Donny Wals

Mastering iOS 11 Programming

Second Edition

Build professional-grade iOS applications with Swift 4 and Xcode 9

Packt

Contents

Chapter 1. UITableView Touch Up

Chances are that you have built a simple app before, or maybe you tried but didn't quite succeed. If you have, there's a good probability that you have used `UITableView`. The `UITableView` is a core component in many applications on the iOS platform. Virtually all applications that display a list of some sort make use of `UITableView`.

Because `UITableView` is such an important component in the world of iOS, I want you to dive in to it straightaway. You may or may not have looked at `UITableView` before, but that's OK. You'll be up to speed in no time and you'll learn how this component achieves that smooth 60 **frames per second** scrolling that users know and love. If your app can maintain a steady 60 fps, your app will feel more responsive and scrolling will feel perfectly smooth to users, which is exactly what you want.

In addition to covering the basics of `UITableView`, like how a `UITableView` uses a pattern called delegation to obtain information about the contents it should display, you'll learn how to make use of `Contacts.framework` to build an application that shows a list of your users' contacts. Much like the native Contacts app does on iOS.

`UITableView` makes use of cells to display its contents. In this chapter, you will create a cell with a custom layout to display the user's contacts. You will learn to do so using Auto Layout. Auto Layout is a technique that will be used throughout this book because it's an important part of every iOS developer's tool belt. If you haven't used Auto Layout before, that's OK. We will start with the basics in this chapter and we'll cover more complex uses of Auto Layout as we go.

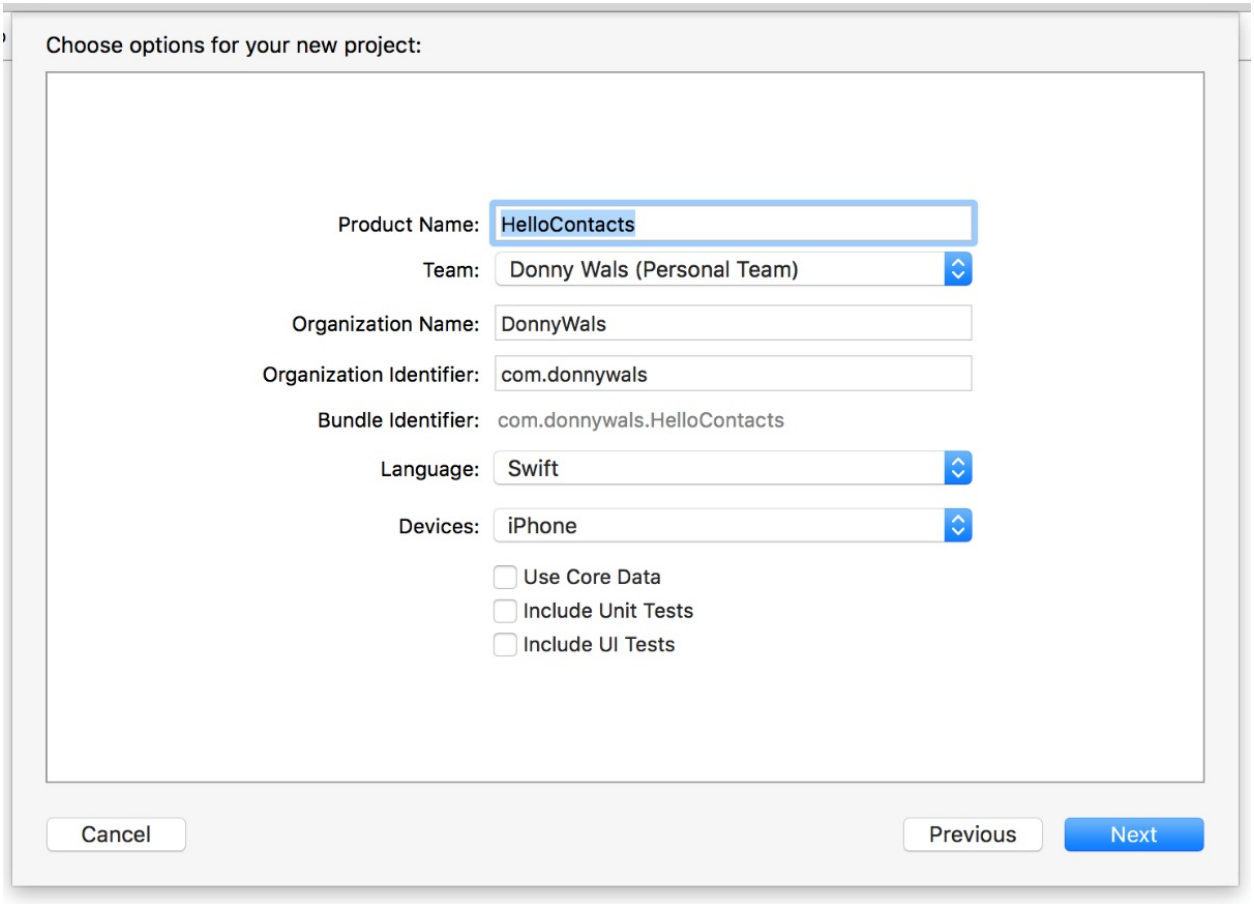
To sum it all up, this chapter covers:

- Configuring and displaying a `UITableView`
- Fetching a user's contacts through `Contacts.framework`
- `UITableView` delegate and data source
- Creating a custom `UITableViewCell`
- `UITableView` performance characteristics

Setting up the user interface (UI)

Every time you start a new project in Xcode, you must pick a template for your application. Each template that Xcode offers provides you with some boilerplate code or sometimes they will configure a very basic layout for you. Throughout this book, the starting point will always be the **`Single View Application`** template. This template provides you with a bare minimum of boilerplate code. This enables you to start from scratch every time, and it will boost your knowledge of how the provided templates work internally.

In this chapter, you'll create an app called *HelloContacts*. This is the app that will render your user's contacts in a `UITableView`. Create a new project by selecting **`File -> New -> Project`**. Select the **`Single View Application`** template, give your project a name (*HelloContacts*), and make sure you select **`Swift`** as the language for your project. You can uncheck all Core Data- and testing-related checkboxes; they aren't of any use to this project. Your configuration should resemble the following screenshot:



Once you have your app configured, open the `Main.storyboard` file. This is where you will...

Fetching a user's contacts

The introductory section of this chapter mentioned that you would use `Contacts.framework` to fetch a user's contacts list and display it in a `UITableView`.

Before we can display such a list, we must be sure we have access to the user's address book. Apple is very restrictive about the privacy of their users, if your app needs access to a user's contacts, camera, location, and more, you need to specify this in your application's `Info.plist` file. If you fail to specify the correct keys for the data your application uses, it will crash without warning. So, before attempting to load your user's contacts, you should take care of adding the correct key to `Info.plist`.

To add the key to access a user's contacts, open `Info.plist` from the list of files in the **`Project Navigator`** on the left and hover over **`Information Property List`** at the top of the file. A plus icon should appear, which will add an empty key with a search box when you click it. If you start typing Privacy – contacts, Xcode will filter options until there is just one option left. This option is the correct key for contact access. In the value column...

Creating a custom UITableViewCell for our contacts

To display contacts in your `UITableView`, you will must set up a few more things. First and foremost, you'll need to create a `UITableViewCell` that displays contact information. To do this, you'll create a custom cell by creating a subclass of `UITableViewCell`. The design for this cell will be created in Interface Builder. To make the views added to the cell in Interface Builder available to our code, `@IBOutlets` are added to the `UITableViewCell` subclass. These `@IBOutlets` are the connections between the visual layout in Interface Builder and your code.

Designing the contact cell

The first thing you need to do is drag a `UITableViewCell` out from the **`Object Library`** and drop it on top of the `UITableView`. This will add the cell as a prototype cell. The design you create in this cell will be replicated in all other cells that are added to the `UITableView` through code.

Next, drag out a `UILabel` and a `UIImageView` from the **`Object Library`** to the newly added `UITableViewCell`, and arrange them as they are arranged in the following screenshot. After you've done this, select both the `UILabel` and and...

Displaying the list of contacts

One easily overlooked fact about `UITableView` is that no matter how simple it might seem to use in your apps it's a pretty complex component. By not using a `UITableViewController`, we expose some of this complexity. You already had to manually set up Auto Layout constraints so the list always covers the full view. Then, you had to manually create and configure a prototype cell that's used to display data.

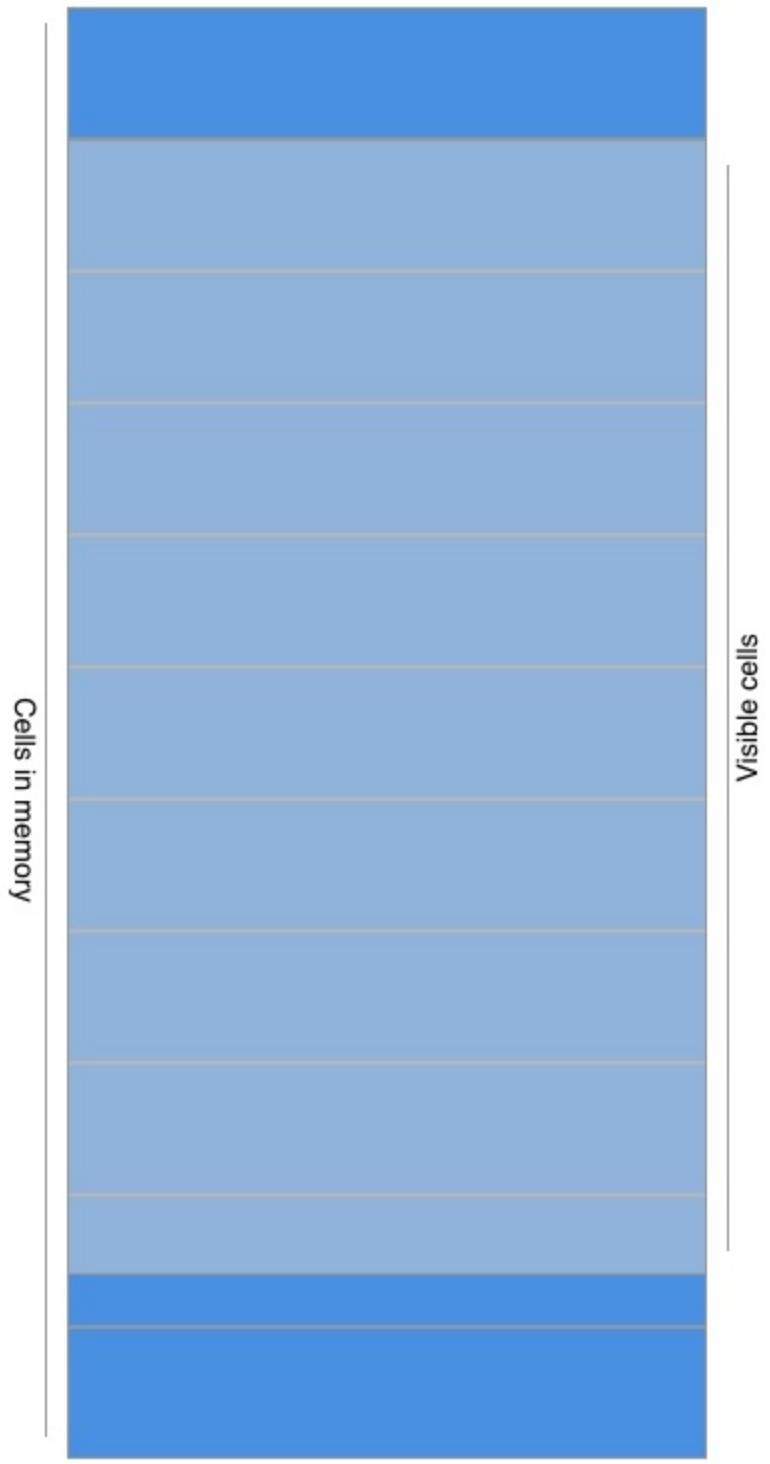
The next step in implementing the list of contacts is to provide the `UITableView` with information about the contents we want it to display. In order to do so, you'll implement the data source and delegate for your `UITableView`. These properties make use of some advanced concepts that you're likely to have seen before, but you have probably never been aware of what they are and how they work. Let's change that right now.

Protocols and delegation

Throughout the iOS SDK and the `Foundation` framework, a design pattern named *the delegation pattern* is used. The delegation pattern enables one object to perform actions on behalf of another object. When implemented correctly, this patterns allows you to separate...

Under the hood of UITableView performance

Earlier in this chapter you briefly read about the cell reuse in `UITableView`. You had to assign a reuse identifier to your `UITableViewCell` so the `UITableView` knows which cell you want to use. This is done so `UITableView` can reuse existing cells. This means that `UITableView` only needs to hold the visible cells and a few offscreen cells in memory instead of all of the cells you might want to show, even if the cell is off screen. Refer to the following figure for a visualization of what this looks like:



You can see that there are only a couple more cells in memory, or rendered, than there are visible cells. The `UITableView` does that so it can display huge amounts of data without rendering the content slower or losing its scroll performance. So no matter how many rows you have in your `UITableView`, it will not use more system resources than needed. This optimization was especially important in the initial days of iOS because the older iPhones were a lot more memory constrained than current devices are. Even though it's not as much of a necessity anymore, it's still one of the reasons users love...

UITableViewDelegate and interactions

Up until this point, the `ViewController` has conformed to `UITableViewDelegate` according to its declaration, but you haven't actually implemented any interesting delegate methods yet. Whenever certain interactions occur in `UITableView`, such as tapping a cell or swiping a cell, `UITableView` will attempt to notify its delegate about the action that has occurred. There are no required methods in the `UITableViewDelegate` protocol, which is why you could conform to it and act as a delegate without writing any implementation code. However, just implementing a list and doing nothing with it is kind of boring, so let's add some features that will make `UITableView` a little bit more interesting. If you look at the documentation for `UITableViewDelegate` you'll see that there's a large collection of methods you can implement in our app.

Note

You can hold the *Alt* key when clicking on a class, struct, enum, or protocol name to make an information dialog pop up. From this dialog pop up, you can navigate to the documentation for the definition you clicked.

In the documentation, you'll find methods for configuring...

Summary

The *HelloContacts* app is complete for now. The next few chapters will focus on improving it with a new layout, a detail page, and a couple more changes. You've covered a lot of ground on the way towards iOS mastery. You've used Auto Layout, the `Contacts` framework, you learned about delegation, custom table view cells, and you've implemented several delegate methods to implement several features on your table view.

If you want to learn more about `UITableView`, I don't blame you! The table view is a very powerful and versatile component in the iOS developer's tool belt. Make sure to explore Apple's documentation because there is a lot more to learn and study. One of the most important patterns you learned about is delegation. You'll find implementations of the delegate pattern throughout this book and `UIKit`. Next up? Converting the `UITableView`to its even more powerful and interesting sibling, `UICollectionView`.

Chapter 2. A Better Layout with UICollectionView

When Apple released iOS 6, they added a new component to UIKit: `UICollectionView`. `UICollectionView` is very similar to `UITableView` in terms of the APIs that are available and the way it handles delegation to other objects. The main difference between the two is that `UICollectionView` is a lot more powerful and flexible. It provides an easy to use grid layout out of the box. However, you're free to create any type of layout that you desire with a `UICollectionView`. You could even create a list that looks like a `UITableView` but has all of the flexibility that `UICollectionView` provides.

In this chapter, you'll build upon the *HelloContacts* app that you built in the previous chapter, Chapter 1, *UITableView Touch Up*. First, all `UITableView` code should be replaced with `UICollectionView` code. This will enable you to create a more interesting grid layout to display contacts in.

You'll also create a good-looking custom cell that shows a contact's image and name. To make your layout really stand out, we need to explore the `UICollectionViewFlowLayout`, which will enable us to implement a pretty...

Converting from a UITableView to UICollectionView

Displaying contacts in a list with `UITableView` is a fine idea. It's functional, looks alright, and people are used to seeing data displayed in a list. However, wouldn't it be nice if you had a more interesting way to display contacts, with bigger images maybe? Alternatively, maybe it would be nice to display contacts in a custom-designed grid?

Interesting and compelling layouts make your users happy. They will notice that you have put some extra effort in your layout to please them. Users enjoy apps that have received some extra attention. With that said, using a grid layout is no silver bullet. When implemented appropriately, it will delight your users. However, different apps and content types require different layouts so make sure that you always pick the right tool for the job.

For the use case of displaying contacts, a grid is a good choice. The goal is to show the user's contacts in an interesting way. We're not really interested in sorting them alphabetically. If we were, we would have used a list; a list is way better at showing a sorted list.

To display contacts in a grid,...

Creating and implementing a custom UICollectionViewCell

When you implemented the `UITableViewCell` in the previous chapter, you designed a custom cell. This cell was a view that was reused by the `UITableView` for every contact in the contacts array. `UICollectionView` also uses cells but you can not use `UITableViewCell` in a `UICollectionView`. However, the two different cells do share a lot of functionalities, such as the `prepareForReuse` method and the `awakeFromNib` method, we saw in the previous chapter, Chapter 1, *UITableView Touchup*.

When you replaced the table view with a collection view, you might have noticed that the collection view immediately contained a default cell. This cell is a lot more flexible than the table view cell was; you can resize both its width and its height while you could not manually resize the table view cell at all.

If you look at the **`Document Outline`** on the left-hand side, you can see an object called **`Collection View`** flow layout. This object is responsible for the layout of `UICollectionView`, and we'll have an in-depth look at it soon. For now, select it and go to the **`Size Inspector`** in the right panel and set...

Understanding the UICollectionViewFlowLayout and its delegate

In its simplest form, a `UICollectionView` has a grid layout. In this layout, all items are evenly spaced and sized. You can easily see this if you open the storyboard for the *HelloContacts* app. Select the prototype cell in the collection view, give it a background color, and then run the app. Doing this makes the grid layout very visible; it also shows how the constraints you set up earlier nicely center the cell's contents.

The ease of use and performance of `UICollectionView` make implementing grid layouts a breeze. However, the current implementation of the grid is not perfect yet. The grid looks alright on an iPhone 6s but on an iPhone SE, the layout looks like it's falling apart and it doesn't look much better when viewed on an iPhone 6s Plus. Let's see if we can fix this by making the layout a bit more dynamic.

In the storyboard, select the **`Collection View Flow Layout`** in the **`Document Outline`**. In the Attributes Inspector, you can change the scroll direction for a `UICollectionView`. This is something that a `UITableView` couldn't do; it only scrolls vertically. If you don't...

Creating a custom UICollectionViewLayout

Implementing something as big and complex as a custom `UICollectionViewLayout` looks like quite a challenge for most people. Creating a custom layout involves calculating the position for each and every cell that your collection view will display. You will have to make sure that your code does this as fast and efficiently as possible because your layout code will directly influence the performance of the entire collection view. Luckily, the documentation for implementing a custom layout is pretty good.

If you look at the documentation for `UICollectionViewLayout`, you can read about its role in a `UICollectionView`. This information shows that a custom layout requires you to handle layout for cells, supplementary views, and decoration views. Supplementary views are header and footer views. The *HelloContacts* app doesn't use these views so we can skip those for now. Decoration views are views that aren't related to the `UICollectionView` data, but are part of the view hierarchy. The only purpose of these views is decoration, as the name already suggests. The *HelloContacts* app doesn't use these either...

