Practical Swift

Eric Downey

Apress®

Practical Swift

Eric Downey

Apress®

Practical Swift

Eric Downey
Columbus, Ohio, USA

ISBN-13 (pbk): 978-1-4842-2279-9 ISBN-13 (electronic): 978-1-4842-2280-5
DOI 10.1007/978-1-4842-2280-5

Library of Congress Control Number: 2016960664

Managing Director: Welmoed Spahr
Lead Editor: Aaron Black
Technical Reviewer: Felipe Laso-Marsetti
Editorial Board: Steve Anglin, Pramila Balan, Louise Corrigan, James DeWolf, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Jessica Vakili
Copy Editor: Rebecca Rider
Compositor: SPi Global
Indexer: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

Printed on acid-free paper

There are a lot of people that I can dedicate this book to. Many people have helped me and pushed me and made this possible, but one person sticks out. I would not have been capable of writing this book without her. I want to dedicate this book to Grace Tay.

Contents at a Glance

Contents

About the Author

Eric Downey an iOS developer from Columbus, Ohio employed at Information Control Company, the largest Ohio-owned IT consulting company. He coordinates the Tech Sig for ICC, monthly meetings focused on a variety of tech talks. As a consultant for ICC, he has spent the last three years working on Java, Javascript, and iOS applications. He has been in iOS development for 6 years, specializing in Swift for the last two years. Eric hold a Bachelor's degree in Computer Science with a minor in Art from Capital University.

About the Technical Reviewer

Felipe Laso-Marsetti is a Senior Systems Engineer working at Lextech Global Services. He's also an aspiring game designer/programmer. You can follow him on Twitter as `@iFeliLM` or on his blog.

Acknowledgments

First, everyone over at ***Apress***, you're awesome! I had a great time writing this book and I wanted to thank you for the opportunity. In particular, Jessica Vakili, James Markham, Steve Anglin, Mark Powers, and Matthew Moodie. Next, Scott Preston, thank you for forcing me to write this book. Also, thank you for putting me in touch with ***Apress*** and for helping me achieve level 50. To the original ICC team (Holden and Bobby) and my current team thank you for challenging me and pushing me to do better. Let me also acknowledge Amber Fiore and Steve Leppert for helping me strengthen my career with ICC.

I also want to take a moment to thank my parents, as well as my brother, Matt, and Aunt Norma. You've always supported me and this book wouldn't be possible without you. Thank you to the local Starbucks, your caffeine kept me awake long enough to write this book. Last, but definitely not least, I wanted to thank my professor, Dr. David Reed. You helped me get my start in iOS and set me on this path, so thank you. There were a bunch of others I wanted to thank, but I'm not going to because it would take too long. You know who you are, so thank you.

Introduction

Welcome to *Practical Swift*. I'm Eric Downey and I have been developing iOS applications for the last six years. When Swift was released over 2 years ago, I dedicated all of my time and energy to learning Swift and its nuances. Starting off, you might have these questions about this book:

Who should read this book?

What am I going to learn?

What sets this book apart?

That's easy! You should read this book if you have a working knowledge of Swift. This book assumes familiarity with the standard iOS frameworks and general programming knowledge. You should read this book if you are an iOS developer looking to increase your skills. You are going to learn the new features and API of Swift 3. You will develop a well-architected iOS app in Swift 3 and the new Xcode.

Why Write This Book?

This book is about Swift 3.0 and iOS. I will explain ideas and concepts that exist in iOS and Swift. I will show how I would write code and provide justifications for my decisions. Then I want you, the reader, to decide for yourself how you want to write your code using your newfound techniques and concepts.

My goal is not to tell you how to write Swift code, but give you the knowledge you need to feel comfortable deciding how to write your own code. I want to structure the book in this way because I have seen too many books and articles that describe how to do a particular task. They give you a list and explain what task each piece accomplishes. However, they never teach you why or how it could change, depending on the circumstances. Software development is an art form and there are many ways to structure the same solution.

Book Format

Each chapter will be broken down as sections. The first section in each chapter will have an overview of the material that will be covered. It will highlight each section and give an overview of what you will learn in the chapter. The end of each chapter will then contain a recap of the material. This section will go over the information that will be presented in each chapter. We will also talk about the playground reference guide we are going to create. Let's start with the walkthrough of the chapters.

Chapter Walkthrough

This section is going to cover each chapter. Chapters have been grouped based on their information. Some chapters feed into others, while some are standalone. Each chapter is also associated with a half of the book. Section I is the first half of the book. This section is the theory and reference building section. We will discuss concepts and techniques. The second half, Section II, will be applying Section I in building our app. Let's get started with Section One and Chapter One.

Section I

Section I is going to include Chapters One through Eight. I will explain key concepts and ideas using Swift 3.0. This book expects the reader to have a working knowledge of iOS and Swift. Using an Xcode playground, I want you to follow along, creating a reference book of your own. This reference will be key in Section II. The following is a walkthrough of the first eight chapters and the information they will contain.

Chapter 1

Let's start at the beginning, the ***Evolution of Swift***. This is probably one of the most, if not the most, important chapter, as it sets the stage for everything. The first chapter is, in part, a look back at the previous versions of Swift. It will walk through the release of Swift 1 and what the new language offered us as iOS developers (I was just starting to get good at Objective-C). We will then jump right into Swift 2 and all the cool new features. The biggest addition and my favorite, Protocol Oriented Programming, was added in this release. If this is not apparent, look at how many times protocols show themselves throughout this book. If I counted right, we will discuss protocols in Chapters Five, Seven, Ten, Eleven, and Twelve. ☺

Then, by far the most important section in Chapter One, is *Swift 3*. We will look at some of the biggest changes in Swift 3. One of the biggest changes in *Swift 3* is the introduction of the API design guidelines. I will try to follow the guidelines as best I can. There will always be room for improvement, but I hope you can look at each and every method, protocol, class, etc. and know exactly what it is meant to do given the grammar used. Chapter One also discussed other major proposals that are going live with Swift 3. We will go into detail in Chapter One.

Chapters 2

Chapters Two and Three, are somewhat external to Swift 3. I included them because they are important topics. We should know what features are available to use in our IDE, Xcode. We will examine previous features in Xcode such as the UI Debugger and Code Coverage as well as the brand new Xcode Source Editor Extensions. Given that we can now extend our IDE, this opens the door for a much better development experience. I know I am going to enjoy all the emojification extensions and making life harder on my coworkers. Chapter Two will also discuss the Address Sanitizer, Thread Sanitizer, and the Memory Graph Debugger. The Memory Graph Debugger is a very powerful tool and I am really looking forward to using it in Xcode 8.

Chapter 3

Chapter Three explains the idea of Swift modules. Swift modules include our apps and dynamic frameworks. Dynamic frameworks make creating third party libraries super easy, which just makes them more important to understand as more can show up our projects. We will also touch on some package managers available, such as Cocoapods, Carthage, and the official Swift Package Manager.

The official Swift Package Manager is where I am very hopeful. The more widely adopted this package manager becomes, the greater the potential for Swift to run anywhere. Since Swift was open sourced, I have been very hopeful to see it pop up in other areas of development, such as game development. The Swift Package Manager and Swift being open source make this a possibility. In Chapter Three we are going to build a basic package of our own.

Chapters 4

Chapter Four is all about architecture. We will examine several design patterns including MVC, MVVM, Presenter, and Dependency Injection. Design patterns are key to software development in general. Everything has its own flavor and iOS is no different. We will only briefly touch on MVC, the main design pattern in iOS. The two main patterns we will go into detail about are MVVM and Dependency Injection. These are the two main patterns we are going to use when building our app in Section II. Chapter Four will also discuss Anti-Patterns. Design Patterns can make our code extensible and allow it to scale. Anti-Patterns are the opposite. We will examine how certain Anti-Patterns can ruin our code bases and how to avoid them. Chapter Four will then feed directly into Chapter 5, Protocol Oriented Programming.

Chapter 5

Protocol Oriented Programming (POP) is a really awesome topic. It can solve a lot of different problems, but it requires a slightly different way of thinking. Chapter Five is going to prepare us to use POP in Section II and our app. POP was introduced in Swift 2 and we will touch on it in Chapter One when discussing the evolution. POP can describe architecture, which is why Chapter Four is so important, but more on that later. Understanding how to effectively use POP can also avoid Anti-Patterns.

Chapter 6

Chapter Six is all about Swift generics. It walks through how generics allow for some pretty awesome functional paradigms in Swift. This is so important to understand as these techniques show up more and more in Swift. Then, it's right back to protocols with associated types. Associated types are really the last piece of Protocol Oriented Programming. We can keep the type information in a particular protocol and use this to our advantage in our extensions.

