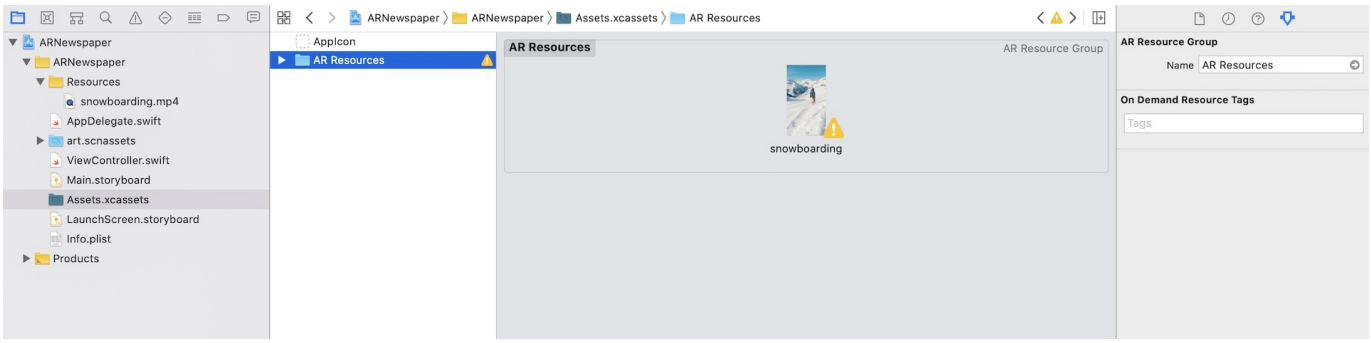


Figure 45.4. Adding the image to the AR Resource Group

While we only have one image for this demo, you can actually put multiple images for image tracking in this group. As you can see, there is a big exclamation mark on the image, which means there is something missing. Now select the image in the group and open the *Attributes inspector*.

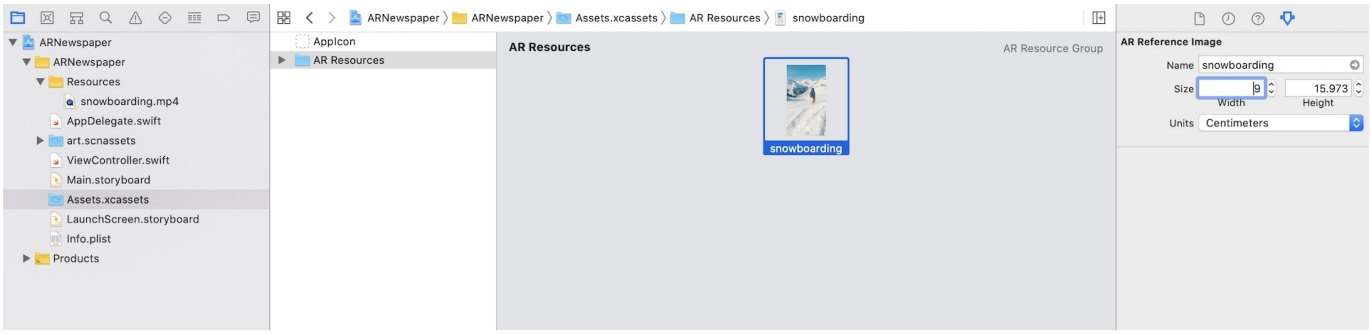


Figure 45.5. Setting the physical size

To detect the actual image, ARKit requires you to specify the real-world size of the image. Assuming you print the newspaper file on a A4-size paper, the image should have a width of 9cm and a height of 16cm. So, change the *Units* to `Centimeters` and set the width to `9`. Once you set the width, Xcode automatically calculates the *height* value.

Please note that the real-world size of the image varies. If you print it on a A3-size paper or display the PDF file on a 42-inch screen, the physical size of the image becomes larger. You will then need to adjust these values accordingly.

Setting up the AR Image Tracking Configuration

In the earlier chapters, we used `ARWorldTrackingConfiguration` to track the device's position and mix the virtual content with the real world. Other than the world tracking configuration, ARKit provides other types of configuration. To perform image tracking, we use a configuration called `ARImageTrackingConfiguration`. You use this configuration to detect and track the motion of known 2D images.

To set up this image tracking configuration, update the `viewWillAppear` method like this:

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)

    // Create a session configuration
    let configuration = ARImageTrackingConfiguration()

    // Define reference images
    guard let referenceImages = ARReferenceImage.referenceImages(inGroupNamed: "AR
 Resources", bundle: Bundle.main) else {
        fatalError("Failed to load the reference images")
    }

    // Specify the images to track
    configuration.trackingImages = referenceImages