UICollectionView performance

We've already established how similar `UITableView` is to `UICollectionView` in terms of how you implement each component. In terms of performance, the similarities just don't stop. `UICollectionView` is optimized to display cells on screen as fast as possible with as little memory usage as possible, just like `UITableView` is. For `UICollectionView`, these optimizations are even more important than they are for `UITableView` because `UICollectionView` typically displays a lot more cells at the same time than `UITableView` does.

The fact that `UICollectionView` can show a lot of cells at once makes it a little bit harder to manage its performance behind the scenes. Before iOS 10 came out, cell reuse was managed as depicted in the next screenshot. All of the cells on a single row are requested right before they need to be displayed. Because we're not scrolling one cell into view but multiple cells, the collection view must obtain multiple cells in the same time frame that a table view requests a single cell.

If you have a complex layout in your cells or if you have a somewhat slow operation in one of your cells, a...

User interactions with UICollectionView

In the previous chapter, you saw how a `UITableView` uses delegation to handle user interactions. If a user interacts with a cell, the delegate can handle that interaction by implementing a method that handles the action performed by the user. `UICollectionView` works exactly the same, except that some of the details may vary from their `UITableView` counterparts. A `UICollectionView` can't be reordered as easily, for example, and it doesn't support swipe gestures for deletion. Because of this, these actions don't have any corresponding delegate methods in `UICollectionViewDelegate`. Similar actions can be implemented regardless, and in this subsection, you'll see how you can do it.

The interactions you'll implement are the following:

- Cell selection
- Cell deletion
- Cell reordering

Cell selection is the easiest to implement; the collection view has a delegate method for this. Cell deletion and reordering are a little bit harder because you'll need to write some custom code for them to work. So, let's start with the easy one: cell selection.

Cell selection

Implementing cell selection for `UICollectionView` works...

Summary

In this chapter, you learned how to harness the powers of `UICollectionView` and `UICollectionViewLayout` in your application. You saw that the collection view layout has a very simple grid by default but that you can implement some delegate methods to improve its rendering to make a nice, tight grid. Next, you created a custom layout by implementing your own `UICollectionViewLayout` subclass. Finally, you learned about performance and you implemented several `UICollectionViewDelegate` methods to enable cell deletion and reordering.

The material covered in this chapter is fairly complex, and it implements some pretty advanced interaction mechanisms, such as custom layouts, gesture recognition, and animation to provide feedback to the user. These concepts are very important in the toolbox of any iOS master because they cover a broad range of features that apps use. The next chapter will cover another important concept in iOS: navigation. We'll look at how to navigate from one view to another using the storyboard in Interface Builder. We'll also dive deeper into **`Auto Layout`** and `UIStackView`.

Chapter 3. Creating a Contact Details Page

So far, your app is able to display an overview of contacts. While this is already quite an accomplishment, it does lack one essential aspect of building apps; navigation. When a user taps one of their contacts, they will usually expect to be able to see more information about the contact they've tapped. You'll learn how to set up navigation by using storyboards and segues. Then, you'll see how to pass data between view controllers, which will make setting up details pages for contacts a breeze.

Also, the *HelloContacts* app is currently built with just the iPhone in mind. In the real world, iOS runs on two devices: iPhone and iPad. This chapter will explain how to design for all screen sizes. You'll learn how to make use of **Auto Layout** and **Size classes**-two tools Apple offers to create adaptive layouts that work on virtually any device or screen type.

Layouts with many elements that are positioned in relation to each other can be quite complex and tedious to set up with Auto Layout. In this chapter, you'll use `UIStackView` to easily create a layout with many elements in it. Lastly, we'll...

Universal applications

It is not uncommon for people to own more than a single iOS device. People who own both an iPhone and iPad tend to expect their favorite apps to work on both of their devices. Ever since Apple launched the iPad, it has enabled and encouraged developers to create universal apps. These are apps that run on both the iPhone and the iPad.

Over time, more screen sizes were added, and the available tools to create a universal, adaptive layout got a lot better. Nowadays, we use a single storyboard file in which a layout can be previewed on any screen size currently supported by Apple. You can even reuse most of your layouts and just apply some tweaks based on the available screen's real estate instead of having to design both versions of your app from the ground up.

If your apps are not adaptive at all, you're giving your users a subpar experience in a lot of cases. As mentioned before, your users expect their favorite apps to be available on each iOS device they own, and Apple actually helps your users if you don't provide a universal app. According to Apple, an iPhone app should be capable of running on the iPad at...

Implementing navigation with segues

Most good applications have more than a single screen. I bet that most app ideas you have in your head involve at least a couple of different screens. Maybe you'd display a `UITableView` or `UICollectionView` and a details view, or maybe users will drill down into your app's contents in some other way. Maybe you don't have any real details views but instead you would like to show some modal windows.

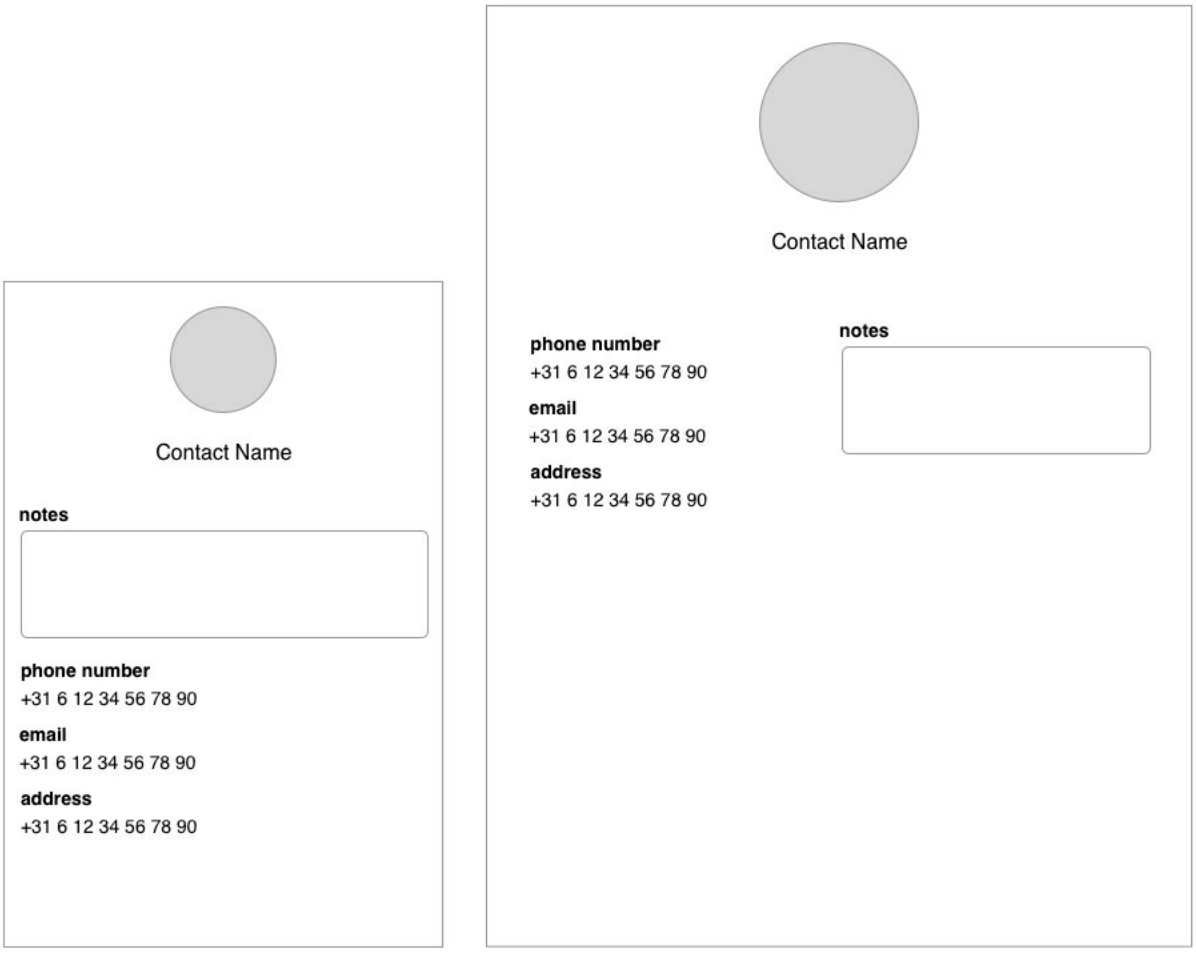
Every time your user moves from one screen to another, they're navigating. Navigation is a very important aspect of building an app, and it's absolutely essential that you understand the possibilities and patterns for navigation on the iOS platform. The easiest way to get started with navigation and to explore the way it works is to experiment inside of your `Main.storyboard` file.

Up until now, we've used the storyboard to create the layout for just a single screen. However, as the name implies, your storyboard isn't there for just a single screen. The purpose of the storyboard is to lay out and design the entire flow of your app. Every screen and transition can be designed right inside your storyboard...

Creating adaptive layouts with Auto Layout

At the beginning of the chapter, you learned what adaptive layouts are and why they are important. You learned a little bit about Size classes and `traitCollection`. In this section, we'll take a deep dive into these topics by implementing an adaptive contact details page. You'll learn some best practices to implement a layout that is tweaked for different Size classes. Finally, you'll learn how to make use of Auto Layout and Size classes in code because it is not always possible to define your entire layout in a storyboard. The layout you are going to build is shown as follows:

>



The **`Contact details`** page contains some of the contact's information, and the user can add notes to a contact. On small screens, the layout is just a single column: the notes fields is at the top, and the rest follows. On larger screens, the layout will be split across two columns to make good use of the available space. Let's see how we can implement both layouts using a **`Storyboard`**.

Auto Layout with Interface Builder

A good strategy to implement a layout like this is to add constraints that apply to all versions of...

Easier layouts with UIStackView

With iOS 9, Apple added an iOS version of macOS's `NSStackView` to `UIKit`. It's called `UIStackView`. This was a huge improvement because this view makes stacked layouts, such as the stack of labels and values the contact details page contains, a whole lot easier to create and maintain. The most common use case for a `UIStackView` is when you have a couple of views that are laid out relative to each other with equal spacing in between the items.

The layout we created earlier for the contact details page can definitely benefit from using `UIStackView`. The list of labels that displays a user's details in a list can be converted to a `UIStackView`. Currently, every label has a constraint to the label beneath it. This means that adding or removing a label right in the middle would involve removing constraints, reordering labels, and then adding new constraints.

With `UIStackView`, you can simply drag in all the labels you want displayed, configure the `UIStackView` class's layout, and reorder the labels by dragging them around. Constraints will be added and maintained by `UIStackView` automatically. You just have to...

Passing data between view controllers

The final bridge to cross for the *HelloContacts* app is to display some actual information about a selected contact. In order to do this, we'll need to add some new outlets to the `ContactDetailViewController`. The data that's loaded for contacts also needs to be expanded a little bit so a contact's phone number, email address, and postal address are fetched. Finally, the contact data needs to be passed from the overview to the details page so the detail page is able to actually display the data. The steps we'll take are as follows:

1. Update the data loading and model.
2. Pass the model to the details page.
3. Implement new outlets and display data.

Updating the data loading and model

Currently, the code in `ViewController.swift` specifies that just the given name, family name, image data, and image availability should be fetched. We need to expand this so the email address, postal address, and phone number are fetched as well. Update the `retrieveContacts(store:)` method with the following code:

```
let keysToFetch = [CNContactGivenNameKey as CNKeyDescriptor,
                   CNContactFamilyNameKey as...
```

Previewing content using 3D Touch

One of iOS's lesser-used features is 3D Touch. 3D Touch allows users to make their intent clear by either tapping the screen or by pressing it a little more firmly. The iPhone 6s and newer devices have implemented this functionality and it allows for some pretty neat interactions. One of these is called peek and pop. With peek and pop, a user can Force Touch an element on the screen and they'll see a preview of the detail page they'd see if they had performed a regular tap on the UI element. The following screenshot shows an example of such a preview:

If the user presses a little bit more firmly than when they saw the preview appear, they commit to the preview, meaning they will navigate to the detail page. To implement this, we only need to perform a small amount of work. First of all, the overview view controller must register its desire to provide preview capabilities for its collection view. Second, the overview view controller should implement the `UIViewControllerPreviewingDelegate` protocol so it can provide the preview with the correct view controller and commit to the preview if...

Summary

Congratulations, you have successfully created an application that runs on all iOS devices and screen sizes. You took an application that had just a single page that worked on an iPhone and turned it into a simple contacts application that works on any device. To achieve this, you made use of Auto Layout and `traitCollections`. You learned about Size classes and what they tell you about the available screen's real estate for your application. You also learned how to make use of Auto Layout through code and how to respond to changes in your app's environment in real time. Finally, you learned how to simplify a lot of the Auto Layout work you've done by implementing `UIStackView`. To top it all off, you saw how `prepare(for:sender)` allows you to pass data from an overview page to a details page.

The lessons you've learned in this chapter are extremely valuable. The increase in possible screen sizes over the past few years has made Auto Layout an invaluable tool for developers, and not using it will make your job much harder. If Auto Layout is still a bit hard for you, or it doesn't fully make sense, don't worry. We'll keep using...

Chapter 4. Immersing Your Users with Animation

The *HelloContacts* app is shaping up quite nicely already. We've covered a lot of ground by implementing a custom overview page and a contact detail page that work great on any screen size that exists today. However, there is one more aspect that we can improve to make our app stand out; animation. You have already implemented some animations but we haven't covered how they work yet.

The `UIKit` framework provides some very powerful APIs that you can utilize in your apps to make them look and feel great and natural. Most of these APIs are not even very difficult to use; you can create cool animations without bending over backward.

In this chapter you will learn about `UIViewPropertyAnimator`, a powerful object that was introduced in iOS 10 and can be used to replace the existing animations in the *HelloContacts* app. Next, you'll learn about UIKit dynamics. UIKit dynamics can be used to make objects react to their surroundings by applying physics. Finally, you'll learn how to implement a custom transition from one view to the next.

To sum everything up, this chapter covers the following...

Refactoring existing animations with UIViewPropertyAnimator

So far, you have seen animations that were implemented using the `UIView.animate` method. These animations are quite simple to implement and mostly follow the following format:

```
UIView.animate(withDuration: 1.5, animations:
{
    myView.backgroundColor = UIColor.red()
})
```

You have also seen this method implemented in other forms, including one that used a closure that was executed upon completion of the animation. For instance, when a user taps on one of the contacts in the *HelloContacts* app, the following code is used to animate a bounce effect:

```
UIView.animate(withDuration: 0.1, delay: 0, options: [.curveEaseOut], animations:
{
    cell.contactImage.transform = CGAffineTransform(scaleX: 0.9, y:
    0.9)
}, completion: { finished in
    UIView.animate(withDuration: 0.1, delay: 0, options:
    [.curveEaseIn], animations:
{
        cell.contactImage.transform = CGAffineTransform.identity
    }, completion: { [weak self] finished in
        self?.performSegue(withIdentifier: "detailViewSegue", sender: self)
    })
})
```

When you first saw the code used to implement this...

Understanding and controlling animation progress

One of `UIViewPropertyAnimator`'s great features is that you can use it to create interactive and reversible animations. Many of the animations you see in iOS are interactive animations. For instance, swiping on a page to go back to the previous page is an interactive transition. Swiping between pages on the home screen, opening the control center, or pulling down the notification center are all examples of animations that you manipulate by interacting with them.

While the concept of interactive animations might seem complex, `UIViewPropertyAnimator` makes it quite simple to implement them. As an example, we'll implement a drawer on the contact detail page in the *HelloContacts* app. First, we'll set up the view so the drawer is partially visible in the app. Once the view is all set up, we'll implement the code to perform an interactive show-and-hide animation for the drawer.

Open **`Main.storyboard`** and add a plain view to the contact detail view controller's view. Make sure that you do not add the drawer view to the scroll view. It should be added on top of the scroll view. Set up Auto...

Adding vibrancy to animations

A lot of animations on iOS look bouncy and realistic. For instance, when an object starts moving in the real world, it rarely does so smoothly. Often, something moves because somebody applied an initial force to it, causing it to have an initial momentum. Spring animations help you to apply this real-world dynamism to your animations.

Spring animations usually take an initial speed. This speed is the momentum the object should have when it begins moving. All spring animations require a damping to be set on them. The naming specifies how much an object can overflow its target value. A larger damping will make your animation feel more bouncy.

The easiest way to explore spring animations is by slightly refactoring the animation you just created for the drawer. Instead of using an `easeOut` animation when a user taps the **`toggle button, let's use a spring animation instead. The following code shows the changes you need to make to setUpAnimation():`**

```
func setUpAnimation() {
    guard animator == nil || animator?.isRunning == false
        else { return }
    let spring: UISpringTimingParameters
    if...
```

Adding dynamism with UIKit Dynamics

Most apps implement simple animations like the ones we have already seen. However, some animations could benefit from a slightly more dynamic approach. This is what UIKit Dynamics are for. With UIKit Dynamics, you can place one or more view in a scene that emulates physics. For instance, you can apply gravity to a certain object, causing it to fall downward on the screen. You can even have objects bumping in to each other; if you assign a mass to your views, this mass is actually taken into account when two objects bump into each other. Or, for instance, when you apply a certain force to an object with very little mass, it will be displaced more than an object with a lot of mass, just like you would expect in the real world.

To see how this all works, let's take a little break from building our contacts app and let's see if we can build a nice physics experiment! Create a new project and name it `CradleExperiment`. Make sure you configure it so only the landscape orientations are supported. In the `Main.Storyboard` file, make sure to set the preview to landscape and add three square views to the...

Customizing view controller transitions

Implementing a custom view controller transition is one of those things that can take a little while to get used to. There are several parts involved that are not always easy to make sense of.

However, once you get the hang of how it all ties together and you are able to implement your own transitions, you have a very powerful tool at your disposal. Proper custom view controllers can entertain and amaze your users. Making your transitions interactive could even ensure that your users will spend some extra time playing around with your app, which is exactly what you want. We'll implement a custom transition for the *HelloContacts* app. First, you'll learn how you can implement a custom modal transition. Once you've implemented that, we'll also explore custom transitions for `UINavigationController`, so we can show and hide the contact details page with a custom transition. We'll make the dismissal of both the modal view controller and the contact detail page interactive, so users can swipe to go back to where they came from.

To reiterate, the following are the steps we will go through:

1. Implement a...

Summary

In this chapter, you've learned a lot about animation. You are now aware that you can easily animate a view's property using the powerful `UIViewPropertyAnimator` object. You learned what timing functions are and how they affect animations. Also, more importantly, you saw how to make use of springs to make your animations look more lifelike. After learning animation basics, you also learned how to add custom animations to view controller transitions. The complexity for this does not necessarily lie in the animations themselves. The web of collaborating objects makes it quite hard to grasp custom view controller transitions at first. Once you have implemented this a few times, it will get easier to make sense of all the moving parts, and you'll see how all the building blocks fall right into place.

Despite the amount of information already present in this chapter, it does not cover every single aspect of animation. Just the most important parts of animations are covered, so you can utilize them to build better, more engaging apps. If you want to learn more about animation, you should look for resources on Core Animation. This...

Chapter 5. Improving Your Code with Value Types

Now that you have a deeper understanding about layout and user interfaces, it's time to take a step back and take a look at the underlying code you have been writing to build the *HelloContacts* app. If you have prior experience with OOP, nothing we have done so far should have frightened you. You might not have seen protocols used in class declarations before, but apart from that, inheritance in Swift is pretty much the same as you might have seen before.