Chapter 7

This is going to be a heavy chapter. There are a lot of figures. We will discuss Auto Layout, constraints, the new device configurations, and trait variations. The trait variations are by far my favorite part of this chapter. We will see how to build a completely different interface for landscape versus portrait. We can see how to activate and deactivate constraints and properties based on different traits. And how could I resist throwing in a protocol. ☺ As the iOS ecosystem expands and we see more devices, the more we are going to have to do as developers to make sure our apps run and display correctly on each device. My hope is this chapter will go a long way in helping you understand how to achieve this goal.

Chapter 8

This chapter is going to be all about testing. The previous chapters will take you through techniques and concepts that can make coding super simple and quick. This chapter is going to discuss how to rein all of these concepts in, so we can write high quality code that is also testable. We will build a small class that has been influenced by another project on Github called MockFive. We will then use this in our tests for our app in Section II. We will build a package in Chapter Three and in Chapter Eight, we will revisit this package and write tests for it.

The last topic we will touch on in Chapter Eight will be Test Driven Development (TDD). We will examine what TDD is and how it works. Testing in iOS is still relatively new and TDD even more so. It is important we talk about this topic and how it relates to iOS so we can effectively write code and tests. We will discuss the benefits and drawbacks.

Section II

This half of the book will be about building our app. The app will be a simple grocery list app. We will be able to create lists and items. The items are then associated with a specific list. The purpose of this simple idea is to keep the focus on the techniques and the architecture. We will have a clean architecture that will be tested. You can then take these skills to any project. Here is a walkthrough for these chapters.

Chapter 9

This is the first chapter where we start our grocery list app. Chapter 9 is all about ***Interface Builder*** and building the app's UI. We are going to build this up first, so we do not have to switch between Interface Builder and our code. This is also how I like to build apps. I like to

start with the interface, so I can plan out how the pieces will communicate and which pieces need to communicate. Even though we build our UI first, this does not mean we cannot change it later.

This chapter is going to use code and discussion about Auto Layout from Chapter Seven. In Chapter Seven we will build UI and constraints that we will use in our app. We are also going to take our code and pull it into the app as well. This will be a fun chapter as we build our UI and figure out how our app is going to look and feel.

Chapter 10

Chapter Ten is where we start coding our grocery list app. We will implement the MVVM pattern in this chapter. The grocery app is going to use *Core Data*, but we will not start by adding any support for *Core Data*. This chapter is going to create a placeholder, so we can get going and not have to deal with Core Data here. We then use code from Chapter Four and use the MVVM pattern to implement the two main pages in our app. We are also going to use Dependency Injection here as well. Dependency Injection will become more clear after Chapter Four but it will be used everywhere.

Chapter 11

Once we finish implementing half of our app, we will take a step back and stop using the hardcoded data that was introduced in the Chapter Ten. This chapter is all about *Core Data. Core Data* is a big topic and I have dedicated this entire chapter to discuss the topic and how to implement it properly. We are going to rely on Protocol Oriented Programming and Dependency Injection heavily to implement our interactions with *Core Data*. We will not advance our grocery list app very much, but this chapter allows our app to use real data. This brings us to the final chapter.

Chapter 12

In the last chapter, we are going to finish building our grocery list app. We are going to examine all the loose ends we have left and we are going to finish them. We are also going to take a look back at the app as a final reflection. It is going to be important to reflect on our app, so we can bring this book together.

Playground Reference

The playground reference is the biggest take away from this book. I will be showing code samples and I want you to use a Swift playground to hold onto all of this information. I then ask you to keep using this playground once you finish this book. I have found it invaluable to keep a running playground of all my ideas and issues I've found. The purpose of the playground is that you use it in your day-to-day development.

I will explain how to effectively use the playground in Chapter Four. Swift playgrounds have a cool markup system so you can document your code. I will document the code samples as I go, but feel free to add more. We are also going to use code from our playground in Section II when we build our app. All right, now that you have a good idea of what each chapter is going to walk through, let's get started with Chapter One, the ***Evolution of Swift***.

Required Materials

Here are the required materials to follow along with this book:

- MacOS El Capitan (or a later version)
- Xcode 8
- Swift 3.0
- iPhone (highly recommended, but not required)

Part I

Building the Reference Guide

Chapter 1

Evolution of Swift

This chapter will walk through all three iterations of Swift and how Swift has evolved over the last two years. This chapter will take a look back at the features of Swift and the core concepts. We will then get to see how these features have evolved over the short span of time Swift has been alive. Then once we reach Swift 3, we will get to see how the newest features of Swift have modernized the language even more. By seeing the influences and evolution of the language, we can see where the language is going and its potential future.

What You'll Learn

In section **Swift 1**, we will review the transition from Objective-C to Swift and discover why Swift was created. In section **Swift 2**, we will review the new `Guard` and `Defer` keywords, multiple optional binding, error handling, protocol extensions, and availability checking. In section **Swift 3**, we will examine a selection of accepted proposals that were implemented in Swift 3. Section **Swift 3** will showcase the brand new API design guidelines. Apple implemented and released these new guidelines to set the standard for Swift 3 and all versions beyond. We are not going to examine all of the implemented proposals in Swift 3, but I have selected a few for us to focus on. Let's get started with Swift 1!

Swift 1

The day was June 2, 2014. I was hard at work when WWDC hit. Everything seemed pretty normal until they announced Swift 1. Once I saw Swift in action, I was hooked. So hooked, I stopped working and downloaded the Xcode 6 beta. I had been using Objective-C for four years at this point, and I was tired of all the square braces. *I know I am not the only one.* This is just one of the many differences between the Swift and Objective-C. This section will discuss the transition from Objective-C to Swift and why Swift was created in the first place.

Electronic supplementary material The online version of this chapter (doi: 10.1007/978-1-4842-2280-5_1) contains supplementary material, which is available to authorized users.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_1

Transition from Objective-C to Swift

Swift 1 was just released, now what? iOS developers all over the world have mountains of Objective-C code and there are hundreds of libraries on GitHub. How do we transition a community such as this to a brand new language that is so different? Well, Apple has made this as simple and seamless as possible. Objective-C projects can contain Swift code and you can also use Objective-C APIs in Swift. The interoperability of these two languages makes the transition from Objective-C to Swift very simple for Apple and developers.

I believe the biggest transition pain into Swift 1 is the introduction of *optional values*. I know this caught me off guard at first. In Objective-C I would always have to check for `nil` and this was just a normal exercise that I was used to. This is also the case for many other languages, such as Java. Swift, however, can guarantee a value to be `non-nil` if it is non-optional. This allows you to keep the `nil` checks to a minimum. It also creates safer code because it forces you to think about `nil` where it can happen. When anything can be `nil`, it is easy to forget to check, and that creates a potential crash in your application.

Another huge difference is the functional style Swift brings. Swift 1 brings *closures* to your everyday iOS toolbox. Objective-C did contain blocks, but these were an addition, whereas Swift has closures built in from the start. Closures can serve many purposes, such as functional programming, handling asynchronous tasks, and much more. Closures can clean up your code and provide a mechanism for keeping less state, but they also bring the opportunity to leak memory, so be careful.

Swift 1 does have its drawbacks. It is a very young language and as it develops, it will bring breaking changes as it solidifies. These reasons, however, should not hold you back. Transitioning to a new language can be difficult. Apple has made it very simple and there is no reason you should not use Swift.

Why Was Swift Created?

That's simple; Swift was created to replace Objective-C. This seems like a daunting task. macOS and iOS are both written in Objective-C and these are two of the biggest platforms in the world. According to Apple

> *Swift is a powerful and intuitive programming language.… Writing Swift code is interactive and fun, the syntax is concise yet expressive.… Swift code is safe by design, yet also produces software that runs lightning-fast.*
>
> —Apple Inc.

Swift was also created to be simple. It is easy to start writing Swift and it allows less-experienced developers to easily create iOS apps. I started iOS when I was in college and I had very little experience in C at the time. This made it harder for me to make iOS apps, as Objective-C has a lot of pitfalls. Swift can also appeal to even younger audiences, allowing kids to get started with programming. Swift was created for many reasons and it has the potential to run anywhere. In my opinion, this tells me that the reason Swift was created was not only to replace Objective-C, but many languages and many systems that currently exist. Swift for all!

Swift 2

Swift 2 witnessed some of the most innovative changes to Swift and the Foundation Framework. The changes not only altered the language and the Foundation Framework, but also how developers think about problems. One of these many changes is called protocol-oriented programming, discussed in depth in Chapter 5. This section will focus on the changes between Swift 1 and 2 and the brand-new features of Swift 2. Swift 2 will discuss the new `guard` and `defer` keywords, multiple optional binding, exception-style error handling, protocol extensions, and availability checking.