    // Run the view's session
    sceneView.session.run(configuration)
}
```

> To accurately detect the position and orientation of a 2D image in the real world, ARKit requires preprocessed image data and knowledge of the image's real-world dimensions. The ARReferenceImage class encapsulates this information.
>
> - https://developer.apple.com/documentation/arkit/arreferenceimage

In the code, we specify to use `ARImageTrackingConfiguration` and tell ARKit what images to track. We call the `referenceImages` method of `ARReferenceImage` to load all referenced images from the asset catalog. The `inGroupNamed` parameter takes in the group name of

the AR resource group. Recalled that we named it `AR Resources`, this is why we pass it to the method.

Once we loaded the reference images, we assigned them to the `trackingImages` property of the configuration. Lastly, we call the `run` method to begin the image tracking operation using the back camera.

Overlaying a Video Player to the Detected Image

Now that we have created the image tracking configuration, the question is how can we overlay a video on top of the detected image?

When ARKit starts the image tracking, it automatically adds an `ARImageAnchor` object when the reference image is recognized. In a Scened-based ARKit app, the `renderer(_:didAdd:for:)` method of `ARSCNViewDelegate` will be called when the anchor is added to the scene. Thus, we can implement the method to overlay the video on top of the detected image.

First, let's declare the video play for the video playback in `ViewController` like this:

```
let videoPlayer: AVPlayer = {
    // Load the video
    guard let url = Bundle.main.url(forResource: "snowboarding", withExtension: "m
p4") else {
        return AVPlayer()
    }

    return AVPlayer(url: url)
}()
```

We load the `snowboarding.mp4` file and assign it to `AVPlayer` for playback. We just prepare the player for the video playback. The video won't start playing until we call its `play` method.

Now implement the method in the `ViewController` class like this:

```
func renderer(_ renderer: SCNSceneRenderer, didAdd node: SCNNode, for anchor: ARAn
chor) {

    guard let imageAnchor = anchor as? ARImageAnchor else {
        return
    }

    // Detected plane
    let plane = SCNPlane(width: imageAnchor.referenceImage.physicalSize.width,
                         height: imageAnchor.referenceImage.physicalSize.height)

    plane.firstMaterial?.diffuse.contents = self.videoPlayer

    let planeNode = SCNNode(geometry: plane)
    planeNode.eulerAngles.x = -.pi / 2

    node.addChildNode(planeNode)

    self.videoPlayer.play()
}
```

As said, every time a new anchor is added to the AR session. The method above will be called. Therefore, we first need to check if the newly added anchor is an `ARImageAnchor`. If it's an image anchor, we create a plane by using `SCNPlane` for holding the video player. The size of the rectangular plane is set to its real-world size that you provided earlier when configuring the reference image.

Next, we instantiate a `SCNNode` with the `plane` we just created and add the plane to the scene by calling `addChildNode`. Lastly, we call the `play()` method of the player to start the video playback.

You may wonder why we need to modify the `eulerAngles` to rotate the plane. By default, `SCNPlane` is vertically oriented. However, the detected image is supposed to be horizontal. To match the detected image's orientation, we have to rotate the plane. Figure 45.6 shows you the default orientation of the plane.

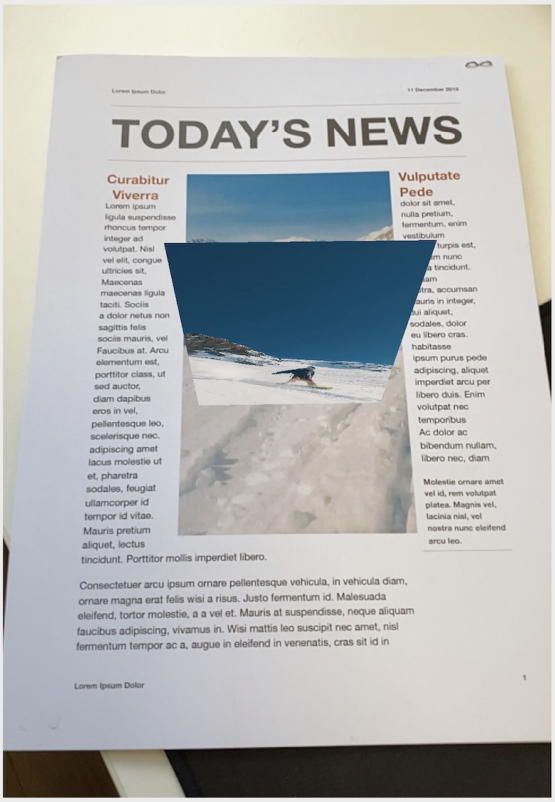


Figure 45.6. The default orientation of SCNPlane

If you can't wait to try out the image tracking, you can deploy the app to your iPhone. Assuming you've printed out the PDF document, you can point the device camera at the newspaper. When the app detects the "Snowboarding" image, it automatically overlay the video player on top of the image and starts the video playback.

In case that you do not have a printer, you may load the PDF document on iPad, which has a similar size of the A4 paper. Alternatively, you can open the file on your Mac for testing. But please try to resize it to A4 size.

Replaying the Video

The image tracking should work now. However, the video playback only occurs once. When the video finishes playing, it will not play again. To fix this issue, insert the following code in the `viewDidLoad` method:

```
NotificationCenter.default.addObserver(forName: .AVPlayerItemDidPlayToEndTime, obj
ect: self.videoPlayer.currentItem, queue: .main) { [weak self] (notification) in

    self?.videoPlayer.seek(to: CMTime.zero)
}
```

We added an observer to listen to the `AVPlayerItemDidPlayToEndTime` notification, which is posted when the video playback ends. In this case, we set the playback time back to zero.

To play the video again, insert the following method in `ViewController`:

```
func renderer(_ renderer: SCNSceneRenderer, didUpdate node: SCNNode, for anchor: A
RAnchor) {

    if let pointOfView = sceneView.pointOfView {
        let isVisible = sceneView.isNode(node, insideFrustumOf: pointOfView)

        if isVisible {
            if videoPlayer.rate == 0.0 {
                videoPlayer.play()
            }
        }
    }
}
```

This method is automatically called by ARKit when the anchor is updated. We first check the reference image is visible on screen and then start the replay if the result is true.

Disabling Statistics

You should notice a statistics bar appeared at the bottom of the screen. If you do not need it, you can disable it by changing the following line of code in `viewDidLoad` and set its value to `false`:

```
sceneView.showsStatistics = false
```

Expand the View to Full Screen

By default, the Scene View generated by Xcode has constraints that were configured respect to the safe area. This is why you see an empty white bar at the bottom of the screen. If you want to make the scene view full screen, you switch over to `Main.storyboard` and change the scene view's bottom constraint. In the Attributes inspector, change the *Second Item* to `Superview.Bottom` and set its constant value to `0`.

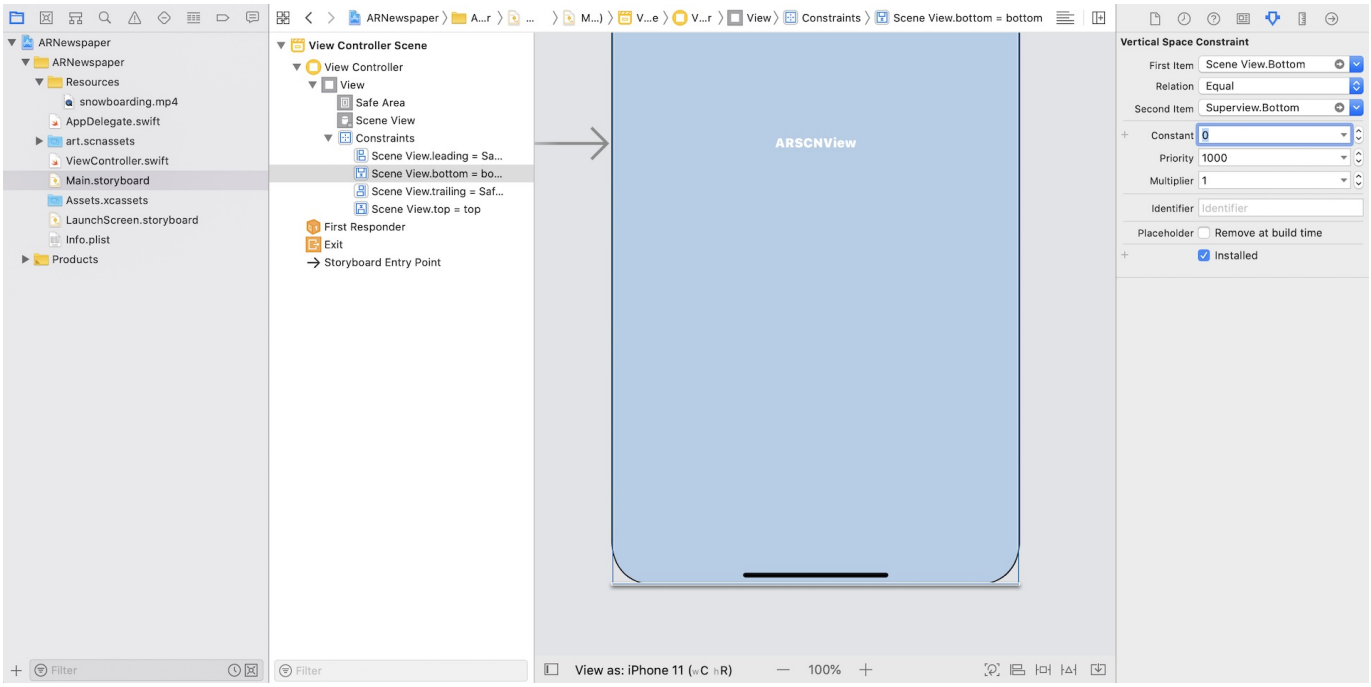


Figure 45.7. Changing the spacing constraint of the scene view

Summary

In this chapter, we explore another configuration provided by ARKit that lets you perform image tracking. With `ARImageTrackingConfiguration`, it empowers developers to build some interesting apps that merges virtual content with the real world environment. The Harry Potter newspaper is just one of the many examples. You may apply the technique to enrich the user experience and engagement of some real-world objects like postcard and restaurant menus.

For reference, you can download the final project from http://www.appcoda.com/resources/swift5/ARNewspaper.zip.