However, as you have been playing around with Swift and iOS development, you might have noticed two object types you have not seen before: structs and enums. These two object types are what we refer to as value types. What this means and why it matters is covered in this chapter.

You will learn what structs and enums are and how you can use them in your own applications. You will also learn how these object types compare to traditional classes and how to make a choice between using a class, an enum or a struct. First, we'll look into classes and how they manifest themselves throughout apps a bit more, so you have a better...

Understanding reference types

In the previous chapters, you've been creating classes to contain your app's logic and user interface. This works perfectly fine, and to make use of these classes, we didn't need to know anything about how classes behave on the inside and how they manifest themselves in terms of memory and mutability. To understand value types and how they compare to reference types, it's essential that you do have an understanding of what's going on under the hood.

In Swift, there are two types of object that are considered a reference type: classes and closures. Classes are the only objects in Swift that can inherit from other classes. More on this later, when we discuss structs. Let's examine what it means for an object to be a reference type first.

Whenever you create an instance of a class, you have an object that you can use and pass around. For example, you could use an instance of a class as an argument of a method. This is demonstrated in the following snippet:

```
class Pet {
    var name: String

    init(name: String) {
        self.name = name
    }
}

func printName(forPet pet: Pet) {
   ...
```

Understanding value types

You just saw how passing around a reference type can yield results you might not expect. The idea behind a value type is that this doesn't happen, because instead of passing around references to be addresses in memory, you're passing around actual values. Doing this will often lead to safer and more predictable code. As if this isn't enough of a benefit on its own, value types are also cheaper to instantiate than reference types. More on that later. We'll focus on the most visible differences first.

Differences between values and references in usage

Let's take a look at the `Pet` example again; this time we'll use a `struct` instead of a `class`:

```
struct Pet {
    var name: String

    init(name: String) {
        self.name = name
    }
}

func printName(forPet pet: Pet) {
    print(pet.name)
}

let cat = Pet(name: "Bubbles")

printName(forPet: cat) // Bubbles
```

The initial example for our value-type exploration looks nearly identical to the reference-type version. The only notable difference is that we're now using a `struct` instead of a `class` in the definition of `Pet`. If we attempt to make the same change...

Using structs to improve your code

You already saw how value types can benefit your code by being more predictable. What you didn't see is one of the struct's most convenient features: default initializers. You saw that when we declared the `struct Pet`, we provided our own `init` method. For classes, this is required; for structs, it's not. If you create a `struct` without an initializer, it will generate its own. The default initializer for a `struct` is generated based on its members. Let's see what this looks like for the `struct Pet`:

```
struct Pet {
    let name: String
}

let cat = Pet(name: "Bubbles")
```

The default initializer that is generated takes all of the `struct`'s properties that don't have a default value. This is a very convenient way to create your structs. When you're challenged with the task of deciding whether you should use a struct or an enum, there are three basic rules of thumb that you can follow to determine whether you should at least try to use a struct instead of a class.

Note

If you add a custom initializer to one of your structs, the default initializer is not generated anymore. You can keep the generated...

Containing information in enums

In Swift, there's a second value type that you can use in your apps. This type is called an enum. Enums are used to store a finite set of options, and they can be used as arguments to functions and methods, and also as properties on structs and classes. Enums are best used when you have property that's, for example, an `Int` or `String`, but it can't take all `String` or `Int` values. Alternatively, an enum can be used to describe a value for which there isn't a suitable type. In that case, the enum is the type.

Whenever you have a predetermined set of possible values, it's probably a wise idea for you to use an enum. Let's take a look at how you can define an `enum` in your own code:

```
enum MessageStatus {
    case sent, delivered, read, unsent
}
```

This `enum` defines a set of statuses a message in a chat app could have. Storing this data in an `enum` allows us to use it as a `status` property on a message struct. Using an `enum`, we know that the `status` can never have an invalid value. Let's take a look at a quick example of using an `enum` as a property on a `struct`:

```
struct Message {
    let contents: String
    let...
```

Summary

In this chapter, you have looked at value types and reference types. The difference between these two types might seem subtle at first. However, you saw that the memory implications make these types fundamentally different. You saw the most important distinction by examining how they behave. You saw how class instances are passed around by passing the address in memory. This means that mutating one instance mutates everything that points to it. Sometimes, it's not obvious that you're mutating more than one instance. Then, you saw how structs are passed around by value. This meant that all the instances essentially get copied when you pass them around. This ensures that you don't accidentally change values you don't expect to change. Finally, you saw how enums are used to contain a finite set of possible cases.

Picking the right option isn't always easy or straightforward, so if you try to stick to the rules of thumb provided, you should be able to make an informed decision. If you turn out to have picked the wrong option, you can always refactor your code to use a different type. It's recommended that you always try to...

Chapter 6. Making Your Code More Flexible with Protocols and Generics

If you've been around a Swift developer long enough, you must have heard them mention protocol-oriented programming at least once. This programming paradigm was introduced by Apple at *WWDC 2015* in a talk that generated a lot of buzz among developers. Suddenly, we learned that thinking in classes and hierarchies leads to code that's hard to maintain, change, and expand. The talk introduced a method of programming that is focused on what an object can do instead of explicitly caring about what an object is.

This chapter will demonstrate to you how you can make use of the powers of POP, and it will show you why it's an important feature of Swift. We'll start off with some simple use cases, and then we'll take a deep dive into its associated types and generic protocols.

Understanding these design patterns and recognizing situations in which a protocol, protocol extension, or a generic protocol can help you improve your code will lead to code that is not only easier to maintain but also a joy to work with. The structure for this chapter is as follows:

- Defining...

Defining your own protocols

Throughout `UIKit`, protocols are first-class citizens. You might have noticed this when you were implementing the custom `UIViewController` transitions. When you implemented these transitions, you had to create an object that functioned as a delegate for the transition and conformed it to the `UIViewControllerTransitioningDelegate` protocol. You also implemented an `NSObject` subclass that conformed to `UIViewControllerAnimatedTransitioning`.

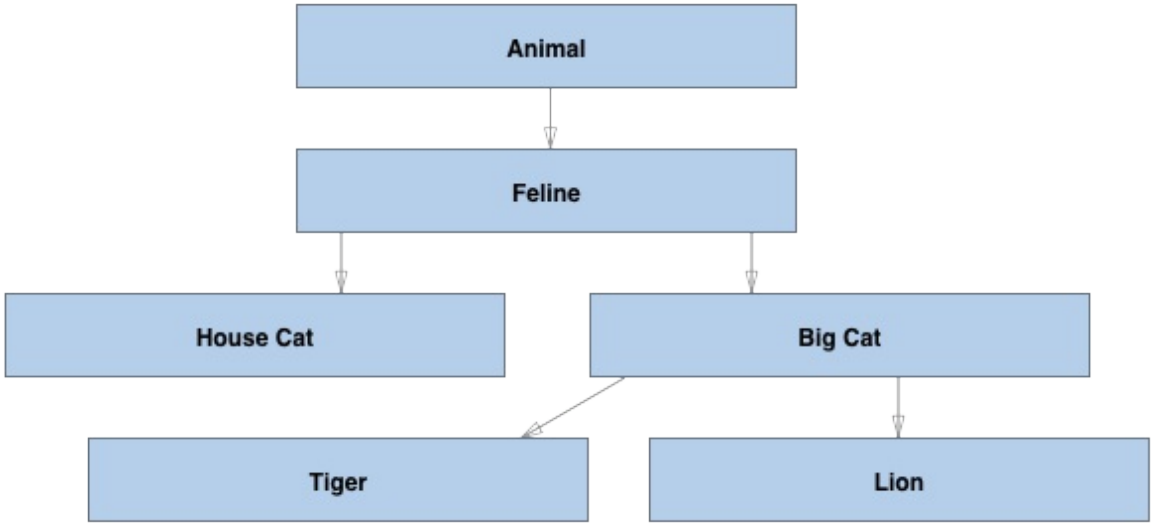
It's possible for you to define and use your own protocols. This usage is not confined to delegate behavior only. Defining a protocol is very similar to defining a class, struct, or enum. The main difference is that a protocol does not implement or store any values on its own. It acts as a contract between whoever calls an object that conforms to a protocol and the object that claims to conform to the protocol.

Create a new **`Playground`** (**`File`** | **`New...`** | **`Playground`**) if you want to follow along, or check out the Playground in the book's Git repository.

Let's implement a simple protocol of our own that establishes a baseline for any object that claims to be a pet. The protocol...

Checking for traits instead of types

In classic OOP, you often create superclasses and subclasses to group together objects with similar capabilities. If you roughly model a group of felines in the animal kingdom with classes, you end up with a diagram that looks like this:



If you tried to model more animals, you would find that it's a really complex task because some animals share a whole bunch of traits, although they are actually quite far apart from each other in the class diagram.

One example would be that both cats and dogs are typically kept as pets. This means that they should optionally have an owner and maybe a home. But cats and dogs aren't the only animals kept as pets because fish, guinea pigs, rabbits, and even snakes are kept as pets. However, it would be really hard to figure out a sensible way to restructure your class hierarchy in such a way that you don't have to redundantly add owners and homes to every pet in the hierarchy, because it would be impossible to selectively add these properties to the right classes.

This problem gets even worse when you write a function or method that prints a pet's home. You would...

Summary

This chapter is packed with complex and interesting information that is essential if you want to write beautiful Swift code. Protocols allow you to write extremely flexible code that doesn't rely on inheritance, which makes it a lot easier to understand and reason with your code. In this chapter, you saw how you can leverage the power of protocols to work with an object's traits or capabilities rather than just using its class as the only way of measuring its capabilities.

Next, you saw how protocols can be extended to implement a default functionality. This enables you to compose powerful types simply by adding protocol conformance, instead of creating a subclass. You also saw how protocol extensions behave depending on your protocol requirements, and that it's wise to have anything that's in the protocol extension defined as a requirement. This makes the protocol behavior more predictable. Finally, you learned how associated types work and how they can lift your protocols to the next level by adding generic types to your protocols, that can be tweaked for every type that conforms to your protocol.

The concepts shown in...

Chapter 7. Refactoring the HelloContacts Application

When we first built *HelloContacts*, we used classes and Object-Oriented Programming techniques. Now that you have seen how value types and protocols can improve your code, it's a good idea to revisit the *HelloContacts* application to see how we can improve it with this newfound knowledge. Even though the app is fairly small, there are a few places where we can improve it and make it more flexible and future-proof.

This chapter is all about making the *HelloContacts* application Swiftier than it is now. We'll do this by implementing elements of the app with protocols and value types. This chapter will cover the following topics:

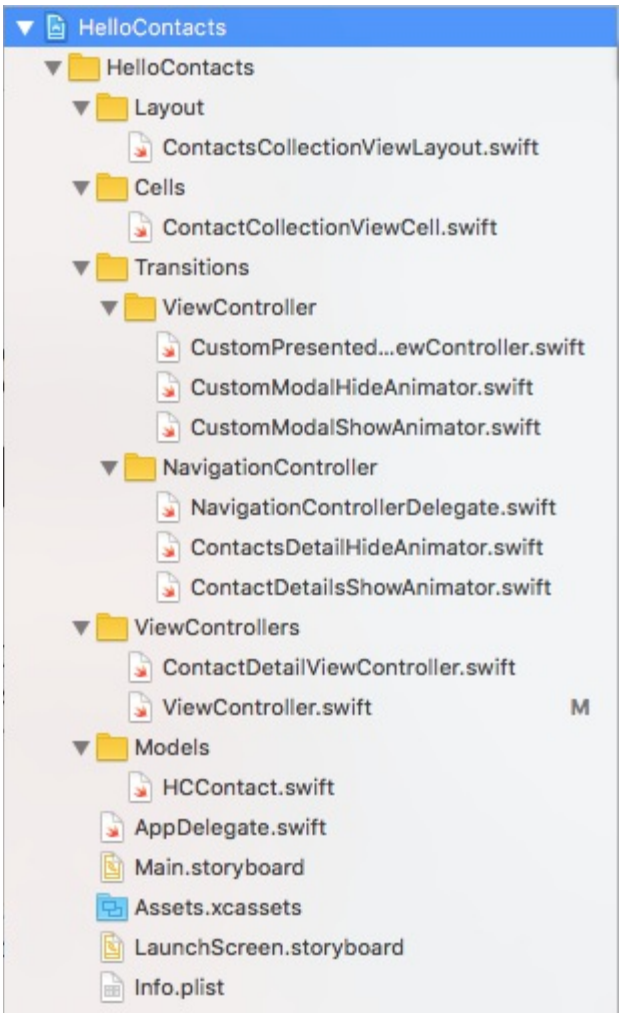
- Properly separating concerns
- Adding protocols for clarity

Let's get started right away.

Properly separating concerns

Before we can improve our project structure with value types and protocols, it's a good idea to improve upon our general structure. We haven't really thought about the reuse of certain aspects of the *HelloContacts* app, which results in code that's harder to maintain in the long run. If you take a look at the source code for this project in the book's code bundle, you' find that the project was slightly modified.

First, all the different files were put together in sensible groups. Doing so makes it easier for you to navigate your project's files, and it creates a natural place for certain files, as shown in the following screenshot:



The structure applied in this project is merely a suggestion; if you feel that a different structure will suit you better, go ahead and make the change. The most important part is that you've thought about your project structure and set it up so it makes sense to you and helps you navigate your project.

With this improved folder structure, you may notice that there's some sort of a divide between certain files. There are files that help with transitions, a model file, and...

Adding protocols for clarity

We've already seen how protocols can be used to improve code by removing complex inheritance hierarchies. You also know how powerful it is when it comes to checking for protocol conformance instead of checking whether a certain object is of a certain type. Let's see how we can improve and future-proof the *HelloContacts* application by adding some protocols.

We will define two protocols for now: one that specifies the requirements for any object that claims to be able to add a special animation to a view, and one that defines what it means to be able to be displayed as a contact.

Defining the ViewEffectAnimatorType protocol

The first protocol we will define is called `ViewEffectAnimatorType`. This protocol should be applied to any object that implements the required behaviors to animate a view. This protocol does not necessarily give us a direct advantage, but there are a few considerations that make this a very useful protocol.

A protocol is not only used to check whether or not an object can do something: It can also formalize a certain API that you came up with. In this case, we've decided that our

Summary

This chapter wraps up our exploration of protocol-oriented programming, value types, and reference types. In the previous two chapters, you saw some theoretical situations that explain the power of these features in Swift. This chapter tied it all together by applying your newfound knowledge to the *HelloContacts* app we were working on before. You now know how you can bump up the quality and future-proof an existing application by implementing protocols and switching to using a struct instead of a class. In order to implement protocols, we had to improve our application by making sure that our `ViewController` didn't contain too many functionalities. This in itself was a huge improvement that we were able to take to the next level with a protocol.

Refactoring from classes to structs isn't always easy; the two aren't interchangeable, as we saw when we changed `HCContact` from a class to a struct. The main difference we encountered was due to how structs handle mutability. Adapting our code for this wasn't too complex, and it made our intent clearer than it was when we used classes.

Now that we have explored some of the best...

Chapter 8. Adding Core Data to Your App

Core Data is Apple's data persistence framework. You can utilize this framework whenever your application needs to store data. Simple data can often be stored in `NSUserDefaults`, but when you're handling data that's more complex, has relationship, or needs some form of efficient searching, Core Data is much better suited to your needs.

You don't need to build a very complex app or have vast amounts of data to make Core Data worth your while. Regardless of your app's size, even if it's really small, with only a couple of records, or if you're holding on to thousands of records, Core Data has your back.

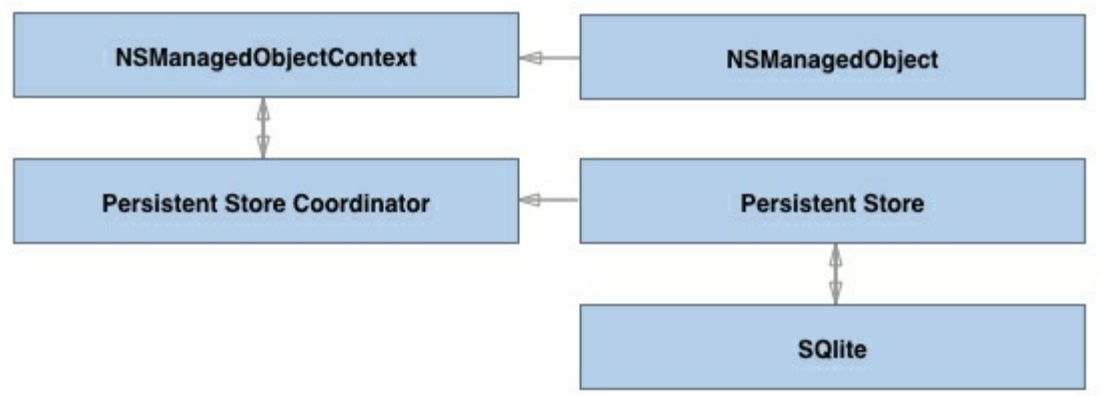
In this chapter, you'll learn how to add Core Data to an existing app. The app we will build keeps track of a list of favorite movies for all members of a family. The main interface will be a table view that shows a list of family members. If you tap on a family member, you'll see the tapped his/her favorite movies. Adding family members can be done through the overview screen, and adding movies can be done through the detail screen.

We won't build the screens in this app from scratch. In the...

Understanding the Core Data stack

Before we delve into the first project and add Core Data to it, we'll take a look at how Core Data actually works, what it is, and what it isn't. In order to make efficient use of Core Data, it's essential that you know what you're working with.

When you work with Core Data, you're actually utilizing a stack of layers that starts with managed objects and ends with a data store. This is often an SQLite database but there are different storage options you can use with Core Data, depending on your application needs. Let's take a quick look at the layers involved with Core Data and discuss their roles in an application briefly:



At the top right of this diagram is the `NSManagedObject` class. When you are working with Core Data, this is the object you'll interact with most often since it's the base class for all Core Data models your app contains. For instance, in the app we will build, the family member and movie models are subclasses of `NSManagedObject`.

Each managed object belongs to an `NSManagedObjectContext`. The managed object context is responsible for communicating with the persistent store...

Adding Core Data to an application

When you create a new project in Xcode, Xcode asks whether you want to add Core Data to your application. If you check this checkbox, Xcode will automatically generate some boilerplate code that sets up the Core Data stack. Prior to iOS 10, this boilerplate code spanned a couple of dozen lines because it had to load the data model, connect to the persistent store, and then it also had to set up the managed object context.

In iOS 10, Apple introduced `NSPersistentContainer`. When you initialize an `NSPersistentContainer`, all this boilerplate code is obsolete and the hard work is done by the `NSPersistentContainer`. This results in much less boilerplate code to obtain a managed object context for your application. Let's get started with setting up your Core Data stack. Open `AppDelegate.swift` and add the following `import` statement:

```
import CoreData
```

Next, add the following `lazy` variable to the implementation of `AppDelegate`:

```
private lazy var persistentContainer:
  NSPersistentContainer = {
    let container = NSPersistentContainer(name:
      "FamilyMovies")...
```

Summary

This chapter was all about getting your feet wet with Core Data. You learned what the Core Data stack looks like and which parts are involved in using Core Data in your application. Even though the `NSPersistentContainer` object abstracts away a great deal of all of the complexity involved in the Core Data stack, it's still good to have a rough understanding of the parts that make up the Core Data stack. After seeing this, you learned how to set up the Core Data stack and how you can use dependency injection to cleanly pass along the managed object context from the `AppDelegate` to the initial view controller.