Guard and Defer

Swift 2 offers new keywords. We will focus on `guard` and `defer`. These two new keywords give us new functionality to keep our Swift code more legible with fewer lines of code. Let's start with the new `guard` keyword.

Guard

Think of `guard` as your basic `if` statement, allowing for control flow. Traditionally, an `if` statement branches when a condition evaluates to true. The `guard` keyword, however, branches if a condition evaluates to false. The `guard` also disallows the execution of the branched code, as well as the code beyond the `guard` statement. This means the inner block of the `guard` must `return` or `throw`. The following code shows an example the new `guard` keyword:

```
let condition = false
guard condition else { return nil } // condition is false
// condition is true
```

You can see this is very useful in making your code more legible with fewer lines of code. You now might be wondering, does this work with optional binding? Absolutely, you can `guard let` any optional value as if it were an `if let` block. This leads us into the next feature Swift 2 added, multiple optional bindings.

Multiple optional bindings is now available in any `if` or `guard` statement. In Swift 1, writing multiple `if lets` was tedious and created illegible code due to multiple nested `if let` statements. Swift 2 introduced a comma-separated list of optional bindings in one `if/guard` statement. Let's see a comparison:

```
// Swift 1
let optionalValue1: String? = someString()
let optionalValue2: String? = someOtherString()
if let value1 = optionalValue1 {
    if let value2 = optionalValue2 {
        value1
        value2
    }
}
```

```
// Swift 2
let optionalValue1: String? = someString()
let optionalValue2: String? = someOtherString()
if let value1 = optionalValue1,
    let value2 = optionalValue2 {
    value1
    value2
}
```

The previous code shows how to use multiple optional bindings in an `if` statement. The syntax for a `guard` statement is exactly the same. You can see how this can really clean up our code. We can avoid all the unnecessary nesting of `guard`s or `if` statements and just comma-separate our bindings.

Defer

The next keyword introduced in Swift 2 is `defer`. Using `defer` forces any code within the statement block to be run at the end of the current scope. Multiple `defer` blocks can be declared within a single scope; however, please know the `defer` blocks will be run in reverse order of appearance. `Defer` blocks are run in reverse order to allow you to refer to resources that are used/cleaned up in early `defer` blocks. The following code shows how `defer` holds the execution of the block until the end of the function:

```
func postFixAdd(inout x: Int) -> Int {
    defer {
        x += 1
    }
    return x
}

var x = 1
postFixAdd(&x)  // Value of x is 1

x  // Value of x is 2
```

This can be a bit of an obscure concept. I have found there are two key use cases for this action. First, I believe it is useful when you want to communicate to others that an action has to be at the end of a scope. If you were to write code where the order matters and someone else comes in, they could change the order. This might break something without anyone knowing. Using `defer` makes this more explicit.

The second use case I have found is around cyclomatic complexity. You might run into a situation where you need to run code at the end of your scope, but there are multiple branches within the current scope. Instead of reworking your branches or adding more, you can use `defer` within the correct branch. Then it will execute at the end of your scope, only when the specific branch is executed. Pretty cool, huh? Up next, we will take a look at the first new major feature introduced in Swift 2, exception-style error handling.

Error Handling

Swift 1 did not have an exception-handling model. The only pattern available for handling errors was injecting an `NSError` object that would either be nil or non-nil, signaling an error occurred. This is also the pattern that has been used for years in Objective-C. Swift 2 fixed this issue. Apple added an exception-handling model using the keywords `try`, `throw(s)`, and `catch`. This new pattern now works perfectly through the use of protocols. The `ErrorType` protocol is the "error" a catch block will catch and `NSError` conforms to this protocol. If you have not heard about the power of protocols and their use in Swift, the next section, as well as Chapter 5, will be going in-depth on protocols and protocol-oriented programming.

Let's now look at an example of how exception handling works in Swift. You may follow along in a Playground; however, this syntax will be in Swift 2 and may need to be converted if you are using Swift 3. This example has a simple function that converts an optional string to a non-optional string. First, let's create our "catchable" exception:

```
enum StringParsingException: ErrorType {
    case NotThere
    case Empty
}
```

You can see our exception is just an `enum` with two cases. The `enum` conforms to the `ErrorType` protocol and since `ErrorType` is just a protocol, anything can conform to this protocol, which makes it very easy to throw exceptions. The first case is `NotThere` and the second is `Empty`. These are the two cases we want to guard against in our function and throw an exception if either of these conditions is met. Now, let's look at how we can use this `StringParsingException enum` in our function.

```
func throwStringError(str: String?) throws -> String {
    guard let s = str else { throw StringParsingException.NotThere }
    if s.isEmpty { throw StringParsingException.Empty }
    return s
}
```

First, we have to mark this function with the keyword `throws` in the function signature. This tells the Swift compiler our function could generate an exception. Now every time you make a call to this function without using the keyword `try`, Xcode will generate this error: "Call can `throw` but is not marked with `try`." The following is how you wrap this function to properly catch the error:

```
do {
    let result = try throwStringError(nil)

    result
}
catch  {
    error
}
```

This first block of code will throw a `NotThere` exception and will be caught by the `catch` block.

```
do {
    let result = try throwStringError("")

    result
}
catch  {
    error
}
```

This previous block will throw an `Empty` exception and will also be caught by the `catch` block. Finally, this last block will not throw an exception and will continue execution.

```
var str = "Hello Exception"


do {
    let result = try throwStringError(str) // str = "Hello Exception"

    result
}
catch  {
    error
}
```

Pretty easy, right? This mechanism is very similar to the exception handling in Java. First, you have to mark a function that can throw with the keyword `throws`**.** Then every subsequent call to this function must be marked with `try`. Now, it would not be Swift without the ? or ! characters. You can write `try?`**,** which will silence the exception and return `nil` if the function does throw an exception. It will return the value of the function if it does not throw an exception. However, using ? will effectively create an optional value even if the function does not return an optional. You can also use `try!`**,** which forces the function to be called without the need for a corresponding `catch` block.

> **Caution** Be warned, using `try!` will cause your application to crash if the function does `throw` an exception.

I know that was a lot to take in, especially for a subsection in the first chapter. This is a very important concept because it will affect the code we are going to write throughout this book. Error handling is important when we build our own frameworks and other people can use our code. This is just a quick introduction to the error handling that is now available in Swift and there are more things you can do with the `try` and `catch` blocks that I will not go into here. Up next, we are going to talk about protocol extensions, which is my favorite feature within the Swift language.

Protocol Extensions

As Swift developers, we use protocols every day. From `UITableViewDataSource` to `NSCoding`, protocols are everywhere in the Objective-C/Swift ecosystem. Protocols are similar to interfaces (Java), and prior to Swift 2, a protocol only offered

> *a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.*
>
> —Apple Inc., The Swift Programming Language

As of Swift 2, protocols have taken on a more `trait` or `mixin` style. You can now extend protocols using the keyword `extension` to give a particular protocol a default implementation or other functionality that should belong to that protocol. The implications of this are huge. Allowing protocols to have an implementation, we can essentially achieve multiple inheritance. A Swift object can conform to as many protocols as it wants. If each of those has some implementation, then our object would have all of that code. If that does not hurt your brain, it probably should.

We are also no longer bound to just object-oriented programming. Objected-oriented programming has its own drawbacks, and protocols can help us get around these issues. I am not going to show any code in this section. I will leave that to Chapter 5. The next feature added in Swift 2 is a small one, but it does offer some noteworthy functionality.

Availability Checking

Who out there has seen the following block of code?

```
if NSClassFromString("<ClassName>") != nil {
    // <ClassName> exists
}
else {
    // <ClassName> does not exist
}
```

A new version of iOS is released every year. Keeping up with the API changes, additions, and deprecations can be a headache if you have to maintain an app for multiple versions of the operating system. Swift 2 is now making this a little easier. Swift 2 added the ability to use `#available` to check for the current OS version number. The following is a code snippet showing how to use `#available`.

```
var constraints: [NSLayoutConstraint] = []


if #available(iOS 9.0, *) {
    let top = view1.topAnchor.constraintEqualToAnchor(view2.topAnchor)
    let bottom = view1.bottomAnchor.constraintEqualToAnchor(view2.bottomAnchor)
    let lead = view1.leadingAnchor.constraintEqualToAnchor(view2.leadingAnchor)
    let trail = view1.trailingAnchor.constraintEqualToAnchor(view2.trailingAnchor)
    constraints += [top, bottom, lead, trailing]
}
```

```
else {
    let top = NSLayoutConstraint(item: view1, ... constant: 0)
    let bottom = NSLayoutConstraint(item: view1, ... constant: 0)
    let lead = NSLayoutConstraint(item: view1, ... constant: 0)
    let trail = NSLayoutConstraint(item: view1, ... constant: 0)
    constraints += [top, bottom, lead, trail]
}

constraints // Contains the 4 constraints
```

In iOS 9.0, Swift added anchors for programmatically working with layout constraints. This makes it easy to generate constraints in code; however, if your app supports iOS 8.0 and above, you will not be capable of using this new feature. With `#available` you can use an `if` statement to check the current iOS version and use these anchors if the iOS version is above 9.0. The `else` statement can then take care of the backward compatibility. Swift does not stop there; it allows you to add `#available` to functions, which means that function can only be called if the OS is above the specified version. This is a great way to keep your app backward compatible with earlier versions of iOS or macOS, but still use the new APIs in a cleaner way. Then, when you are ready to drop support for an old OS, all you would need to do is get rid of your `#available` if statements.

This wraps it up for Swift 2. We went through a lot of the new features and what is available to us, such as a new exception-style error handling, protocol extensions, and achieving backward compatibility through availability checking**.** The next section is going to cover the major changes in Swift 3. I hope everyone is excited and ready!

Swift 3

Now, what you have all been waiting for, Swift 3 is making breaking changes. Although Swift 2 brought some of the biggest language updates, Swift 3 brings the biggest API updates. This release marks the first major release since Swift went open source. This means that not only has Apple been hard at work updating Swift, the community has as well. All implemented proposals can be found at the following link: `https://github.com/apple/swift-evolution`. This section will go over a few of the accepted proposals and the reasons these changes are being introduced.

> *The primary goal of this release is to solidify and mature the Swift language and development experience.*
>
> —Apple Inc., Swift Evolution

Argument Labels

Argument labels are one of the first most notable changes of Swift 3. The goal of this change is to normalize all parameter labels across method invocations and initializers. This change is based on proposal SE-0046. The authors of this proposal are Jake Carter and Erica Sadun. The entire proposal can be found at the following link: https://github.com/apple/swift-evolution/blob/master/proposals/0046-first-label.md.

In previous versions of Swift, the first parameter in a method or function declaration does not appear in the corresponding invocation. An extra label must be specified to include it for the first parameter. However, all parameters following the first label are included. Swift 3 is changing this behavior.

Now, all parameters are listed in the signature unless otherwise specified. The first block of code illustrates a method or function declaration for Swift 2 and the second block illustrates the new Swift 3 conventions.

```
// Swift 2
func swift2(arg1: Int, arg2: Int, arg3: Int) -> Int {
    return arg1 + arg2 + arg3
}
swift2(1, arg2: 2, arg3: 3)

// Swift 3
func swift3(arg1: Int, arg2: Int, arg3: Int) -> Int {
    return arg1 + arg2 + arg3
}
swift3(arg1: 1, arg2: 2, arg3: 3)
```

You can see how the first parameter is now used in the invocation. You might remember when Swift 2 was released; first parameter labels were omitted by default. This behavior comes from an old Objective-C style. One of the reasons this change has been introduced is to create a Swift standard, as opposed to using an old style from Objective-C. This change also ensures a consistent behavior among initializers, methods, and functions. It is possible to omit the first parameter label in Swift 3. You must specify an underscore preceding the first argument. In the next section, we will be discussing multiple proposals, but they are all related to the new API Design Guidelines released by Apple.

API Design Guidelines

Apple is making big changes to the way they design and write their APIs. There are three separate Swift proposals related to the new API Design Guidelines. These proposals are SE-0023, SE-0006, and SE-0005. SE-0023 is the umbrella proposal to implement the new guidelines. SE-0006 and SE-0005 are specifically applying these guidelines to the *Standard Library* and ensuring the guidelines can be met when translating Objective-C APIs into Swift, respectively. Before we talk about each of these proposals, let's examine the guidelines themselves.

The guidelines can be found at https://swift.org/documentation/api-design-guidelines/. There is too much content on the guidelines to reasonably fit in this section. I will be going through the CliffsNotes version here to give you an overview, and we will be adhering to these guidelines throughout this book. We want to enforce these guidelines as we build our playground reference and our app to ensure the code is easily recognizable as Swift. I highly recommend reading through the swift.org documentation on the new guidelines. There is a lot of good information there and as Swift developers, we should all be adhering to the official guidelines. Here are the fundamentals of the new API Design Guidelines:

- Clarity at the point of use
- Clarity is more important than brevity
- Write a documentation comment

—Apple Inc., API Design Guidelines

Let's examine what each of these mean in terms of code. The following code is going to be from our grocery list app. First is "*clarity at the point of use*." The following code is from some class that is used to retrieve grocery lists from some backend.

```
func getLists() -> [GroceryList] {
    ...
}
```

There is nothing technically wrong with this method. However, the language is very ambiguous. You can imagine this being used in `viewDidLoad` or `viewWillAppear` to retrieve the grocery lists, but you cannot be sure. This could be retrieving some other type of list. What if the app has two different types of lists? Obviously, this is not clear. To disambiguate this method, let's change the API to look more like the following:

```
func retrieveGroceryLists() -> [GroceryList] {
    ...
}
```

Now, this is a lot clearer and gives us insight into the intent of this method. The next fundamental is "*clarity is more important than brevity*," which means, if you need to write a full sentence to get your point across, do it. Okay, maybe not a full sentence. The following code illustrates brevity, but not clarity.

```
func loadList(tableView: UITableView, list: [GroceryList], complete: Void -> Void) {
    ...
}
loadList(UITableView(), list: [], complete: {})
```

This method takes in three parameters, a `UITableView`, an array of `GroceryList` objects, and a `completion` block. This code is brief and saves the user from writing a lot of code. However, looking at this line without documentation, it would be very difficult to discern the meaning. Let's take a look at the same function written in a *Swiftier* way.

```
func animateReload(onTableView tableView: UITableView, withGroceryLists list: [GroceryList],
andCompletion completion: Void -> Void) {
    ...
}
animateReload(onTableView: UITableView(), withGroceryLists: [], andCompletion: {})
```

There is a lot more code here, but it is very clear about its purpose and the operations it performs. Just from the method signature, we can tell; this method is going to reload the `UITableView` with the list of `GroceryList` objects we give it, and then it is going to run a `completion` block when everything is finished animating.

The next and last fundamental of the API Design Guidelines is "*write a documentation comment*." If you were to give a documentation comment to every method, function, class, struct, enum, and so on, nothing would be ambiguous. Now, we are all human. We are not going to write documentation for every single one of these items every time we code. If we try to follow the fundamentals, even if we are not perfect every time, our code will become less ambiguous and will be self-documenting. Self-documenting code makes using our own code easy and makes using other developer's code a dream. Also, the more self-documenting our code is, the more we might be able to get away with fewer documentation comments. ☺

In the last code example, I included another part of the API Design Guidelines. The method starts with `animateReload`. The other part of the guidelines is to give methods and functions names that correspond to their side effects. The guidelines state this:

> *"Those without side-effects should read as noun phrases" and "Those with side-effects should read as imperative verb phrases."*
>
> —Apple Inc., API Design Guidelines

We renamed our method to use two verbs as the base: `animate` and `reload`. These both speak to the side effects of this method. This method will perform an animation on the view and it will reload our table view. I wanted to point this out because I believe it is a very important piece to new guidelines, and it can be confusing to start thinking and writing code in this way.

These are the very basics of the new API Design Guidelines. There is a lot more to them and all the guidelines and conventions are discussed in length at the link provided at the beginning of this section. I would highly recommend reading through the article to completely familiarize yourself with the guidelines. Next, we are going to look at all three proposals that were mentioned earlier in this section.

Swift 3 Proposals

As I stated in the previous section, the three proposals we are going to examine in this section are SE-0006, SE-0033, and SE-0005. These proposals encompass the work necessary to apply the API Design Guidelines to the Swift language and frameworks. We are first going to look at SE-0006 where the Standard Library is being overhauled.

SE-0006

SE-0006 is the proposal to apply the new API Design Guidelines to the *Standard Library*. This proposal was authored by Dave Abrahams, Dmitri Gribenko, and Maxim Moiseev. The full proposal can be found at https://github.com/apple/swift-evolution/blob/master/proposals/0006-apply-api-guidelines-to-the-standard-library.md.