Then, you saw how to define your Core Data models and how to set up relationships between them. Doing this is made relatively simple in the editor because we can add properties by clicking on a plus icon. You saw that there's a range of options available to you when you add a new property but for a small application, such as the one we're building, most options can be set to their default values. Finally, we took a look at generating our model classes. You saw three ways that Xcode allows us to create classes for our...

Chapter 9. Storing and Querying Data in Core Data

Now that you know what Core Data is, how it works, and how you can integrate it in your application, the time is right to figure out how to store and query data with it. Adding the Core Data stack wasn't very complex. However, handling actual data in your apps is a bit more complex since you might have to deal with things such as multithreading, or objects that suddenly get updated when you've just read them from the database. Dealing with databases isn't easy, especially if you're trying to ensure data integrity. Luckily for us, Core Data helps tremendously with that.

In this chapter, we will insert some pieces of data into our database and read them. You'll learn how to update the user interface whenever the underlying data changes. You will also learn how you can ensure that you don't accidentally try to read or write objects on the wrong thread and why it matters. To read and filter data, you will use predicates. Finally, you will learn about some of the ways iOS 10 helps in making sure that you always use the correct objects from your database. To wrap everything up, we'll...

Storing data with Core Data

The first step to implement data persistence for your app is to make sure that you can store data in the database. The models that we will store were already defined with Xcode's model editor in the previous chapter. First, we'll take a look at what's involved in storing data, and then we'll implement the code that persists our models to the database. Finally, we'll improve our code by refactoring it to be more reusable.

Understanding data persistence

Whenever you want to persist a model with Core Data, you must insert a new `NSManagedObject` into a `NSManagedObjectContext`. Doing this does not immediately persist the model you created. It merely stages your object in the current `NSManagedObjectContext`. If you don't properly manage your managed objects and contexts, this is a potential source of bugs. For example, not persisting your managed objects results in the loss of your data once you refresh the context. Even though this might sound obvious, it could lead to several hours of frustration if you don't carefully manage this.

If you want to properly save managed objects, you will need to tell the managed...

Reading data with a simple fetch request

The simplest way to fetch data from your database is to use a fetch request. The managed object context forward fetches requests to the persistent store coordinator. The persistent store coordinator will then forward the request to the persistent store, which will then convert the request to an SQLite query. Once the results are fetched, they are passed back up this chain and converted to `NSManagedObjects`. By default, these objects are called faults. When an object is a fault, it means that the actual properties and values for the object are not fetched yet, but they will be fetched once you access them. This is an example of a good implementation of lazy variables because fetching the values is a pretty fast operation, and fetching everything up front would greatly increase your app's memory footprint because all values must be loaded into memory right away.

Let's take a look at an example of a simple fetch request that retrieves all `FamilyMember` instances that were saved:

```
let request: NSFetchRequest<FamilyMember> = FamilyMember.fetchRequest()

guard let moc = managedObjectContext,
   ...
```

Filtering data with predicates

A common operation you'll want to perform on your database is filtering. In Core Data, you make use of predicates to do this. A predicate describes a set of rules that any object that gets fetched has to match.

When you are modeling your data in the model editor, it's wise to think about the types of filtering you need to do. For instance, you may be building a birthday calendar where you'll often sort or filter dates. If this is the case, you should make sure that you have a Core Data index for this property. You can enable this with the checkbox you saw earlier in the model editor. If you ask Core Data to index a property, it will significantly improve performance when filtering and selecting data.

Writing predicates can be confusing, especially if you try to think of them as the `where` clause from SQL. Predicates are very similar, but they're not quite the same. A simple predicate will look as follows:

```
NSPredicate(format: "name CONTAINS[n] %@", "Gu")
```

A predicate has a format; this format always starts with a key. This key represents the property you want to match on. In this example, it would be the...

Reacting to database changes

In its current state, our app doesn't update its interface when a new managed object is persisted. One proposed solution for this is to manually reload the table right after we insert a new family member. Although this might work well for some time, it's not the best solution to this problem. If our application grows, we might add a functionality that enables us to import new family members from the network. Manually refreshing the table view would be problematic because our networking logic should not be aware of the table view. Luckily, there is a better suited solution to react to changes in your data.

First, we'll implement a fetched results controller to update our list of family members. Next, we'll listen to notifications in order to update the list of a family member's favorite movies.

Implementing a NSFetchedResultsController

The `NSFetchedResultsController` class notifies a delegate whenever its fetched data is changed. This means that you won't have to worry about manually reloading the view and you can simply process whatever updates the fetched results controller passes on.

Being a delegate for...

Understanding the use of multiple NSManagedObjectContexts

It has been mentioned several times throughout the past two chapters that you can use multiple managed object contexts. In many cases, you will only need a single managed object context. Using a single managed object context means that all of the code related to the managed object context is executed on the main thread. If you're performing small operations, that's fine. However, imagine importing large amounts of data. An operation like that could take a while. Executing code that runs for a while on the main thread will cause the user interface to become unresponsive. This is not good, as the user will think your app has crashed. So how do you work around this? The answer is using multiple managed object contexts.

In the past, using several managed object contexts was not easy to manage, you had to create instances of `NSManagedObjectContext` using the correct queues yourself. Luckily, the `NSPersistentContainer` helps us manage more complex setups. If you want to import data on a background task, you can either obtain a managed object context by calling on the persistent...

Summary

This chapter focused on inserting and retrieving data with Core Data. You saw how you can make use of fetch requests, predicates, fetched result controllers, and notifications to fetch data. The examples started out simple, by inserting a new object by writing all the boilerplate code that's involved with this. Then you saw how you can refactor this kind of boilerplate code by using extensions; this makes your code more readable and maintainable. It's highly encouraged that you create your own extensions whenever you encounter the same kind of patterns as we did in this chapter. Once you saw how to insert data, you learned about fetch requests and predicates. Even though these two objects aren't very complex, they are very powerful and you'll often use them when you work with Core Data.

Next, you learned how to react to database changes with `NSFetchedResultsController` and `NotificationCenter`. A fetched results controller gives you a very powerful way to subscribe to changes for a set of results. Notifications are more of a general purpose solution, but our implementation makes sure that we are actually only observing a...

Chapter 10. Fetching and Displaying Data from the Network

Most modern applications communicate with a web service. Some apps rely on them heavily, acting as a layer that simply reads data from the web and displays it in app form. Other apps use the web to retrieve and sync data to make it locally available, and some others only use the web as backup storage. Of course, there are a lot more reasons to use data from the internet other than the ones mentioned.

In this chapter, we will expand the `FamilyMovies` application so that it uses a web service to retrieve popularity ratings for the movies our family members have added as their favorites. We'll store these popularity ratings in a Core Data database and display them together with the names of the movies.

In this chapter, you'll learn about the following topics:

- URLSession
- Working with JSON in Swift
- Updating Core Data objects with fetched data

Let's see how you can fetch data from the web first, shall we?

Fetching data from the web

Retrieving data from the web is something that you will do often as an iOS professional. You won't just fetch data from a web service; you'll also send data back to it. For example, you might make an HTTP `POST` request to log in a user or to update their information. Over time, iOS has evolved quite a bit in the web requests department, making it easier, simpler, and more consistent to use the web in apps, and we're currently in a pretty good spot with the `URLSession` class.

Note

HTTP (or HTTPS) is a protocol that almost all web traffic uses for communication between a client, like an app, and a server. The HTTP protocol supports several methods that signal the request's intent. `GET` is used to retrieve information from a server. A `POST` request signals the intent to push new content to a server. For instance, submitting a form.

The `URLSession` class makes asynchronous web requests on your behalf. This means that iOS loads data from the internet on a background thread, ensuring that the user interface remains responsive throughout the duration of the entire request. If the request is performed synchronously, the...

Working with JSON in Swift

The following snippet shows how you can convert raw data to a JSON dictionary. Working with JSON in Swift can be a little tedious at times, but overall, it's an all right experience. Let's look at an example:

```
guard let data = data,
    let json = try? JSONSerialization.jsonObject(with: data, options: [])
    else { return }

print(json)
```

The preceding snippet converts the raw data that is returned by a URL request to a JSON object. The print statement prints a readable version of the response data, but it's not quite ready to be used. Let's see how you gain access to the first available movie in the response.

If you look at the type of object returned by the `jsonObject(with:options:)` method, you'll see that it returns `Any`. This means that we need to typecast the returned object to something we can work with, such as an array or a dictionary. When you inspect the JSON response we received from the API, for instance by using print to make it appear in the console, like you did before with Google's homepage HTML, you'll notice that there's a dictionary that has a key called `results`. The `results` object is...

Updating Core Data objects with fetched data

So far, the only thing we have stored in Core Data is movie names. We will expand this functionality by performing a lookup for a certain movie name through the movie database API. We will use the fetched information to display and store a popularity rating for the movies in our database.

A task such as this seems straightforward at first; you could come up with a flow such as the one shown in the following steps:

1. The users fill out their favorite movie.
2. It fetches popularity.
3. It stores the movie and its popularity.
4. The interface updates with the new movie.

At first sight, this is a fine strategy; insert the data when you have it. However, it's important to consider that API calls are typically done asynchronously, so the interface stays responsive. Also, more importantly, API calls can be really slow if your user doesn't have a good internet connection. This means that you would be updating the interface with noticeable lag if the preceding steps are executed one by one.

The following would be a much better approach to implement the feature at hand:

1. The users fill out their favorite movie.
2. The...

Wrapping the features up

All of the code is in place, and now you understand multithreading and how callbacks can be used in a multithread environment. You learned about the defer statement and how it can be used to execute a block of code at the end of a scope. Yet, if you build and run your app and add a new movie, the rating won't be displayed yet.

The following are the three reasons why this is happening:

- We aren't setting the movie's rating on the table view cell
- The network request doesn't succeed because of App Transport Security
- We're not observing the update

We'll solve these issues in order, starting with the table view cell.

Adding the rating to the movie cell

Currently, the movie table view displays cells that just have a title. `UITableViewCell` has a built-in option to display a title and a subtitle for a cell. Open the `Main.storyboard` and select the prototype cell for the movies. In the `Attributes Inspector` field, change the cell's style from basic to subtitle. This will enable us to use the `detailTextLabel` on our table view cell. This is where we'll display the movie rating.

In `MoviesViewController`, add the following line...

Summary

This chapter was all about adding a small, simple feature to an existing app. We added the ability to load real data from an API. You saw that networking is made pretty straightforward by Apple with `URLSession` and data tasks. You also learned that this class abstracts away some very complex behavior regarding multithreading, so your apps remain responsive while data is loaded from the network. Next, you implemented a helper struct for networking and updated the Core Data model to store ratings for movies. Once all this was done, you could finally see how multithreading worked in the context of this app. This wasn't everything we needed to do, though. You learned about ATS and how it keeps your users secure. You also learned that you sometimes need to circumvent ATS, and we covered how you can achieve this.

Even though the feature itself wasn't very complex or big, the concepts and theory involved can be quite overwhelming. You suddenly had to deal with code that would be executed asynchronously, in the future. And not just that. The code even used multiple threads to make sure that its performance was optimal. The concepts...

Chapter 11. Being Proactive with Background Fetch

So far, the `FamilyMovies` application is shaping up to be quite nice. You can add family members and associate movies with them, and these movies are automatically enriched with the first rating we can find from the movie database API. There are still some movies we added earlier that don't have a rating, and although the ratings for the existing movies might be correct for now, we don't know whether they will still be accurate after a couple of days, weeks, or months. We could update the ratings whenever the user accesses them, but it's not very efficient to do this. We would potentially reload the same movie a couple of times in a single session.

Ideally, we'd update the movies in the background, when the user isn't using the app. We can do this relatively easily with background fetching. This chapter is all about fetching data on behalf of the user while the app isn't active in the foreground. Implementing this feature can greatly benefit your users because your app will have fresh content every time the user opens up the app. No pull to refresh is needed, and users love these...

Understanding how background fetch works

Any application that provides users with some form of data that updates over time is able to implement background fetch. An application that implements background fetch is woken up by iOS periodically, and it's given a small window of time to fetch and process new data that has become available. The OS expects applications to call a callback when they're done with fetching and processing the data. The application uses this callback to inform the OS about whether or not new data was fetched. We'll take a look at a broad overview of background fetch first and then we'll highlight each of the moving parts in more detail before implementing background fetch in the `FamilyMovies` application.

Looking at background fetch from a distance

Background fetch allows apps to download new content in the background without draining the battery because iOS manages the wake-up intervals as efficiently as possible. Since iOS will not allow your application to stay active for a very long time, you must keep in mind that your app might not be able to perform all the tasks you would like it to perform. If this is...

Implementing the prerequisites for background fetch

In order to implement background fetch, you will need to take the following three steps:

1. Add the background fetch capability to your app.
2. Ask iOS to wake your app up.
3. Implement `application(_:performFetchWithCompletionHandler:)` in `AppDelegate`.

We'll implement step 1 and 2 right now; step 3 will be implemented separately because this step will involve writing the code to fetch and update the movies using a helper struct.

Adding the background fetch capabilities

Every application has a list of capabilities they can opt in for. Some examples of these capabilities are Maps, Home Kit, and Background Modes. You can find the **`Capabilities`** tab in your project settings. If you select your project in the file navigator, you can see the **`Capabilities`** tab right next to your app's Delete settings.

If you select this tab, you will see a list of all the capabilities your app can implement. If you expand one of these capabilities, you're informed about what the capability does and what happens automatically if you enable it. If you expand the **`Background Modes`** capability, you'll see the following...

Updating movies in the background

The final step in enabling background fetch for our application is to add the `application(_:performFetchWithCompletionHandler:)` method. As explained before, this method is called by iOS whenever the app is awoken from the background and it allows us to perform an arbitrary amount of work. Once the app is done performing its task, it must call the completion handler that iOS has passed into this method.

Upon calling the completion handler, we will inform iOS about the results of the operation. It's important to correctly report this status because background fetch is intended to improve user experience. If you falsely report to iOS that you have new data all the time so your app is woken up more often, you're actually degrading user experience. You should trust the system to judge when your app is woken up. It's in the best interest of your users, their battery life, and ultimately your app to not abuse background fetch.

In order to efficiently implement background fetch, we will take the following steps:

1. Updating the data model so we can query the movie database more efficiently.
2. Refactoring the...

Updating movies in the background

We have almost all of the building blocks required to update movies in the background in place. All we need now is a way to fetch movies from the movie database using their remote ID instead of using the movie database search API.

In order to enable this way of querying movies, another fetch method is required. The simplest way to do this would be to copy and paste both the fetch and URL building methods and adjust them to enable fetching movies by ID. This isn't the best idea; if we add another fetch method or require more flexibility later, we will be in trouble. It's much better to refactor this into a more flexible format right away.

Preparing the helper struct

In order to maintain a clear overview of the available API endpoints, we will add a nested enum to the `MovieDBHelper`. Doing this will make other parts of our code more readable, and we can avoid errors and abstract away duplication with this enum. We'll make use of an associated value on the enum to hold on to the ID of a movie; this is convenient because the movie ID is part of the API endpoint.

Add the following code inside of the

Summary

We started this chapter with an overview of background fetch, how it works, and how it benefits your users. You learned about the prerequisites and best practices with regards to this feature. After establishing this basic understanding, you learned how background fetch works in concert with your application. Then, we continued to implement the required permissions and asked iOS to wake up the FamilyMovies application periodically so we could update the movies' ratings.

Once we did this, we needed to refactor a good portion of our application to accommodate the new feature. It's important to be able to recognize scenarios where refactoring is a good idea, especially if it enables smooth implementation of a feature later on. It also demonstrates that you don't have to think about every possible scenario for your code every time you implement a feature. After refactoring our application, we were finally able to implement background fetching behavior. In order to do this, we glanced over dispatch groups and how they allow you to group an arbitrary amount of asynchronous work together in order to be notified when all of the...

Chapter 12. Enriching Apps with the Camera, Motion, and Location

Your iPhone is a device that is capable of many things. So far, you've seen how the iPhone does networking and data management. You also know that the iPhone is really good at displaying content to your users with the help of components such as the collection view. However, the iPhone can do much more. It's a device that people carry with them every day, and it contains all kinds of sensors that provide unique opportunities to build apps that make use of these sensors for fun or even to truly improve people's lives.

People use their phones to navigate through traffic or to figure out where they're heading using maps and compass apps. The iPhone is also used to track workouts or even runs. Also, let's not forget that both the iPhone and iPad are used by people to take pictures and shoot videos. When we build apps, we should be aware that the iPhone's sensors can be used to create a rich and interactive experience.

In this chapter, we'll take a look at the following three sensors that are part of the iPhone. We'll have a look at some simple use cases for these sensors...

Accessing and using the camera the simple way

We will focus on building a nice login screen for a fictional app named ArtApp. This application is an **Augmented Reality** (**AR**) application that focuses on art. The login form fields will be positioned at the center of the screen. The camera will provide a background, and as the user moves their phone, we'll make the login field move around a bit. This effect will look similar to the iOS wallpaper parallax effect you might have noticed while using your iPhone. We'll use location data to provide a fictional indication of how many pieces of Augmented Reality art are near the user's current location. Before you get started, create a new single page app named ArtApp; you don't have to include Core Data or unit tests for this application.

Note

The application we will build in this chapter requires you to test it on a real device. The iOS simulator has limited support for GPS, but it doesn't have a camera or motion sensor access. Make sure that you have a device that runs iOS 11 nearby.

There are different ways to access the camera in iOS. A lot of applications don't need direct access to the...

Implementing CoreMotion

The `CoreMotion` framework implements many motion-related classes. The functionality for these classes varies, from counting steps with `CMStepCounter`, to figuring out the user's altitude with `CMAltimeter`, to accessing gyroscope data with `CMGyroData`, or even to reading whether a user is walking, running, or driving with `CMMotionActivityManager`. Reading the hardware sensors, such as the gyroscope, is done through an instance of `CMMotionManager`. The motion manager class provides an interface that enables developers to read data from the accelerometer, gyroscope, and more.

Our application will use a combination of sensors to function properly. However, before we get to that, we'll explore the available sensors a bit more because there are a lot of interesting features present in `CoreMotion`. We'll cover the pedometer first, and then we'll take a look at reading other sensors, such as altitude, accelerometer, and gyroscope.

Note

If you use `CoreMotion` in your application, the new privacy restrictions in iOS 10 dictate that you must include the `NSMotionUsageDescription` key in your app's `Info.plist`. This key is similar...

Using CoreLocation to locate your users

The final feature that needs to be added to the login screen of the ArtApp application is a label that shows where a user is currently located. Devices such as the iPhone contain a GPS chip that can accurately determine a user's location. Incorporating this into your apps can greatly improve the user experience. However, implementing location services poorly can and will frustrate your users, and could even drain their battery if you're not careful.

Before we add location services to ArtApp, we should explore the `CoreLocation` framework to see what features it has and how we can efficiently implement it into ArtApp and other applications. Some of the features and examples you will be familiar with are the following:

- Obtaining a user's location
- Geofencing
- Tracking a user while they're moving

Let's dive right in with the basics.

Obtaining a user's location

The documentation for `CoreLocation`states the following as a one-line definition of what `CoreLocation` does.

> *Determine the current latitude and longitude of a device. Configure and schedule the delivery of location-related events.*

In other words,...

Finishing the ArtApp login screen

In order to finish the login screen, you'll need to make sure that you have the correct prerequisites present from the `CoreLocation` examples. First and foremost, you'll need to make sure that the correct privacy keys are present in the app's `Info.plist`. Once you've done this, it's time to write the code that will query the user's location and update the user interface. After implementing the code, we'll get to implement the interface and hook up the outlets. Most of the code we need to add is already covered in the section on `CoreLocation`, so most of the code presented now won't be covered in depth.