There are too many changes to discuss here. This includes such things as renaming `Generator` to `Iterator`, removing the `Type` suffix from protocol names such as `BooleanType` and `CollectionType`, and one of the most notable changes, `String` methods that mirror their `NSString` counterpart have been renamed. The following code is the method to convert the `string` to an `NSData` object:

```
// Swift 2
public func dataUsingEncoding(encoding: NSStringEncoding, allowLossyConversion:
Bool = false) -> NSData?

// Swift 3
public func data(usingEncoding encoding:  String.Encoding, allowLossyConversion:
Bool = false) -> Data?
```

This new API fits perfectly in line with the guidelines. The method returns an object and has no side effects on the original string; therefore, a noun phrase is used. You can also see in the signature that that the prefix `NS` has been removed from the return value. In Swift, the use of modules avoids potential naming collisions and thus the prefix `NS` has been removed. We will be discussing modules at length in Chapter 3.

The encoding parameter in the new API is also a little different. It is of type `String.Encoding`. The old Objective-C API asked for the encoding as a normal `String`. Now, in Swift 3, the Objective-C constants are imported into Swift code in a much safer way. The last part of this section is going to discuss two proposals: SE-0005 and SE-0033. The proposals are better translation of Objective-C APIs into Swift and importing Objective-C constants as Swift types. Also, the encoding is now a sub struct within `String`. Now, let's examine these two proposals starting with SE-0033.

SE-0033

SE-0033, importing Objective-C constants as Swift types. This proposal was created by Jeff Kelley. Many constants in Objective-C are in the form of `strings` and this does not lend itself to Swift very well. The initializers in Swift that rely on these constants have to be *failable*. If you have never heard of a *failable* initializer, the syntax is as follows:

```
init?(constant: String) {
    if constant != "<expected>" {
        return nil // Failed to initialize
    }
    // Initialized
}
```

A failable initializer is marked with the ? character. Then, when the initializer does not have the expected values, returning `nil` will cause the initializer to fail and return a `nil` value. Any variable that is created from a failable initializer will be optional. This is not a very Swift API. The proposal uses the HealthKit API, and it is a perfect example of why this is necessary. Here are some of the constants used in HealthKit:

- `HKQuantityTypeIdentifierBodyMassIndex`
- `HKQuantityTypeIdentifierBodyFatPercentage`
- `HKQuantityTypeIdentifierHeight`
- `HKQuantityTypeIdentifierBodyMass`
- `HKQuantityTypeIdentifierLeanBodyMass`

All of these constants are strings and the `HKQuantityType` has this API:

```
HKQuantityType.quantityTypeForIdentifier(identifier: String)
```

This method returns an optional `HKQuantityType` because you can pass it any string you want. The aim of this proposal is to convert these string types to a Swift type such as an `enum` or a `struct`. This would allow the API to be

```
HKQuantityType.quantityType(for identifier: HKQuantityTypeIdentifier)
```

where `HKQuantityTypeIdentifier` is an `enum` that is defined as such:

```
enum HKQuantityTypeIdentifier : String {
    case BodyMassIndex
    case BodyFatPercentage
    case Height
    case BodyMass
    case LeanBodyMass
}
```

This will allow you to pass the previous method one of these values and the method can then return a `non-nil` value. This will allow APIs to be less optional, but it gives us an added benefit. We can now use Swift's powerful type inference instead of having to type out the full constant, which saves us time and keeps our code leaner.

SE-0005

The last proposal we are going to examine is SE-0005, which is a better translation of Objective-C APIs into Swift, and was written by Doug Gregor and Dave Abrahams. This proposal brings massive breaking changes to our code. The bullet points of this proposal are the following:

- Prune redundant type names.
- Add default arguments.
- Add first argument labels.

- Prepend “is” to Boolean properties.
- Lowercase values.

As stated in the new API Design guidelines, conciseness and clarity are the goals of the new APIs and Objective-C can be redundant. Take a look at this code:

```
addGestureRecognizer(recognizer: UIGestureRecognizer)
setTextColor(color: UIColor)
stringByTrimmingCharactersInSet(set: NSCharacterSet)
```

The first aim of this proposal is to remove this redundancy. In Swift 3, the previous APIs might become the following:

```
add(recognizer: UIGestureRecognizer)
setText(color: UIColor)
trimming(set: NSCharacterSet)
```

This makes Swift code even more readable, clear, and concise. The next two bullet points are adding default arguments and first argument labels. These are self-explanatory. We only want to write purely necessary code and nothing else. The argument labels were the proposal we examined first and is important to how these APIs will be read and understood. The last two bullet points are simple. Any Boolean property will now be prefixed with “is” and values such as `CGColor` will be renamed `cgColor`.

This section has walked through three of the biggest proposals that have been implemented in Swift 3. There are way too many proposals to talk about here and they are all important.

I wanted to make sure to thank anyone who has submitted a proposal or implemented a proposed change to Swift 3. I absolutely love the Swift language and I appreciate all the work that has gone into making it the best possible language it can be.

Wrap Up

One down, eleven more to go! This chapter walked you through the evolution of Swift from the release of Swift 1 in 2014 to the newest release of Swift 3 in 2016.There are a lot of new features in Swift 3 and I have shown only a few.

I have presented a lot of information here and will be reinforcing topics from this chapter throughout this book, especially the new API Design Guidelines. Swift has the potential to go anywhere and it is getting stronger and more robust with every release. Up next, we will be walking through Xcode, including some of the more advanced features like the Address Sanitizer and the new Memory Graph Debugger.

Chapter 2

Xcode

We just covered the evolution of Swift from Swift 1 through Swift 3. Now that you have seen a few of the new features in Swift 3, let's take a step back and examine the new tools we have in Xcode 8. The new features I want to highlight in this chapter pertain to things we will use in this book and what I believe every Swift developer should know how to use effectively. I am going to provide a general overview of the new features and tools, but I would highly recommend more research. The best way to learn a new tool is to just use it, so you should try to work these new features into your everyday development.

What You'll Learn

This section is going to walk through the new features within Xcode 8 and some features introduced prior to Xcode 8 that are worth talking about. First, we have the "What's New" section. In this section, we will look at some of the new features in Xcode 8. One of those features was added to storyboards and the other is the amazing addition of Xcode extensions. After this section, you should feel comfortable using the new storyboard Device Configuration panel and the features associated with this update. You will also learn about the new app extensions that can be built for Xcode. This is a really exciting new feature that will hopefully open the door to more community support for Xcode.

After we look at a couple of the new features, we are going to look back at previous features that have been in Xcode. We are going to quickly look over Xcode code coverage. Code coverage can give you a good idea about the effectiveness of your tests. This will be more important when we get to later chapters and when we will build our app. The last old feature we will examine is UI Debugging. This is a pretty cool feature within Xcode and it is very well hidden.

Looking through these features will be fun, but the focus of this chapter is going to be centered on the Address Sanitizer, Thread Sanitizer, and the Memory Graph Debugger. These awesome tools are a bit difficult to understand and use. After this chapter, you should feel comfortable using these tools and understanding what they offer you. Let's get started with a few of the new features in Xcode 8.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_2

> **Caution** Feel free to follow along in Xcode. However, this chapter is more of an overview for the features of Xcode and not necessarily a walkthrough.

What's New

First, I want to bring your attention to the Issue Navigator panel in Xcode 8. You might notice there are two new sub-tabs within this panel. This can be seen in Figure 2-1. The first sub-tab is *called Buildtime.* This is the tab we should all be familiar with because it is where we see all of the warnings and build errors we are so used to. The second sub-tab, however, is new. This one is called *Runtime*. Now, in Xcode 8, we will get to see issues that arise during runtime, such as potential memory leaks. The new runtime issues work hand in hand with the new Memory Graph Debugger, which we will discuss later in this chapter.

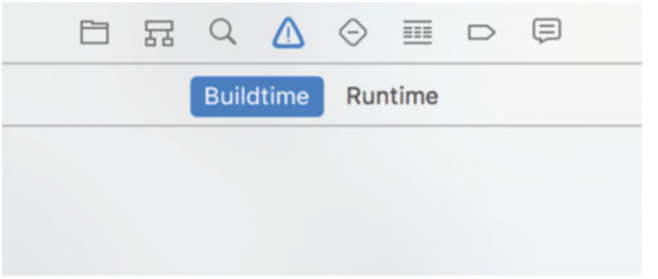


Figure 2-1. *New Issue Navigator panel with a Buildtime errors tab and a Runtime errors tab*

The new Runtime sub-tab can also display potential threading issues that will lead us into the Thread Sanitizer. The last type of issue that can show up here is related to Auto Layout. When a view has conflicting constraints at runtime, those issues will show up here as well.