Add the following two properties to the `ViewController` class:

```
@IBOutlet var nearbyArtLabel: UILabel!
let locationManager: CLLocationManager = CLLocationManager()
```

The need for these properties should speak for itself. We need a label in order to display the information we want to communicate to our user, and the location manager should be implemented so that the app can query the user's location. We don't need a property for the location because we will use the location updates we receive from the...

Summary

This chapter covered several of sensors that are available in iOS devices. First, you learned about each of the frameworks in isolation. We covered `AVFoundation`, `CoreMotion`, and `CoreLocation`. Piece by piece, you set up a login screen for an AR app. To properly do this, you had to learn a lot about all the motion sensors, the camera, and location services.

The sensors that modern devices contain are impressive, and they enable you to build great apps that truly immerse users. However, we should be honest with each other right now. The login screen we built might be nice, but it's not truly AR. The next chapter will show you how to build an application using ARKit, Apple's new framework that makes implementing AR a breeze.

Chapter 13. Extending the World with ARKit

When Apple announced iOS 11 at *WWDC 2017*, one of the new features that had many developers excited was ARKit. ARKit is Apple's new framework that enables developers to create great Augmented Reality experience using techniques that are already familiar to them.

This chapter will focus on teaching you how ARKit works and which techniques you can use to create a great experience for your users. In the previous chapter, you built a login screen for an Augmented Reality art gallery.

In this chapter, we will cover the following topics:

- Understanding how ARKit works
- Exploring SpriteKit
- Implementing an Augmented Reality gallery

Understanding how ARKit works

For quite some time now, AR has been a topic that many people and app creators have been interested in. Implementing AR has never been easy, and most AR applications have not quite lived up to the hype. Small details such as object placement and lighting have always made creating a proper AR experience extremely complex.

In the previous chapter, you explored a very basic and rudimental example of what an AR app should do. When you think about an AR app, you're thinking of an app that does at least the following:

- Show a camera view
- An overlay of content on top of the camera
- The content responds to movements from the device
- The content is attached to a certain point in the real world

Even though these four bullet points are relatively simple to come up with, they are really hard to implement on your own. Reading movement from the sensors in an iOS device is not very complex, we've already established that. However, making sense of this data and putting it to good use in an AR application is a very different story.

To help developers create great AR experiences, Apple has released a new framework called ARKit...

Exploring SpriteKit

As mentioned before, SpriteKit's main usage is to build two-dimensional games. The SpriteKit framework has been around for quite some time already and it has enabled developers to build many successful games over the years. SpriteKit contains a full-blown physics simulation engine and it is able to render many sprites at a time. A sprite represents a graphic in a game. It could be an image for the player, but also a coin, an enemy, or even the floor that a player walks on. When we mention sprites in the context of SpriteKit, it means that we refer to a node that is visible on the screen.

Because SpriteKit has a physics engine, it can also detect collisions between objects, apply forces to them, and more. This is pretty similar to what UIKit Dynamics is capable of. If you're a bit rusty on what UIKit Dynamics are and how they work, truncated

To render content, SpriteKit uses scenes. These scenes can be considered levels or major building parts of a game. In the context of AR, you will find that you typically only need a single scene. A SpriteKit scene is responsible for updating the position and state of the...

Implementing an Augmented Reality gallery

So far, most of our focus has been on exploring the theory and background of ARKit, but we haven't really seen it in action yet! To get a very basic idea of ARKit and what it can do, all you need to do is create a new AR project in Xcode and choose either SceneKit or SpriteKit as the rendering engine. If you choose SceneKit, you are given a starter project that positions a model of an airplane in the room. You can walk around this plane and view it from all angles. Have a look at the code that is required to set up this scene.

You might be amazed how little code is used! Earlier you learned how a SceneKit augmented reality experience essentially means that a 3D environment is created and the user's physical camera is the viewport into this 3D environment. The SceneKit starter project does a pretty good job of demonstrating this since you can immediately see how you are viewing the 3D scene through the device's camera.

If you start a new AR project in Xcode and you pick SpriteKit as the rendering engine, you are given the ability to add an emoji to the scene every time you tap the screen....

Summary

This chapter served as a primer in augmented reality. You even got to play with it a little bit by implementing your own basic AR experience. Of course there is much more to learn about augmented reality and with SceneKit you can create extremely immersive and mind-boggling experiences. Building an experience like this is, unfortunately, a slightly beyond the of scope of this book since it requires very specific knowledge about 3D modelling and 3D game design. If you are experienced with game development in a 3D world, transferring this knowledge to ARKit should be possible using the information in this chapter. And if you are not experienced in this area at all, at least you have a clear idea of where to start building great experiences right now.

I personally believe that augmented reality will thrive on iOS thanks to the introduction of ARKit. In the first weeks after the iOS 11 beta shipped, amazing demos have started popping up on the internet and I believe these demos are only the beginning. If you want to delve into AR and want some inspiration or if you're curious what others are doing with ARKit, make sure to...

Chapter 14. Exchanging Data with Drag and Drop

If you have ever wanted to move a couple of photos from one app on your iPad to another, chances are that you had to copy and paste each photo individually. If you compare this experience with the desktop, where you can grab one or multiple items and simply drag those items from one place to the other, iOS has a pretty subpar drag and drop experience. Actually, the drag and drop experience on iOS has essentially been non-existent up until iOS 11.

With iOS 11, users can finally enjoy a full drag and drop experience. Implementing drag and drop in your own apps has been made surprisingly simple since all you need to do is implement a couple of required methods from a handful of protocols. This chapter is aimed at showing you how to handle drag and drop in the augmented reality gallery you have built before with ARKit. The gallery app will be expanded so that users can add photos from their own photo library, the internet, or any other source to their art library.

This chapter is focused on the following topics:

- Understanding the drag and drop experience
- Implementing basic drag and drop...

Understanding the drag and drop experience

The drag and drop experience is quite simple to use; pick up an item on the screen, drag it somewhere else, and let it go to make the dragged item appear in the new place. However, iOS hasn't had this behavior until iOS 11. And even now, its full range of capabilities is only available on the iPad. Despite this limitation, the drag and drop experience is really powerful on both the iPhone and iPad.

Users can pick up an item from any app, and move it over to any other app that implements drag and drop, as they please. And dragging items is not even limited to just a single item, it's possible for users to pick up multiple items in a single drag session. The items that a user adds to their drag session don't even have to be of the same type; this makes the drag and drop experience extremely flexible and fluid. Imagine selecting some text and a picture in Safari and dragging them over to a note you're making. Both the image and the text can be added to a note in just a single gesture.

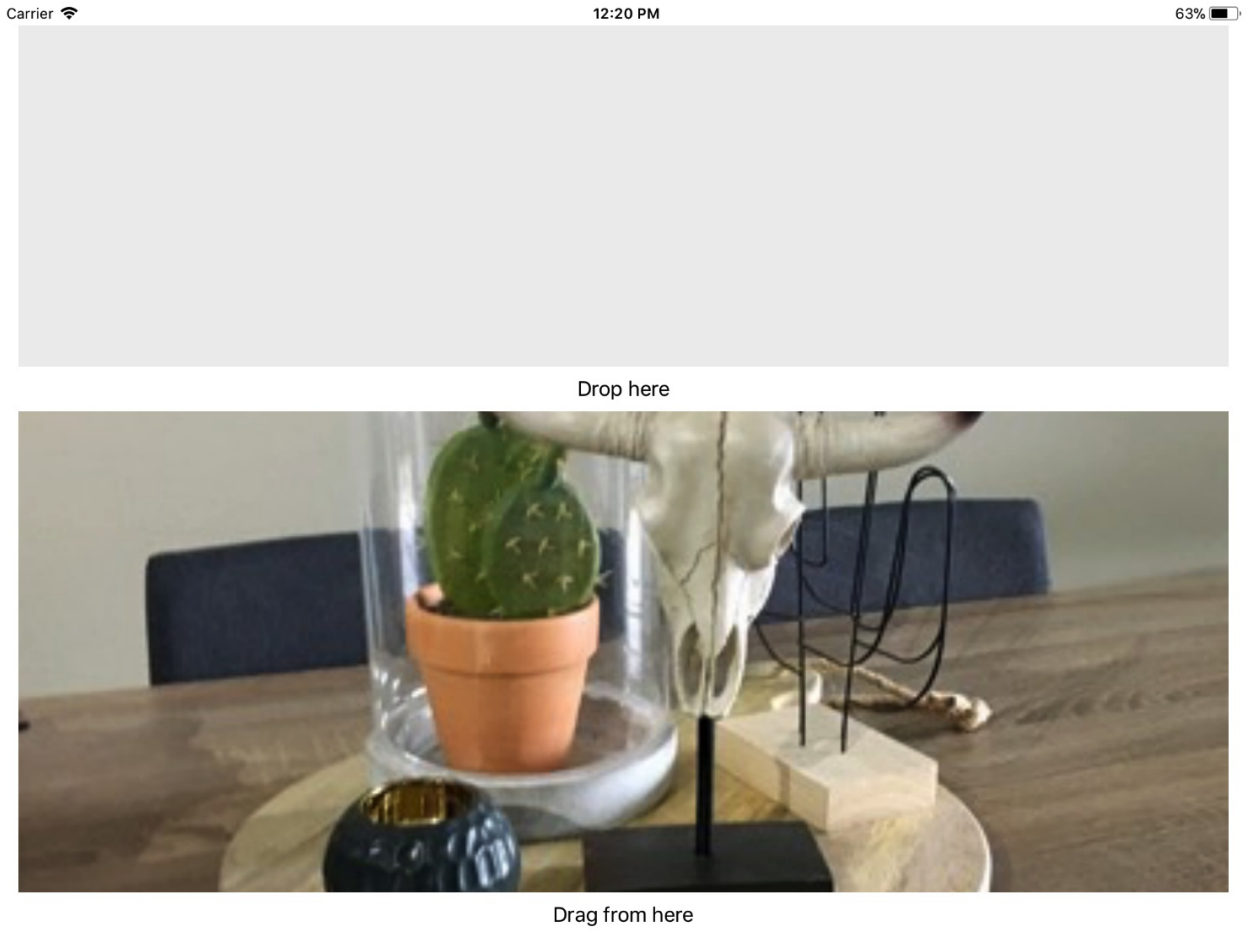
Unfortunately, apps are not able to handle drag and drop out of the box; you'll need to do a little bit of work...

Implementing basic drag and drop functionality

The previous section explained how drag and drop works from a theoretical point of view. This section focuses on implementing drag and drop in a sample app. First, we'll explore how a simple, regular implementation of drag and drop might work. Next, you'll see how Apple has implemented drag and drop for `UICollectionView` and `UITableView`. These two components have received special treatment, making it even easier to implement drag and drop in certain apps.

Adding drag and drop to a plain UIView

Before we implement drag and drop in the Augmented Reality gallery, let's see how we can implement a simple version of drag and drop with a simple view and an image. In the code bundle for this book, you'll find a sample project named `PlainDragDrop`. Open the starting version for this project and run it on an iPad simulator. You'll see the user interface shown in the following screenshot:



The goal for this example is to allow users to perform the following actions:

1. Drag the image to the drop area.
2. Drag an external image to the drop area.

3. Drag the bottom image to an external app.

While this might sound...

Customizing the drag and drop experience

Sometimes you will find yourself working on an app where the default implementations simply don't work for you. For instance, any application's drop delegate can propose a move action instead of a copy action. However, it's possible that you don't want to support this. You can restrict the allowed proposals by implementing `dragInteraction(_:sessionAllowsMoveOperation:)`. If you only want to allow copy operations, you can return `false` from this method. Another restriction you can enable through a delegate method is `dragInteraction(_:sessionIsRestrictedToDraggingApplication:)`. If you return true from this method, users won't be able to drag content from your app to another app.

Other methods on both the drag and the drop delegates are related to monitoring the state of the drag or drop session. Imagine that your app supports move proposals. When the user decides to move one or more objects from your app to another, you'll need to update the user interface accordingly once the drop is finished. You can implement `dragInteraction(_:sessionDidTransferItems:)` to implement this or perform any other...

Summary

This chapter showed you how to implement a smooth drag and drop experience for your users. Drag and drop has limited functionality on the iPhone but this doesn't stop it from being a powerful feature that can be used to support the dragging of contents within an application or to reorder a `CollectionView`. Collections and tables have received special treatment because they have their own delegate methods and enable you to easily access the cell that is selected by the user for dragging.

While drag and drop might seem complex at first glance, Apple did a great job of containing this complexity in a couple of relatively simple delegate methods that you can implement. A basic drag and drop implementation only requires you to implement fewer than a handful of methods, which is quite impressive for such a powerful feature! Now that your AR gallery supports drag and drop, let's make it a little bit smarter by adding some machine learning to it, shall we?

Chapter 15. Making Smarter Apps with CoreML

Over the past few years machine learning has gained popularity. For the past couple of iOS releases, Apple has mentioned how they used advanced machine learning to improve Siri, make smart suggestions for the Apple Keyboard on iOS, and improve Spotlight and many more features. While this is all great, machine learning has never been easy to implement on a mobile device. And if you do succeed in implementing machine learning, chances are that your implementation is not the most performant and energy efficient implementation possible.

Apple aims to solve this problem in iOS 11 with the **CoreML** framework. CoreML is Apple's solution to all problems they have run into themselves while implementing machine learning for iOS. As a result, CoreML should have the fastest, most efficient implementations for working with complex machine learning models through an interface that is as simple and flexible as possible.

In this chapter you will learn what machine learning is, how it works, and how you can use trained models in your own apps. We'll also have a brief look at the new vision framework that...

Understanding what machine learning is

A lot of developers sooner or later hear about the topic of machine learning, deep learning, or neural networks. You might have already heard about these topics. If you have, you know that machine learning is a pretty complex field that requires very specific domain knowledge. However, machine learning is becoming bigger and more popular every single day and it is used to improve many different types of applications.

For instance, machine learning can be used to predict what type of content a certain user might like to see in a music app based on music that they already have in their library, or to automatically tag faces in photos, connecting them to people in the user's contact list. It can even be used to predict costs for certain products or services based on past data. While this seems like magic, the flow for creating machine learning experiences like these can be split roughly in two phases:

1. Training a model.
2. Using inference to obtain a result from the model.

In order to perform the first step, large amounts of high quality data must be collected. If you're going to train a model that...

Understanding CoreML

Due to the complex nature of machine learning and using trained models, Apple has built CoreML to make incorporating a trained model as simple as possible. On top of making CoreML as simple as possible, another goal was to ensure that whenever you implement machine learning using CoreML, your implementation is as fast as possible, and energy efficient. Since Apple has been enhancing iOS with machine learning for a couple of years now, they have loads of experience implementing complex models in apps.

If you have ever done research into machine learning, you might have come across cloud based solutions. Typically, you send a bunch of data to such a cloud based solution and the result is passed back as a response to your request. CoreML is very different since the model lives on the device instead of in the cloud. This means that your user's data never has to leave the device, which is very good for your user's privacy. Also, having your trained model on the device means that no internet connection is required to use CoreML which saves both time and precious data. And since there is no potential bottleneck in...

Combining CoreML and computer vision

When you work on an app that works with photos or live camera footage, there are several things you might like to do. For instance, it could be desirable to detect faces in an image. Or maybe you would like to detect rectangular areas such as traffic signs. You could also be looking for something more sophisticated like detecting the dominant object in a picture.

In order to work with computer vision in your apps, Apple has created the Vision framework. You can combine the Vision framework and CoreML to perform some pretty sophisticated image recognition. Before we get to implementing the detection of dominant objects in our user's Augmented Reality galleries, let's take a quick look at the Vision framework so you have an idea of what it's capable of and when you might like to use it.

Understanding the Vision framework

The Vision framework is capable of many different tasks that revolve around computer vision. The Vision framework is built upon several powerful deep learning techniques in order to enable state of the art facial recognition, text recognition, barcode detection, and more. When...

Summary

This chapter wraps up the work we'll do on the Augmented Reality app. By now you have built an app that implements some of iOS 11's most exiting new technologies. In this chapter we focused on CoreML, Apple's machine learning framework. You saw that adding a machine learning model to your app is extremely simple since you only have to drag it to Xcode and add it to your target app. You also learned how you can obtain models and where to look in order to convert existing models to CoreML models. Creating a machine learning model is not simple so it's great that Apple has made it so simple to implement machine learning by embedding trained models in your apps.

In addition to CoreML, we also looked at the Vision framework. Vision combines the power of CoreML and smart image analysis to create an extremely powerful framework that can perform a huge amount of work on images. Convenient requests like facial landmark detection, text detection, and barcode detection are available out of the box without adding any machine learning models to your app. If you're looking for custom analysis, you can always add your own CoreML model...

Chapter 16. Increasing Your App's Discoverability with Spotlight and Universal Links

Many users of macOS and iOS love one feature in particular: Spotlight search. Spotlight keeps an index of all files, contacts, contents, and more on your Mac, and it's been in iOS too for a while now. With iOS 9, Apple made it possible for developers to index the contents of their own apps, enabling users to discover app contents right from the Spotlight search they know and love. Ever since opening up Spotlight on iOS to developers, Apple has been pushing to make Spotlight better, more relevant, more impactful and more helpful to users. In this chapter, we'll explore what Spotlight can do for your apps and how you can make use of the `CoreSpotlight` APIs to provide your users with an amazing search experience that helps them find and navigate your app.

In this chapter, the following topics are covered:

- Understanding Spotlight search
- Adding your app contents to the Spotlight index
- Handling search result selection
- Adhering to Spotlight best practices
- Increasing your app's visibility with Universal Links

Let's not waste any time and jump right in to...

Understanding Spotlight search

If you have a Mac, which you most likely do, you have probably used the blazingly fast Spotlight search feature on it. Furthermore, if you love this feature on your Mac, you have probably used it on iOS as well. Spotlight is a highly optimized search engine that enables you to find content throughout your device.

Note

If you haven't used Spotlight, try swiping down on your home screen on iOS and use the search box. This is Spotlight. Or, on your Mac, press *command* + space to open the Spotlight search dialog.

Spotlight has been around since iOS 3. However, it wasn't until iOS 9 that Apple opened up Spotlight to developers. This means that, starting with iOS 9, Spotlight indexes everything from web results, apps, app store results, and any content you add to the Spotlight index yourself. Results are presented with an image, a title, and some extra information, and sometimes extra buttons are added to make a phone call or to start turn-by-turn navigation to an address.

The results that are displayed by Spotlight are a mix of public or online contents and private results. If your app has indexed items that...

Adding your app contents to the Spotlight index

If you have ever worked on a website, you must have heard something about SEO (Search Engine Optimization). More importantly, you will know that any website you create and publish is indexed by several search engines. All you have to do is make sure that you write semantic and structured HTML markup and any web spider will understand what your website is about and what parts of it are more important. Search engines, such as Google, have indexed billions of web pages based on their contents and semantic markup.

Apps tend to be a little less neatly structured, and crawling them is a lot harder if not impossible. There is no structured way to figure out what content is on screen and what this content means. Also, more importantly, a lot of content you'd want to index is only available to users who have logged in or created content of their own.

This is why Apple decided that the developers themselves probably know their app's contents best and should be in charge about how, when, and why a particular content is indexed. Even though this does put a little bit of manual burden on the...