The next feature added is a small one, but still pretty cool. Xcode 7.1 Swift playgrounds added *color*, *image*, and *file literals*. These literals made it very simple to embed content such as an image or color reference without using code. Now, Xcode 8 has the color and image literals available to us in all of our projects. Figure 2-2 shows Xcode 8 adding a new color literall set a constant color for the navigation bar tint color. To create one of these color literals, just start typing **Color Literal** in the source; clicking on Xcode's suggestion should present the menu from Figure 2-2.

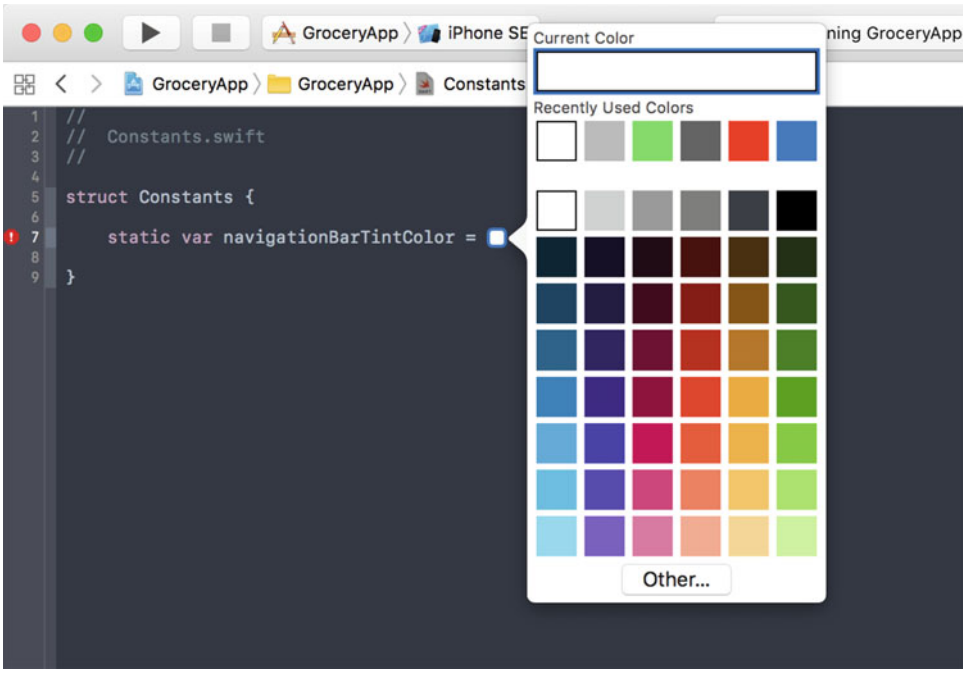


Figure 2-2. *Xcode 8 adding the new color literal to a Constants struct in an Xcode project*

The next small feature Xcode 8 added was the addition of *automatic code signing*. I cannot tell you how many times I have struggled with code signing. This is a huge addition. In Figure 2-3, you can see the new sections in the app's Target settings under the General tab. It has *Automatically manage signing* turned off, but once I turn it on, I can choose my *Team*; I then get an Xcode managed provisioning profile, the signing certificate from my Apple ID.

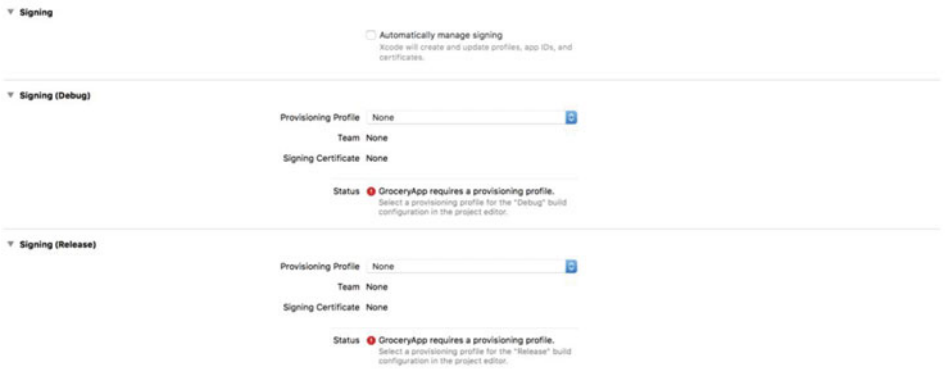


Figure 2-3. *New automatic signing section in the Target settings under the General tab*

That's just a few of the new features that are available in Xcode 8. There is a lot more that happened and it can be found at https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/WhatsNewXcode/introduction.html. Now, let's get to the meat of this section. We are going to start with the new changes in storyboards and Auto Layout.

Storyboards and Auto Layout

Storyboards and Auto Layout are going to be the focus of this section. There are some great new features and we will be using these new features heavily in Chapter 7 and Chapter 9. Chapter 7 will be all about Auto Layout and storyboards. Chapter 9 is where we build the app's UI, but it all starts here.

Figure 2-4 shows the new Device Configuration panel that was added in Xcode 8. This panel can be found at the bottom of Xcode when editing a storyboard by clicking on *View as: <device name>*. This panel has taken the place of size classes. Before, you were able to select a size class to edit your view such as wR hC. This stands for width regular and height compact. If you look closely at Figure 2-4, you can still see the size class, but this new panel now allows us to use actual device configurations instead of abstract size classes. This makes things so much easier. We can switch between devices and orientations with ease to see how our view reacts.

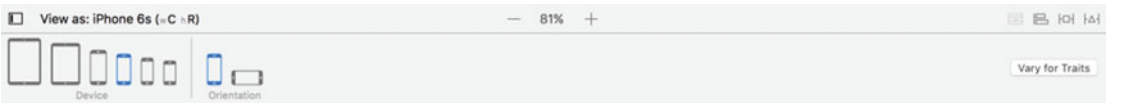


***Figure 2-4.** The new Device Configuration panel at the bottom of a storyboard file. Using this panel, we can see view layouts in all devices and orientations*

You will also notice there is a button on the right that says *Vary for Traits*. Trait variations are a new topic we are going to examine more in-depth in Chapter 7. The short explanation is that when you click this button, you can add or modify the traits of a particular element for a particular configuration.

The last feature that was added to storyboards in Xcode 8 is Autoresizing Masks. Actually, this feature has also been in storyboards. Before Auto Layout, in the dark days, there was Autoresizing Masks. When building views with Autoresizing, we could allow elements on the screen to adapt to the screen size, but this was before iOS added so many devices with different screens and resolutions to support. Auto Layout was added to allow our views to be more dynamic and resize to any device configuration. However, many times, our views do need the power of Auto Layout depending on their simplicity. This is where autoresizing is coming back.

In previous storyboards, when a view element did not have constraints with Auto Layout, implicit constraints were added. This is no longer the case. Now, we can configure our view elements to use the old style autoresizing. This will also help with migrating old style layouts to Auto Layout. Let's see how we can use this on a view. Figure 2-5 shows a view I configured with a button that says *Do Something*.

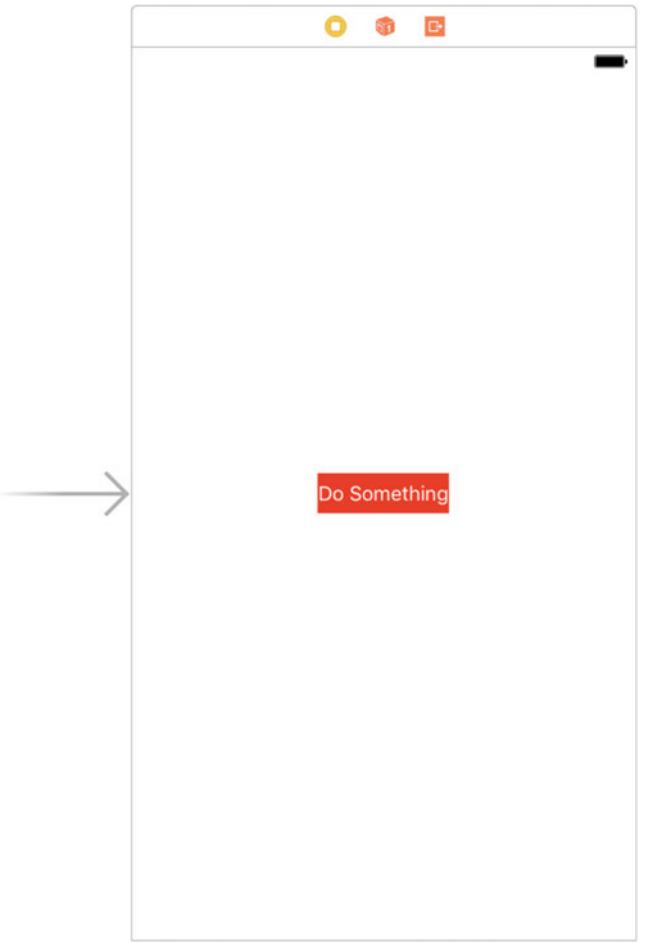


Figure 2-5. View with a single button that has no constraints

Now selecting the button, if you flip over to the Size Inspector panel on the Utilities panel, you will see Figure 2-6, which shows just the *Autoresizing* section. Focusing on this section, you can see that I have configured the leading / trailing to stick to the sides of the screen and that I've told the inner spring to expand horizontally and vertically.