Increasing your app's visibility with Universal Links

A Universal Link is very similar to a deep link. Deep links allow apps to link users straight into a certain section of an application. Before Universal Links, developers had to use a custom URL scheme to create their deep links.

You might have seen a URL with a custom scheme in the past. These URLs are easily recognized and look as follows:

```
familymovies://FamilyMember/jack
```

It's obvious that this isn't a regular web URL because web URLs start with a scheme of either `http://` or `https://`. An application can register itself as capable of opening URLs with a certain scheme. So, the `familymovies` app we've been working on could manifest itself as a handler of `familymovies://urls`.

However, there are a couple of downsides to this approach. First and foremost, this URL isn't sharable at all. You can't send this URL to any friends that don't have the same app installed. If you were to send this URL to somebody and they didn't have the corresponding app installed, they wouldn't be able to open this URL. This is inconvenient because, in order for others to access the same content, assuming...

Summary

This chapter focused on getting your app indexed with the powerful `CoreSpotlight` framework. You saw how you can use user activities to index items that the user has already seen and how you can use searchable items to index content that the user might not have seen. You also learned how to make use of unique identifiers to ensure that you don't end up with duplicate entries in the search index. We also took a look at continuing user activities from either a user activity or a searchable item.

After you knew everything about how `CoreSpotlight` can index your items, you learned about Universal Links and web content. It's important to think about publishing content on the web because it helps your Spotlight indexing tremendously and it enables Apple to show your Spotlight results in Safari. We also covered the metadata that you should add to your web pages and Smart App Banners.

In the next chapter, we'll make our app visible in one more place on the user's device by creating a widget for the notification center.

Chapter 17. Instant Information with a Notification Center Widget

When Apple added the Notification Center to iOS in iOS 6, they also added some stock widgets to a section called the **Today View**. These widgets had limited functionality and, as developers, we had no ability to add our own widgets to this view. Users were stuck with a weather widget, one for stocks, one for the calendar, and a few more.

It wasn't until iOS 8 that Apple decided that developers should be able to create extensions for the Notification Center in the form of widgets. Together with that addition, Apple has added the ability for developers to create several extensions that integrate apps with iOS. This chapter will focus mainly on creating a **Today Extension:** A widget.

You'll learn what the life cycle of an extension looks like and how extensions integrate with the OS and your app. The basis for this extension will be a small, simple app that has content that varies throughout the day.

We will cover the following topics in this chapter:

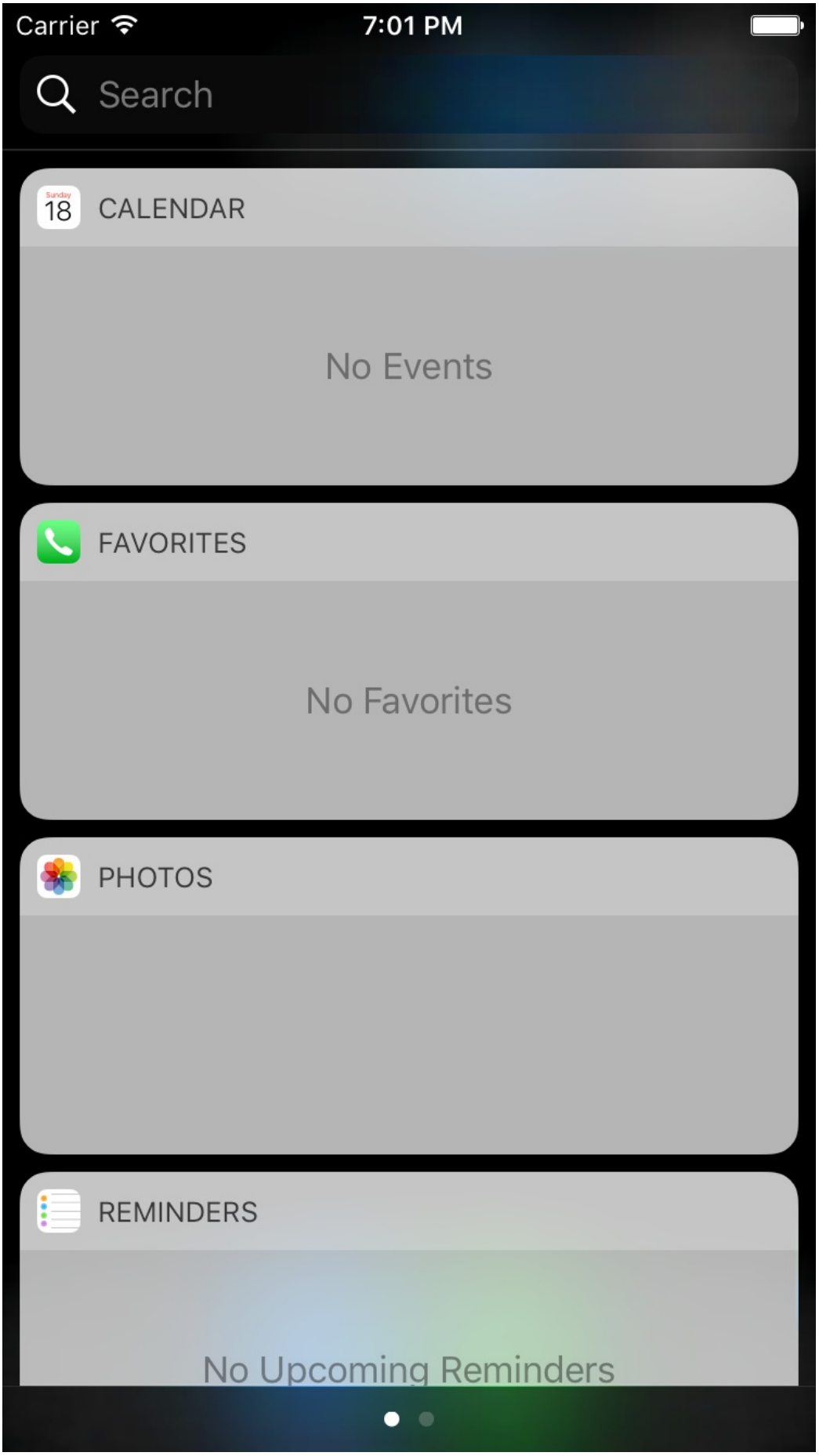
- Understanding the anatomy of a Today Extension
- Adding a Today Extension to your app

Even though we only have two bullet...

Understanding the anatomy of a Today Extension

The terms `Today Extension` and widget can be used interchangeably; they both refer to a component that is present in iOS's Notification Center. In this chapter, we'll mostly stick to the term widget, unless we're talking more broadly in the context of extensions.

If you swipe down from the top of the screen to open Spotlight and swipe right after that, you're presented with the Today View. On the simulator, this view tends to look rather empty, but on your device, there's probably a lot more going on. The following screenshot shows the Today View on the simulator with a couple of widgets added to it:


Carrier
7:01 PM
Search
CALENDAR
No Events
FAVORITES
No Favorites
PHOTOS
REMINDERS
No Upcoming Reminders

Users can scroll to the bottom of this view and manage their widgets from there. They can add new widgets and remove existing ones. All these widgets have one thing in common: They provide relevant information for the current moment or day. For the **CALENDAR**, this means showing events that are planned for today or tomorrow. The **FAVORITES** widget in the screenshot usually shows contacts that the user interacts with often.

Any widget that you add to your app should aim to provide your users...

Adding a Today Extension to your app

In this book's code bundle, you can find an app named *The Daily Quote*. If you want to follow along with this section of the chapter, its a great idea to go ahead and check out the starter code for this app. If you look at **`Main.storyboard`** and the `ViewController.swift` file, you'll find that there isn't too much going on there. The interface just contains two labels, and the `ViewController` grabs a `Quote` and displays it.

Even though it doesn't look like much, each file contains something cool. If you select one of the labels in the **`Storyboard`** and examine its attributes, you'll find that the font for the quote itself is actually **`Title 1`** and the font for the quote created is **`Caption 1`**. This is different from the default system font that we normally use. Selecting one of the predefined styles enables the text in our app to dynamically adjust to the user's preferences. This is great from an accessibility standpoint and it costs us very little effort.

Note

If your interface allows it, its recommended that you make use of accessible type, simply because it will make your app easier to use for all your...

Sharing data with App Groups

When you're developing an application or extension that's part of a suite of apps, extensions, or a combination of both, you're probably going to want to share some code and data. You already saw how to share code by including a file in multiple targets. This doesn't immediately allow for data sharing, though.

To share data between apps, you make use of **`App Groups`**. An **`App Group`** is a group of applications that have the **`App Group`**s entitlement enabled in their capabilities. More importantly, the applications in an **`App Group`** must specify that they want to share information with a certain group.

An **`App Group`** is specified as a unique string. Every app or extension that is part of an **`App Group`** must have this same unique string added to the list of groups the target belongs to. A single target can be part of several **`App Groups`**. To enable **`App Groups`**, select your target and go to its Capabilities tab. Next, enable the **`App Groups`** capability and specify a unique identifier for your group.

Let's do this for the *The Daily Quote* and its **`Today Extension`** right away. Refer to the following screenshot to make sure that...

Summary

This chapter was all about app extensions, and more specifically **`Today Extension`**. We started by exploring the basic anatomy of an extension in iOS. You learned that extensions are mostly view controllers that live separately from your main application. You learned how you can add an extension target to your project and how to run it in the Simulator. After getting the extension up-and-running, you learned how to share pieces of code and functionality between your application and the extension by adding a file to both targets.

This left the app and widget in a state where they were unable to share data. We implemented data sharing by enabling **`App Group`**s so that our app and its extension use a shared `UserDefaults` store. With this powerful tool and knowledge under your belt, we're going to delve even more deeply into the iOS Notification Center. We're going to implement the new notification features that iOS 10 implements through Notification Extensions.

Chapter 18. Implementing Rich Notifications

The previous chapter introduced you to App Extensions. You learned that App Extensions enables you to tightly integrate your app with iOS. In iOS 10, Apple added extension possibilities for notifications while also completely revamping how notifications work. In this chapter, we're going to explore how notifications work and how you can implement extensions that will have a tremendous positive impact on the user experience of receiving notifications.

First, we will look at notifications from a broad perspective and how they manifest throughout iOS. Even though we won't go into a lot of detail straight away, you'll gain a deep understanding of notifications by looking at the different types of notification and the different ways of scheduling notifications that are available. Next, we'll have a detailed look at how your app can schedule and handle notifications. Finally, we'll look at the extension points that have been added by Apple in order to implement richer and more engaging notifications.

By the end of this chapter you'll know everything there is to know about notifications and...

Gaining a deep understanding of notifications

Since you're planning to become a master at iOS development, you've probably used an iPhone or iPad more than a couple of times. It's also very likely that you've had applications sending you notifications while you were using the device, or maybe while the device was sitting idle on your desk, in your pocket, or somewhere else.

Notifications are a perfect way to inform users about information in your app that is of interest to them. New messages in a messaging app, breaking news events, or friend requests in social apps are just a few examples of great scenarios for notifications. From the get-go, it's important that you're aware of the fact that a lot of apps are fighting for the user's attention at all times. Notifications are a great way to gain the user's attention, but if you send too many notifications, or only use notifications as a marketing tool instead of a way to provide meaningful information, it's very likely that the user is going to disable push notifications for your app.

There are four forms of notification that you can send to your users:

- Sounds
- Vibrations
- Visual...

Scheduling and handling notifications

Every notification you see on iOS has been scheduled or pushed to the user in one way or an other. But before an app is even allowed to send notifications, we actually have to go through a few steps. The steps you need to follow can be roughly divided as follows:

1. Registering for notifications.
2. Creating notification contents.
3. Scheduling your notification.
4. Handling incoming notifications.

Not every step is identical for push and local notifications, so whenever relevant, the difference will be described. Because of the UserNotifications framework, these differences will be minimal, but they're there. We'll expand the quotes app from the previous chapter with a daily notification as a way to showcase and test notifications.

Registering for notifications

The first step you must go through if you're planning on showing notifications to your user is asking permission to do so. We call this registering for notifications. It's important that you make sure your user understands why they're being prompted for this permission. After all, you're interrupting your user while they're trying to achieve something...

Implementing Notification Extensions

The most exciting new feature in the realm of notifications is Notification Extensions. Apple has added two extension points to notifications, which enables you to take the notification experience up a notch from custom actions. The available extensions are Service Extensions and Content Extensions. These extensions are both very powerful and relatively simple to implement. We'll have a look at Service Extensions first.

Adding a Service Extension to your app

Service Extensions are intended to act as middleware for push notifications. The Service Extension receives the notification before it's displayed to the user. This allows you to manipulate or enrich the content before it's displayed.

A Service Extension is perfect if you're implementing end-to-end encryption, for example. Before this extension was introduced in iOS 10, notifications had to be sent in plain text. This means that any app that implements end-to-end encryption still wasn't completely secure because push notifications were not encrypted. Since iOS 10, you can push the encrypted message from your server to the device and decrypt...

Summary

In this chapter, we've explored the amazing world of notifications. Even though notifications are simple in nature and short-lived, they are a vital part of the iOS ecosystem and users rely on them to provide timely, interesting, and relevant updates about subjects that matter to them, right then and there. You saw how to go from the basics, simply scheduling a notification, to more advanced subjects such as custom actions. You saw that Service Extensions allow you to implement great new features, such as true end-to-end encryption, or enrich push notifications with content that is stored on the device.

Finally, we explored Content Extensions and you saw how a Content Extension can take a simple, plain notification, and make it interesting, rich, and more relevant. The example of a calendar that appears with event invites comes to mind immediately, but the possibilities are endless. Proper usage of Notification Extensions will truly make your app stand out in a positive and engaging way.

The next chapter focuses on another extension; the iMessage extension!

Chapter 19. Extending iMessage

As part of Apple's effort to introduce more and more extension points into the iOS ecosystem, iMessage extensions have been introduced in iOS 10. These extensions allow your users to access and share content from your app with their friends right inside the Messages app. This chapter will show you exactly how to tap into these extensions points and build your own iMessage extensions.

We'll go over two types of extension: Sticker packs and iMessage apps. Note that we call these extensions apps and not just extensions. This is because iMessage extensions behave a lot like apps even though they're not apps. Also, iMessage apps are distributed through their own iMessage App Store. More on this later.

As with other extensions we've covered so far, the first step is to gain a broad understanding of iMessage apps and what possibilities they give you as a developer. Once we've established this basic understanding, we'll dive into several aspects of iMessage apps in more detail. To be more specific, the following are the topics that we'll cover in this chapter:

- Creating a sticker pack for iMessage

Understanding iMessage apps

An iMessage app is really an app extension that lives inside iMessage. It's kind of a weird type of extension because it behaves like a hybrid between an extension and an application. Extensions typically contain a single view controller and, more importantly, they can't be distributed through the iOS ecosystem without containing an application.

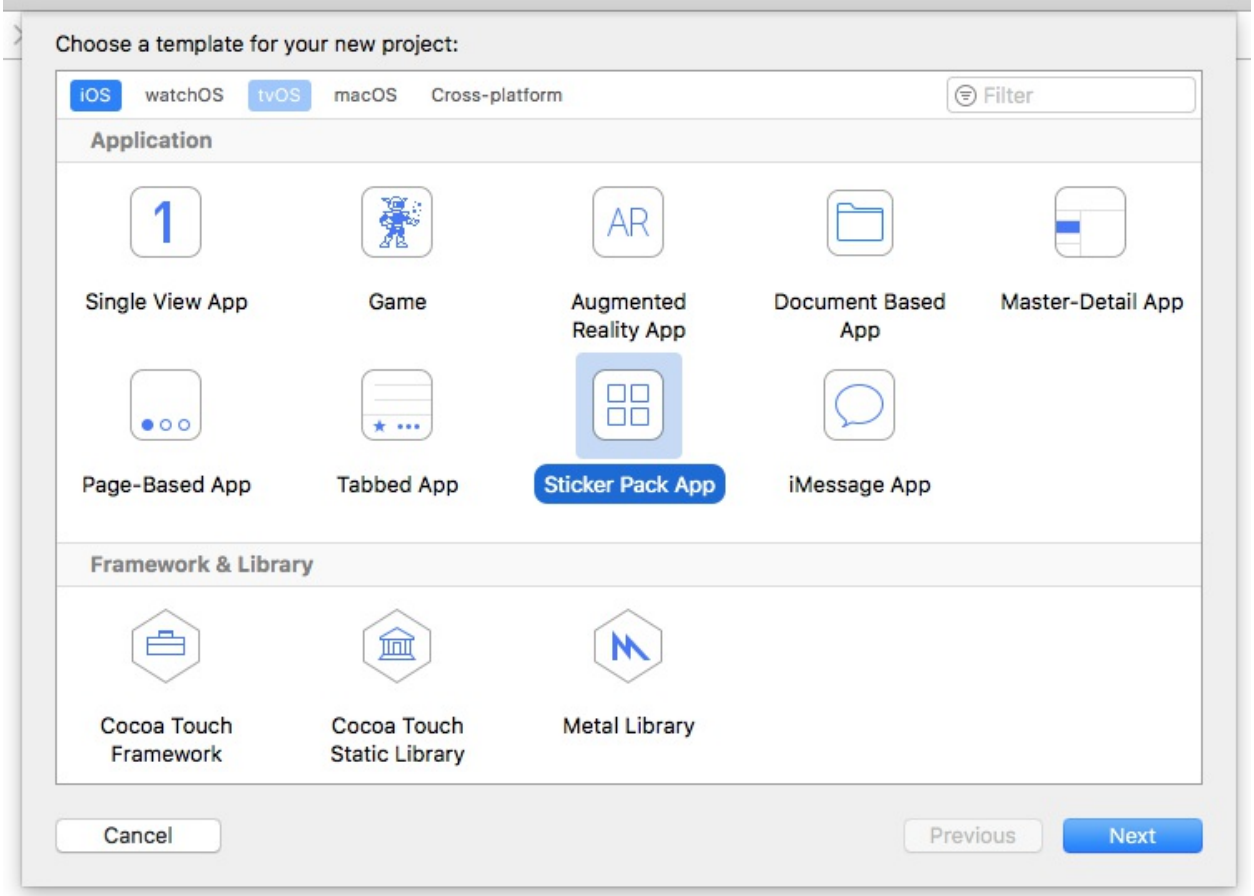
This rule does not apply to iMessage apps. An iMessage app can be distributed directly through the iMessage app store without a containing application. This means that you can build an iMessage app, for example, a sticker pack, and distribute it without associating any other app to it. Also, you can add in-app purchases to an iMessage app. In-app purchases are normally unavailable to extensions, which makes iMessage apps behave more like an app than an extension.

Apart from distribution and in-app purchases, iMessage apps behave just like other extensions. They are created on behalf of a host application, Messages in this case, and they have a relatively short lifespan. An iMessage app specializes in sending messages from one user to another. These messages are...

Creating an iMessage sticker pack

Stickers in iMessage are a fun way to share images with your friends. Stickers can be slapped onto a message as a response to a message you've received or just for fun with no particular reason at all.

To create a sticker pack, open Xcode and create a new project. One of the available project templates is the **`Sticker Pack App`**; select it and click **`Next`**. Give your sticker pack a name and click **`Next`** again to create your project:



In the generated project, you'll find two folders. One is named `Stickers.xcstickers` and the other is named `Products`. We're only interested in the `stickers` folder. If you open it, you'll find an app icon template and a folder named `Sticker Pack`.

All you need to do to create your sticker pack is to drag images into this `Sticker Pack` folder. After doing this, you can build and run your extension and try it out in the simulator. You'll find that the simulator comes with two active conversations in the Messages app. These are dummy conversations and you use them to test your iMessage app.

The simulator cannot send messages outside the simulator so you must use these predefined...

Implementing custom, interactive iMessage apps

Sticker apps are nice, but they're not particularly useful for every use case. We can build far more interesting and interactive applications for iMessage through the Messages framework. Some of the larger, well-known apps on iOS have been able to implement iMessage applications that make sharing content from their apps easier. There are people that have built games in iMessage already. The Messages framework enables developers to build a wide range of extensions straight into the Messages app.

We've just seen how you can build sticker packs and how you can create a somewhat customized sticker pack by picking the app template instead of the sticker pack template in Xcode. We haven't gone in-depth into the different life cycle methods that Xcode generates for us when we create a new iMessage app.

Let's do this while we build an iMessage app for `The Daily Quote`, the app you've already built a notification extension and a widget for. First, we'll look at the life cycle of an iMessage app. Then we'll implement the compact view for our app. Finally, we'll implement the expanded view for our...

Understanding sessions, messages, and conversations

So far we have been looking at user interface-related aspects of iMessage apps only. When you're creating an iMessage app, your app will sooner or later have to send a message. To do this, we make use of `MSMessage` and `MSSession`. In addition to messages and sessions, there is the `MSConversation` class. These three classes together enable you to send messages, identify recipients in conversations, and even update or collapse existing messages in the messages transcript.

When an iMessage extension is activated, `willBecomeActive(with:)` is called in the `MessagesViewController`. This method receives a `MSConversation` instance that you can use to send messages, stickers, and even attachments. More importantly, the conversation contains unique identifiers for participants in the conversation and the currently selected message.

The `localParticipantIdentifier` and `remoteParticipantIdentifier` properties respectively, identify the current user of your app, the person who is sending messages with your app, and the recipients of these messages. Note that these identifiers are unique to your app...

Summary

In this chapter, you've learned everything about iMessage apps. This new type of extension, which is very similar to an app, is an exciting, powerful new way for you to integrate your apps further into the iOS ecosystem. You now know how you can create a sticker pack to provide your users with a fun, refreshing way to share imagery from your app. You saw how to create a standard and custom sticker pack and you even learned how to create interactive experiences that allow your users to compose stickers together by using a messages session and conversation objects.

Most importantly, you learned that, even though iMessage apps can be very complex, they can also be simple. Some iMessage apps will make heavy use of interactive messages, others are simply standard sticker packs, or somewhere in between, like The Daily Quote. The possibilities are endless and you can have a lot of fun with iMessage extensions. Just be aware that during testing you'll only be able to use your extension in the simulator so, unfortunately, you can't surprise your friends by suddenly sending them stickers from a sticker pack you're still...

Chapter 20. Integrating Your App with Siri

At the *WWDC* conference in 2016, Apple announced a new framework named **SiriKit**. SiriKit enables developers to integrate their apps with Apple's digital assistant. This chapter will teach you all the ins and outs of SiriKit, the framework that's used to build extensions for Siri. Similar to iMessage apps, widgets, and notifications, Siri makes use of extensions to integrate with other apps. This is convenient because a lot of the knowledge we've already gained from studying extensions in iOS carries over to Siri extensions.

Even though SiriKit is built upon familiar ideas and concepts, it also introduces a whole new set of terminologies and implementation details. One example of this is that Siri extensions use intents and a specialized vocabulary to integrate with your app. These concepts have been introduced especially for Siri extensions, so you'll need to learn about them before you can implement a Siri extension.

The topics covered in this chapter show the entire flow that your extension goes through when a user uses Siri to make a request for your app. In this chapter, we won't be...

Understanding intents and vocabularies

Siri is a powerful, smart, and ever-learning personal assistant that aims to give natural responses to natural speech input. This means that there is often more than one way to say something to Siri. Some users like to be extremely polite to Siri, saying please and thank you whenever they ask for something. Other users like to be short and to the point; they simply tell Siri what they want and that's it.

This means that Siri has to be really smart about how it interprets language and how it converts the user's requests to actionable items. Not only does Siri take into account the language used, it's also aware of how a user is using Siri. If user activates Siri by saying *Hey Siri*, Siri will be more vocal and verbose than when a user activates Siri by pressing and holding the home button, because it's likely that the user is not looking at their device if they didn't press the home button.

To convert a user's spoken requests into actions, Siri uses intents. An intent is a way to describe an action. These intents are supported by app-specific vocabularies; this allows you to make sure that your...

Adding intents to your extension

Any time a user asks Siri for something, the user's query is resolved to an intent. Every intent has a number of parameters or variables associated with it. Siri will always attempt to fill in these parameters to the best of its ability. Every app that integrates with Siri should have an extension. This extension is used by Siri to make sure that all the parts of an intent are present and valid.

You might expect intents to be a part of SiriKit. In reality, they're not. Intents have their own framework in iOS, enabling other applications, such as Maps, to make use of intents as well. This means that any knowledge regarding intents that you will obtain in this chapter translates over to other extensions that make use of intents.

To enable Siri handles certain intents with your applications, you must specifically register for these intents in your extension's `Info.plist` file. In this book's repository, you'll find a project named `SiriKitMessaging`. We'll add extensions to this application in order to create a fake messaging app.

Before we explore the project, let's set it up so that it is able to...

Adding a custom UI to Siri

When your user is using Siri, they aren't always looking at their device. But when they are, it's desirable that the experience a user has when using your app through Siri looks and feels a lot like when they're directly interacting with your app. One of the tools you've already seen is custom vocabularies. You can use a vocabulary to map user and app-specific terms to Siri's vocabulary.

Another way we can customize the Siri experience is through an **`Intents UI Extension`**. Whenever you add an **`Intents Extension`** to your project, Xcode asks you if you also want to add an interface extension. If you select this checkbox, you don't have to do anything to add the UI extension since it's already there. However, if you didn't check the checkbox, you should add a new **`Intents UI Extension`** through the targets menu, as you do for all extensions.

A custom user interface for an intent works a lot like other UI extensions. When you create the extension, you're given a storyboard, a view controller, and a `.plist` file. The `.plist` file is expected to specify all of the intents that this specific UI extension can handle. In...

Summary

In this chapter, you've seen everything you need to know about integrating your app with Siri. You know everything about intents and the Intents framework. We've implemented a Siri extension that fakes sending messages to groups of users. We saw that there are methods that can be implemented to help Siri resolve a user's query and fill in the missing parameters that Siri needs to build an intent. You also learned that you can add custom vocabularies to Siri, meaning that your Siri extension can make use of user and app-specific terms. These custom vocabularies allow your users to communicate with Siri in terms that fit in with terminology that you might use in your app. Finally, you saw how to create a custom user interface for your Siri extensions, enabling a maximum amount of app recognition to the user.

A good integration with Siri can improve your app tremendously, but only if it fits into one of the predefined domains. Even though you can stretch the meaning of certain intents slightly, it's recommended to only attempt to integrate with Siri if it truly makes sense to do so. A bad implementation will frustrate users...

Chapter 21. Ensuring App Quality with Tests

In all chapters up to this one, we've mostly focused on writing code that's used in our apps. The apps we've built are small and can be tested manually, quite quickly. However, this approach doesn't scale well if your apps become larger. This approach also doesn't scale if you want to verify lots of different user input, lots of screens, complex logic, or even if you want to run tests on many different devices.

Xcode comes with built-in testing tools. These tools allow you to write tests so you can make sure that all of the business logic for your app works as expected. More importantly, you can test that your user interface works and behaves as intended in many different automated scenarios.

Many developers tend to shy away from testing, postpone it until the end of the project, or simply don't do it at all. The reason for this is often that it's pretty hard to figure out how to write proper tests if you just start out. For instance, many developers feel like large parts of testing are so obvious that writing tests feels silly. However, if not approached properly, tests can be more of...

Testing logic with XCTest

If you're just starting out with testing your app logic, there are a couple of thoughts you might already have on the subject, or maybe none at all. To start testing your code, you don't need to have a computer science degree or spend days studying the absolute best way to test your code. In fact, chances are that you're already sort of testing your code and you don't even know it.

So, what does it really mean to test your code? That's exactly what we'll cover in this section. First, we'll look at testing code and the different types of test you might want to write. Then we'll dive into `XCTest` and setting up a test suite for an app. Finally, we're going to see how we can optimally test some real code and how we should refactor it to improve testability.

Understanding what it means to test code

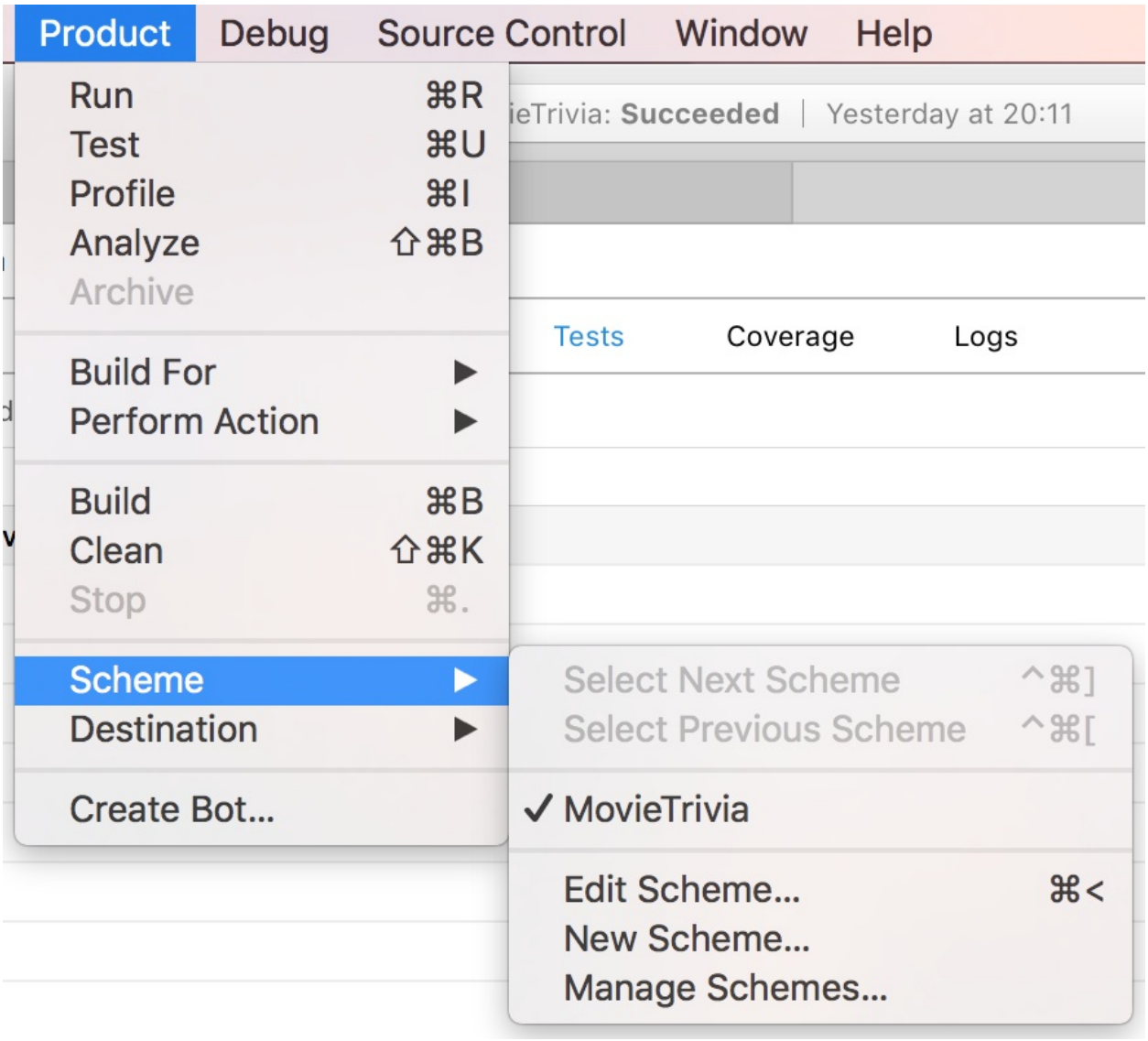
When you test your code, you're essentially making sure that certain inputs produce the desired outputs. A very basic example of a test would be to make sure that calling a method that increments its input by a given value does, in fact, produce the output we expect.

Whenever you launch your application in a simulator...

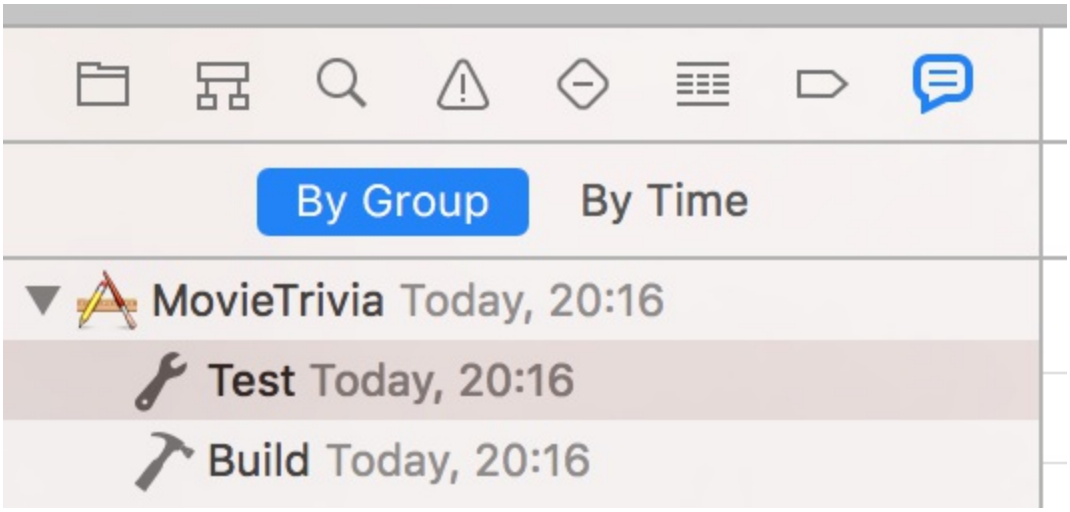
Gaining insights through code coverage.

With the introduction of Xcode 7, Apple has greatly improved its testing tools. One of the tools newly introduced with Xcode 7 is Code Coverage. The Code Coverage tool aims to provide us with insights about how thorough our tests are. Not only that, it also informs us about which parts of our code are tested in our unit tests and which parts of our code aren't.

To enable `Code Coverage`, first open the scheme editor through the `Product` | `Scheme` menu as shown in the following screenshot:



Select the testing action and make sure the `Gather coverage` data checkbox is checked as shown in the following screenshot:



Note

You can also press Cmd + < to quickly open the scheme editor.

After doing this, close the scheme editor and run your tests. This time, Xcode will monitor which parts of your code have been executed during this tests, and which parts haven't. This information can give you some good insights about which parts of your code could use some more testing. To see the coverage data, open the Report navigator in the leftmost sidebar in Xcode. The rightmost icon in this sidebar represents the Report...

Testing the user interface with XCUITest

Knowing that most of your app logic is covered with tests is great. What's not so great, however, is adding your view controllers to your logic test. Before Xcode 7 came out, you had to put quite some effort into testing view controllers. This led to people simply not testing a lot of view controller code, which would ultimately lead to bugs in the user interface that could have been prevented with tests. We were able to perform some degree of tests using UI Automation in Instruments, but it simply wasn't great.

In Xcode 7, Apple introduced interface testing and the `XCUITest` framework. With this feature, we suddenly gained a great framework to write interface tests in; we also gained a tool that allows us to record these tests as we go through them manually.

To top it all off, `XCUITest` uses the accessibility features in iOS to gain access to the interface. This means that implementing user interface tests forces you to put at least a little bit of effort into accessibility for your applications. Apps that are hard to navigate through accessibility features will be harder to test than apps...

Summary

Congratulations! You've made it to the end of this lengthy, information-packed chapter. You should know enough about testing and accessibility right now to begin exploring testing in greater depth than we have in this chapter. No matter how small or big your app is, writing automated tests will ensure that your app is of high-quality. More importantly, instead of assuming that something works because it worked before, your automated tests will guarantee that it works because your tests don't pass if you broke your code.

You also learned that writing testable code sometimes requires you to refactor large portions of code. More often than not, these refactoring sessions leave your code in a much better state than it was before. Code that is easy to test is often cleaner and more robust than code that is hard to test. If you're interested in learning more about the topic of testing, you could have a look at *Test-Driven iOS Development with Swift 3*, written by Dr. Dominik Hauser. This book takes a practical approach to developing an iOS application using TDD principles. Now that you know how to cover your app with tests, let's...

Chapter 22. Discovering Bottlenecks with Instruments

In order to debug and improve your apps, you need to understand which tools are available to you. One of these tools is called **`Instruments`**. **`Instruments`** is a collection of measurement tools that help you to profile and analyze your app in order to debug and detect complex problems or performance bottlenecks. For example, Instruments can help you figure out if your app is suffering from memory leaks. Tracking a memory leak blindly is tedious and nearly impossible. A good tool such as Instruments helps you track down several possible causes for a memory leak, saving both your time and your sanity.

In this chapter, we're going to look at an app named *Instrumental*. This app is a very plain app and it's not even close to a real app, but it has some issues. The app is very slow, and the longer it's used and the more the user interacts with the app, the worse the problems seem to get. We're going to use **`Instruments`** to measure and analyze this app so we can make improvements where needed and hopefully, we'll end up with a fast app that works well.

This chapter is divided into the...

Exploring the Instruments suite

In this book's Git repository, you'll find a project named *Instrumental*. The app is still in its early stages of development, so there is plenty of work that needs to be done to improve the app. Some implementations are not ideal and the app contains some placeholders that might not make a lot of sense.

Even though the app isn't finished and the code isn't perfect, the app should be working fine, especially in the performance department. If you launch the app and click around, you'll notice that the app isn't working fine. It's actually really slow! Let's see if we can find the issue that's making the app slow.

If you dig through the source code, you'll immediately see that the app has three main screens: a table view, a collection view, and a detail view. The table view displays a list of 50 items, each of which links to an infinitely scrolling collection view with a custom layout. The layout for the `collectionview` is custom because we needed to have exactly one pixel of spacing between items, and this layout can be achieved with a reusable layout that can be used in other collections or apps....

Discovering slow code

Whenever you find that your app is slow or choppy, chances are that something in your code is taking longer than it should, especially if your memory usage appears to be within reasonable range. For instance, if your app uses less than 50 MB, memory is not likely to be an issue for you, so seeking the problem in your code makes a lot of sense.

To discover slow code, you should profile the app by either selecting **`Product`** | **`Profile`** in the toolbar of Xcode or by pressing Cmd+ *I*. To figure out what the code is doing, you need to select the **`Time Profiler`** template once **`Instruments`** asks you which template you want to use. This template measures how long certain blocks of code run.

To record a profiling session of our app, make sure that a device is connected to your Mac and make sure that it's selected as the device that your app will run on by selecting your iOS device from the list of devices and simulators in the scheme toolbar menu in Xcode. Once you've selected your device, start profiling the app. When **`Instruments`** launches, pick the **`Time Profiler`** template and hit record. Now use the app to navigate to a...

Closing memory leaks

Usually, if you navigate around in your app, it's normal to see memory usage spike a little. More view controllers on a navigation controller's stack mean that more memory will be consumed by your app. This makes sense. When you navigate back, popping the current view controller off the navigation controller's stack, you would expect the view controller to be deallocated and the memory to be freed up.