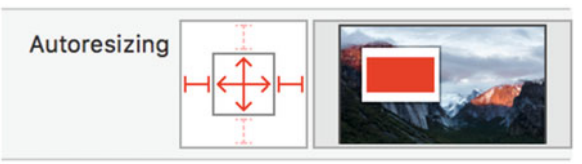


Figure 2-6. Autoresizing section in the Size Inspector

With this configuration, nothing happens in the current view, but let's switch between landscape and different devices. Figure 2-7 shows what happens to the red button using the current device in landscape.

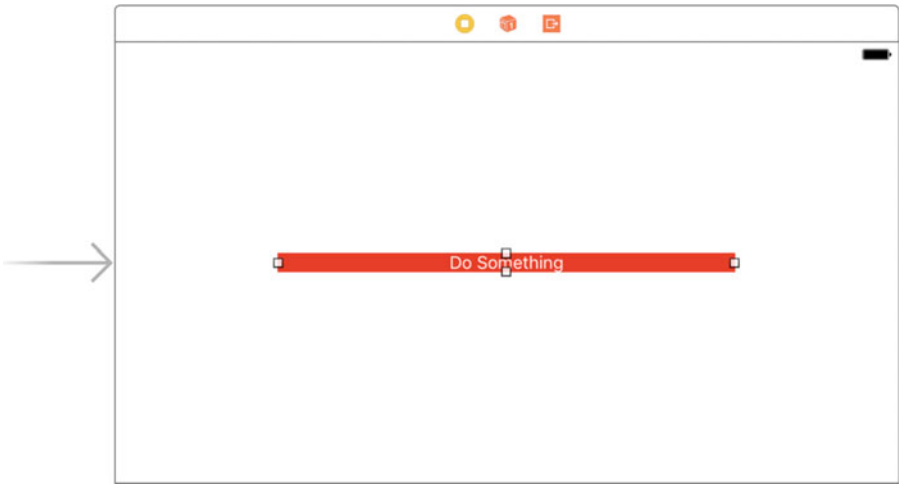


Figure 2-7. Red button resizes on an iPhone 6 in landscape

Using the Autoresizing capabilities, the red button's left and right sides are proportionally the same distance from the side of the view as they were in Figure 2-5. The horizontal spring configuration allows the button to scale horizontally. In Chapter 7 and Chapter 9, we will be looking more into the new trait variations and we can use the old autoresizing behavior to configure the simple views in our app. The last feature we are going to examine in this section is the new Xcode extensions.

Xcode Source Editor Extensions

You better sit down for this next part. Xcode 8 has now opened up development for everyone. In OS X Yosemite, App Extensions allowed developers to create small features for apps such as Finder and Notification Center. This same app extension system is now available in Xcode 8 and is called Xcode Source Editor Extension It is important to note that you will need to create a full macOS app to create an Xcode extension. Figure 2-8 shows the steps to create a new source editor extension. Step 1 in Figure 2-8 first creates a new macOS Cocoa app. After creating a macOS Cocoa application, go to the project settings and add a new target. Step 2 shows the menu for adding a new target and at the bottom is the Xcode Source Editor Extension icon.

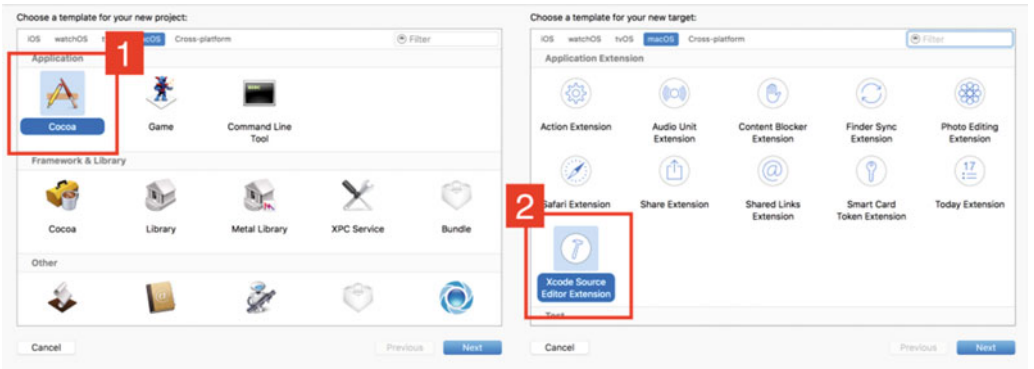


Figure 2-8. *The process to create an Xcode source editor extension*

> **Note** (1) Create a new macOS Cocoa project. ➤ (2) Add a new target. ➤ (3) Add a new Xcode source editor extension. Steps 1 and 3 are highlighted in the above figure.

Apple made the decision to use the app extension system to allow developers to create an extension and sell it in the App Store. These extensions are very limited right now. Currently, the only action you can perform with an Xcode extension is string replacement within the current file. This might not seem like much, but the 2016 WWDC video showed an extension that converted all color and image code references to the new color and image literals. The following code block shows the default code you get when you start a new Xcode source editor extension.

```
import Foundation
import XcodeKit

class SourceEditorCommand: NSObject, XCSourceEditorCommand {

    func perform(with invocation: XCSourceEditorCommandInvocation, completionHandler:
    (NSError?) -> Void ) -> Void {
                // Implement your command here, invoking the completion handler when done.
//Pass it nil on success, and an NSError on failure.

completionHandler(nil)
    }
}
```

The code uses the class `XCSourceEditorCommandInvocation`. This class gives you access to a text buffer for the file that is currently open when running the extension. Unfortunately, there is no information about the project or any other context within Xcode. Sadly, this means we will not be able to get refactoring functionality from an extension.

This should not deter you from checking out this new functionality. Being capable of extending the Xcode app is a huge leap forward and opens the door for more community support in the future. Up next, we will be diving into previous Xcode features that I want to highlight in this chapter.

What's Old

This section is going to highlight two previous features of Xcode. Those features are *code coverage* and *UI debugging*. I wanted to quickly go through these features because we will use them when we build our app starting in Chapter 9. Let's start with Xcode's code coverage.

Code Coverage

The Xcode code coverage tool is not new in Xcode 8. It was actually released in Xcode 7. Code coverage is a very important tool. It can tell you how much of your code is covered in tests and what you still have untested. It can also lie. Code coverage tools, whether they are in Xcode or not, cannot discern the meaning of your test or how effective your test is at asserting particular cases.

Xcode's code coverage tool is very minimal and it gives you a count of your coverage right in the source file. This coverage count is based on the number of times a set of instructions is run from a test. So even though a test might not assert anything, it would show up as covered if it ran the instructions of a method or class.

You can quickly scan through a file to see coverage. Uncovered code will show up red and covered code will show up as green. Xcode also has a more detailed coverage report in the Report Navigator. Figure 2-9 shows the Report Navigator. In Figure 2-9, you can see that I have three passing tests for one file. I have also added the results of the Coverage tab. It shows the code we have tests for is 100% covered and everything else is uncovered. At the top of the coverage report, it shows the coverage for the entire app.

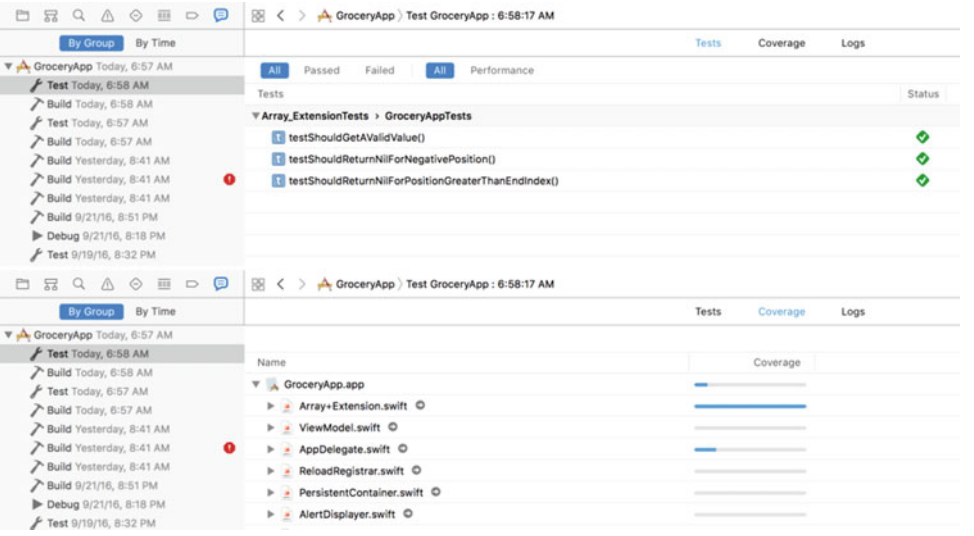


Figure 2-9. *Two views within the Report Navigator displaying the passing tests and the code coverage*

To enable this feature, you have to edit the scheme you want test. So, you would select the active scheme you want to enable code coverage for and then select *Edit Scheme*. Figure 2-10 shows the scheme configuration. Select the test section and then you will see an option for *Code Coverage*. This is turned off by default.