The preceding scenario is exactly how *Instrumental* should work. It's OK if we use some more memory if we're deeper in the navigation stack, but we expect the memory to be freed back up after the back button is tapped.

In the *Instrumental* app, the memory just keeps growing. It doesn't really matter if you drill deep into the navigation stack, hit back, or scroll a lot, once memory has been allocated it never seems to be deallocated. This is a problem, and we can use `Instruments` to dig into our app to look for the issue. Before we do this, though, let's have a deeper look at memory leaks, how they occur, and what the common causes are.

Understanding what a memory leak is

When your app contains a memory leak, this...

Summary

You've learned a lot about measuring your app's performance in this chapter. You've also learned how to find common issues and how to use `Instruments` to figure out what's going on behind the scenes for your app. In your day-to-day development cycle, you won't use `Instruments` or Xcode's memory debugger very often. However, familiarizing yourself with these tools can literally save you hours of debugging. It can even help you to discover memory leaks or slow code before you ship your app.

Try to audit and measure several aspects of your app while you're developing it, and you can see how the performance of certain aspects of your app improves or degrades over time. This will help you to avoid shipping an app that's full of slow code or memory leaks. However, don't go overboard with optimizing until you encounter actual problems in your app. Prematurely optimizing your code often leads to code that is hard to maintain. The fixes we applied in this chapter are pretty simple; we had a few small bugs that could be fixed in just a few lines of code. Unfortunately, fixing issues in your app aren't always this easy. Sometimes, your...

Chapter 23. Offloading Tasks with Operations and GCD

The previous chapter showed you how to use instruments to measure your code's performance. This is a vital skill in discovering slow functions or memory leaks. You saw that sometimes it's easy to fix slow code and increase performance by simply fixing a programming error. However, the fix isn't always that easy. Some code simply can't be written to be fast.

An example of this that you've already seen in this book is networking. Whenever you fetch data from the network, you do so asynchronously. If you don't make networking code asynchronous, execution of your app would halt until the network request is finished. This means that your app could freeze for a couple of seconds in bad circumstances.

Another example of slow code is loading an image from the app bundle and decoding the raw data into an image. For small images, this task shouldn't take as long as a networking request, but imagine loading a couple of larger images. This would take longer than we'd want to and it would significantly damage your scrolling performance, or render the interface unresponsive for longer than...

Writing asynchronous code

In Chapter 10, *Fetching and Displaying Data from the Network*, you had your first encounter with asynchronous code and multithreading. We didn't go into too much detail regarding multithreading and asynchronous code because the subject of threading is rather complex, and it's much more suited for a thorough discussion in this chapter.

If you're unclear on what has already been explained, feel free to skip back to Chapter 10, *Fetching and Displaying Data from the Network*, to review the information presented there. The biggest takeaway for this chapter is that networking is performed on a background thread to avoid blocking the main thread. Once a network request is done, a `callback` function is executed, which allows you to use the main thread to update the user interface.

In iOS, the main thread is arguably the most important thread you need to keep in mind. Let's see why.

Understanding threads

You've seen the term thread a couple of times now, but you never learned explored any of the details about threads. For instance, you never really learned what a thread is, or how a thread works. This section aims to...

Creating reusable tasks with Operations

We just explored `DispatchQueues` and how we can use them to schedule tasks that need to be performed on a different thread. You saw how this speeds up code and how it avoids blocking the main thread. In this section, we're going to take this all one step further. The first reason for this is because our asynchronous work would be better organized if we had an object that we could schedule for execution rather than a closure. Closures pollute code and they are much harder to reuse.

The solution to this is using an `Operation` instead of a closure. And instead of queueing everything in a dispatch queue, we should queue `Operation` instances on an `OperationQueue`. The `OperationQueue` and the `DispatchQueue` are similar, but not quite the same. An `OperationQueue` can schedule `Operations` on one or more `DispatchQueue`. This is important because of the way in which `Operations` work.

Using an `OperationQueue`, you can execute `Operations` in parallel or serially. It is also possible to specify dependencies for `Operations`. This means that we can make sure that certain `Operations` are completed before the next...

Summary

This chapter showed you that asynchronous code can be hard to understand and reason with, especially when a lot of code is running at the same time; it can be easy to lose track of what you're doing. You also learned that `Operations` can be a convenient way to reduce complexity in your application, resulting in code that is easier to read, change, and maintain. When an operation depends on multiple other `Operations` to be completed, it can be extremely convenient to use an `OperationQueue`, as it greatly reduces the complexity of the code you write.

It's been mentioned before, but if you intend to make use of `Operations` in your app, do make sure to check out *Apple's Demonstration of Advanced Operations* from *WWDC 2015*. The `Operations` are capable of far more than we've just seen, and it's strongly recommended to see how Apple uses `Operations` in order to create rock-solid apps. Once your app is covered by tests, measured with instruments, and improved with asynchronous code and `Operations`, it's probably time that you go and share your app with others. The next and final chapter of this book will show you how to set yourself up...

Chapter 24. Wrapping Up the Development Cycle and Submitting to the App Store

Possibly the most exciting part of the development cycle is getting your app out to some real-world users. The first step in doing so is usually to send out a beta version of your app so you can get some feedback and gather some real data about how your app is performing, before you submit to the App Store and release your app to the world. Once you're satisfied with the results of your beta test, you must submit your app to Apple so they can review it before your app is released to the App Store.

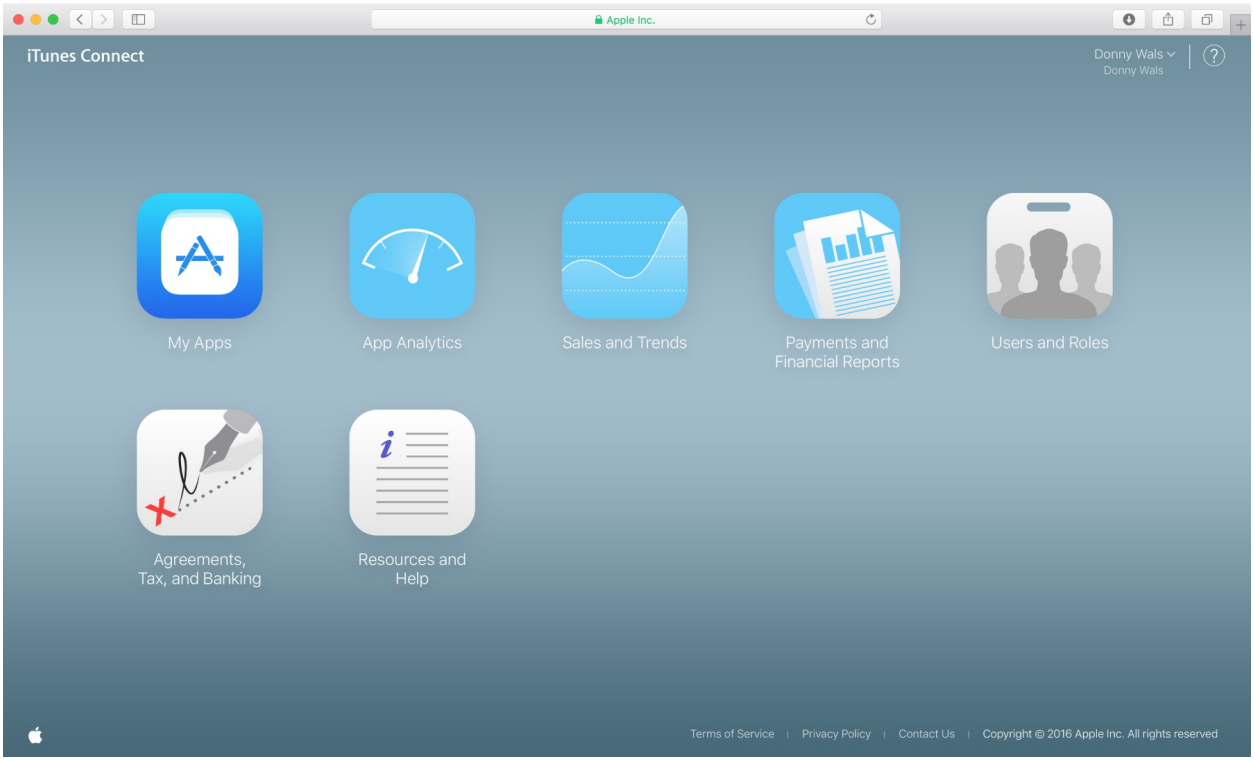
In this chapter, you'll learn everything about packing up your app and submitting it to iTunes Connect. From iTunes Connect, you can start beta testing your app and you can also submit it to Apple for review. The iTunes Connect portal is also used to manage your app's App Store description, keywords, imagery, and more. We'll cover how to properly fill everything out as well. We'll go through the following steps over the course of this chapter:

1. Adding your application to iTunes Connect.
2. Packaging and uploading your app for beta testing.
3. Preparing your app for...

Adding your application to iTunes Connect

The first thing you're going to want to do when you're gearing up to release your app is to register your app with `iTunes Connect`. To do so, you must be enrolled in the Apple Developer program. You can do this through Apple's developer portal at https://developer.apple.com. After purchasing your membership, you can log in to your `iTunes Connect` account on https://itunesconnect.apple.com using your `Apple ID`.

After logging into your `iTunes Connect` account, you are presented with a screen that has a couple of icons on it:

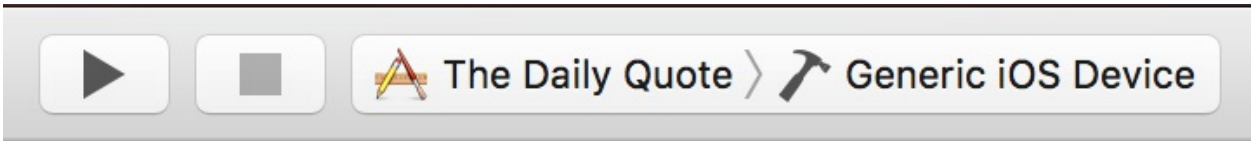


This screen is your portal to manage your App Store presence. From here, you can manage test users, track your app's downloads, track usage, and more. But most importantly, it's where you create, upload, and publish your apps to Apple's beta distribution program called `TestFlight` and to the App Store. Go ahead and peek around a bit; there won't be much to see yet but it's good to familiarize yourself with the `iTunes Connect` portal.

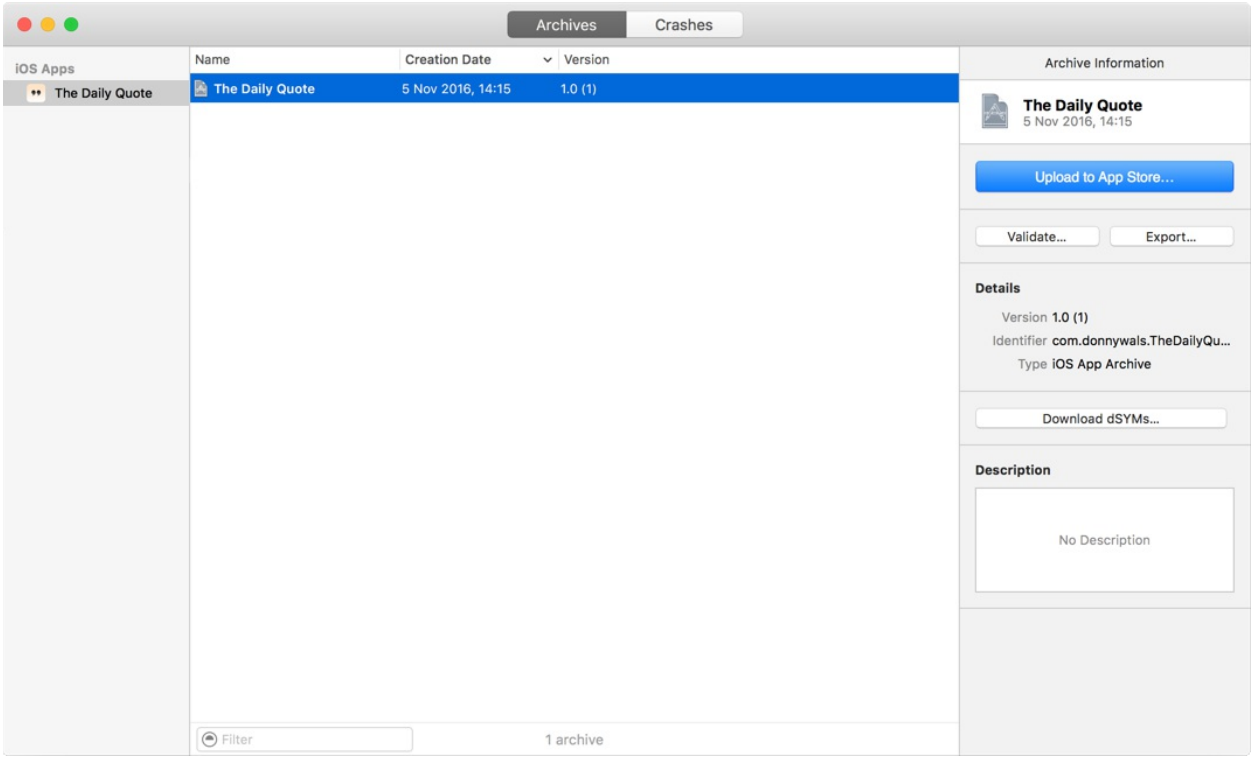
The first step in getting your app out to your users is to navigate to the `My Apps` section. Once you're inside of the `My Apps` area, you'll see all...

Packaging and uploading your app for beta testing

To send your app out to both beta testers and real users, you must first archive your app using Xcode. Archiving your app will package up all contents, code, and assets. To archive your app, you must select **`Generic iOS Device`** from the list of devices your app can run on:



With this build device selected, select **`Product`** -> **`Archive`** from the top menu of Xcode. When you do this, a build will start that's a lot slower than usual. That's because Xcode is building your app in a way that enables it to run on all iOS devices. Once the archive is created, Xcode will automatically open the organizer panel for you. In this panel, you get an overview of all apps and archives that you have created:

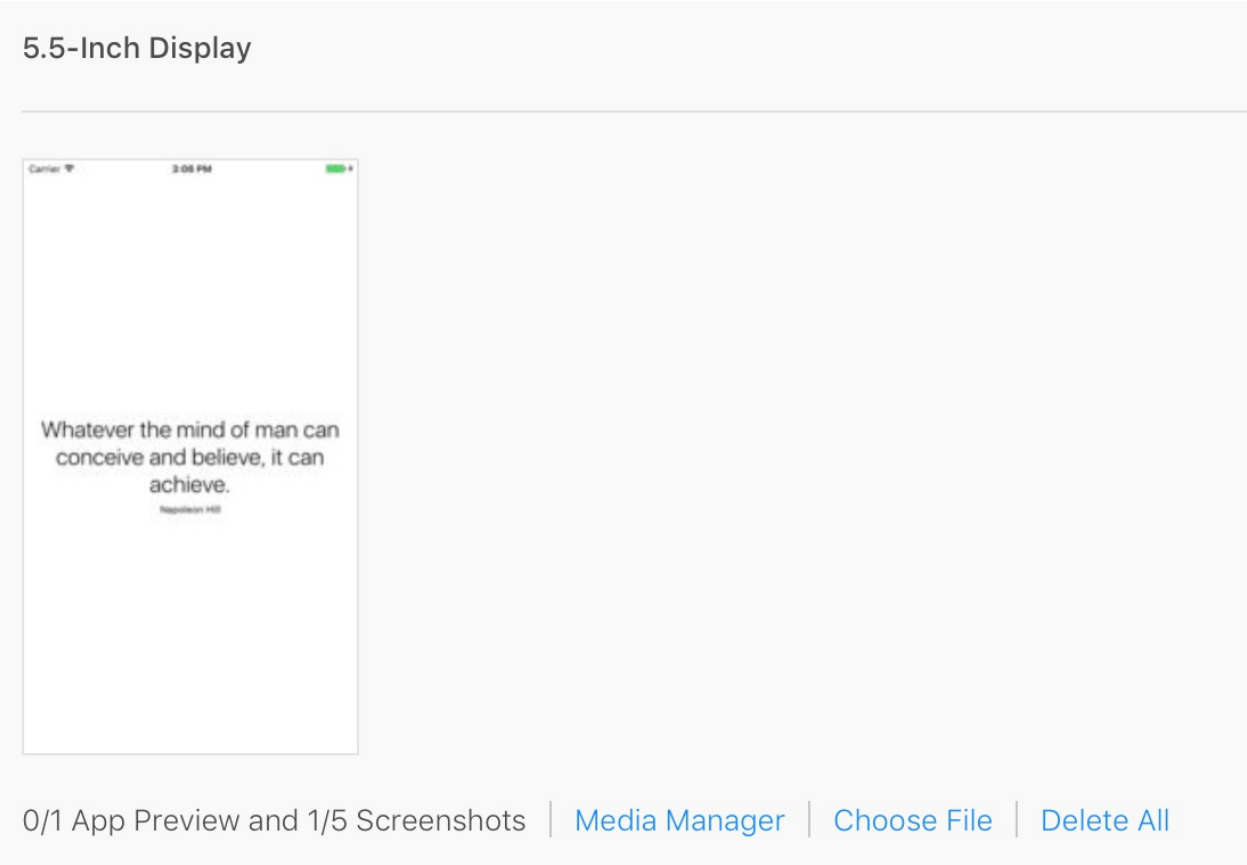


Before you archive your app, you should make sure that your app is ready to release. This means that you must add all of the required app icon assets to the `Images.xcassets` resource. If your app icon set is incomplete, your app will be rejected upon upload to **`iTunes Connect`** and you'll have to generate your archive all over again.

When you're ready to upload your build to **`iTunes Connect`** and make it...

Preparing your app for launch

Moving from beta testing to releasing your app does not require much effort. You use the exact same version of your app as you've already exported and tested with your users. In order to be able to submit your app for review by Apple, you simply have to add more information about your app and you should set up your *App Store* presence. The first thing you should do is create a couple of screenshots of your app. You add these screenshots to your *App Store* page and they should look as good as possible because potential users will use screenshots to determine whether they want to buy or download your app or not. The simplest way to create screenshots is to take them on a 5.5-inch iPhone and a 12.9-inch iPad. Doing this will allow you to use the **`Media Manager`** feature in **`iTunes Connect`**, right under the screenshot area, to upload the large-sized media and have it scale down for smaller devices:



After submitting screenshots, you should also fill out a description and keywords for your application. Make sure that your description is clear, concise, and convincing. Your keywords should be used as much as you...

Summary

This final chapter covered preparing to release your app. You learned how to archive and export your app. You saw how to upload it to `iTunes Connect` and how to distribute your app as a beta release. To wrap everything up, you saw how to submit your app for review by Apple in order to release it to the *App Store*. Releasing an app is exciting; you simply don't know how well your app will perform or if people will enjoy using it. A good beta test will help a lot, you'll be able to spot bugs or usability issues, but there's nothing like having your app in the hands of actual users.

A lot of developers invest a lot of time and effort into building their apps and you are one of them. You picked up this book and went from an iOS enthusiast to an iOS master who knows exactly how to build great apps that make use of iOS 10's new, awesome features. When you're ready to launch your own app into the *App Store*, you'll learn how exciting and nerve-racking it can be to wait for Apple to review and hopefully approve your app. Maybe you get rejected on your first try; that's possible. Don't worry too much about it, even the biggest names...