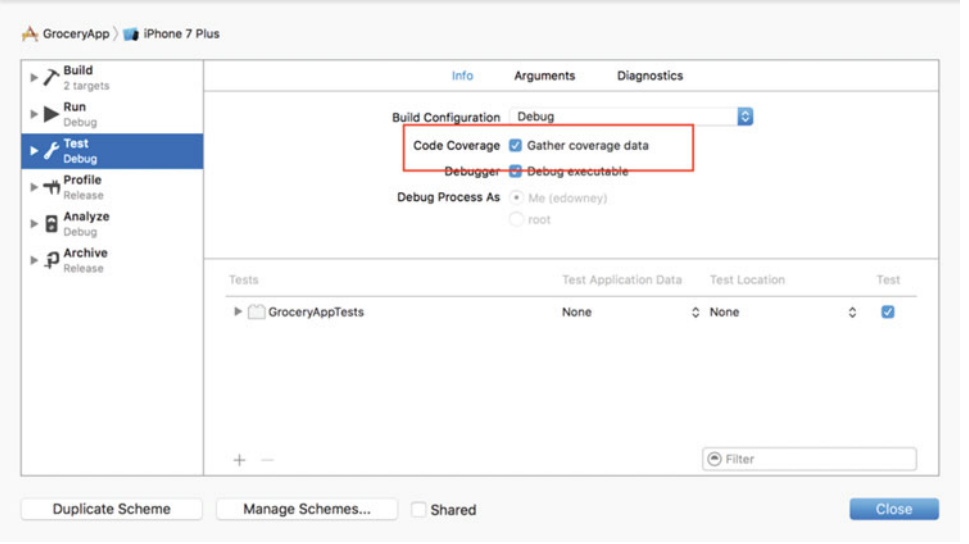


Figure 2-10. *Edit your project's scheme to turn code coverage on*

Once you have the code coverage option turned on for your scheme, navigate to a source file and click the *Editor* menu. Figure 2-11 shows the Editor menu and the option at the very bottom is how you show code coverage in source files. After selecting this option, the code coverage for source files will show up in the right gutter.

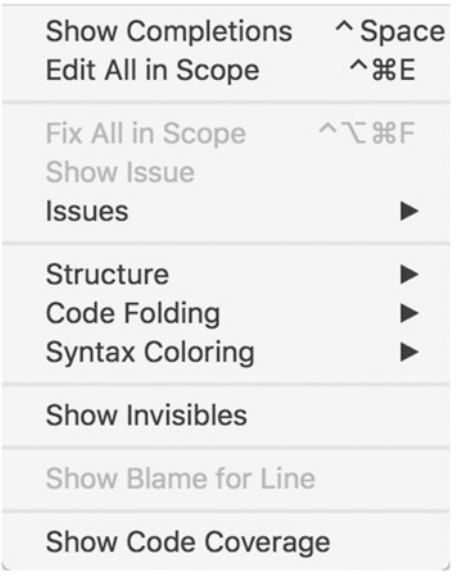


Figure 2-11. *The Show Code Coverage option in the Editor menu*

Code coverage is a pretty good tool for getting a sense for the amount of tests you have and sometimes even the quality of your tests. We are going to use this feature when we build our app in Chapter 9. The next section is going to briefly discuss the UI Debugging tool available in Xcode.

UI Debugging

UI Debugging is probably one of my favorite tools in Xcode. It was introduced way back in Xcode 6 and iOS 8. The goal of the tool is to give developers the ability to see the entire view hierarchy at runtime. It's also pretty cool because you can configure the tool to see views that are cut off or off screen. It also allows you to see the constraints that are applied to each and every view. It can be invaluable when you have problems with your view and cannot figure out why from the code or Interface Builder.

The UI Debugger gives you a 3D snapshot of your app while it is running. Figure 2-12 shows how to activate the UI Debugger in Xcode. The button is at the top of the bottom pane that is available while your app is running. It is directly to the right of the breakpoint options.

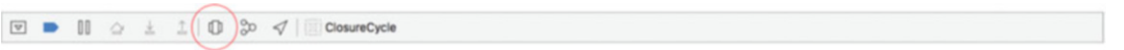


***Figure 2-12.** Shows how to start the UI Debugger. The button is at the top of the bottom pane that is available while your app is running*

> **Note** You can also activate the UI Debugger from this menu: Debug ➤ View Debugging ➤ Capture View Hierarchy

You can rotate your UI around and see if views are hiding behind others or are somewhere off in space. This tool can also display the constraints that are active on any particular view. This can be a huge time saver when you have ambiguous layouts. The UI Debugging stops execution of your app in the same way a breakpoint does. This means you can inspect elements on your view via the memory address. Figure 2-13 shows the UI Debugger in action.

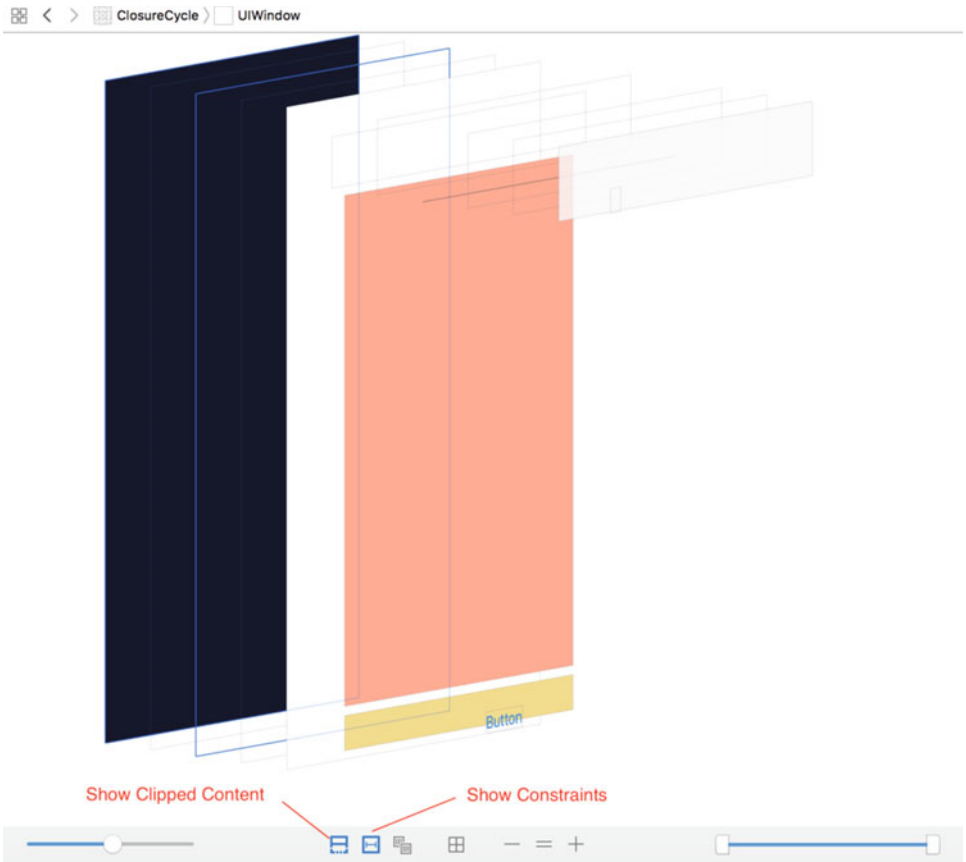


Figure 2-13. The UI Debugger with constraints and subview clipping turned off

The view in Figure 2-13 shows why this is one of my favorite tools in Xcode. It shows the 3D view of the view hierarchy at runtime. This is so helpful when you have views that are hiding behind others. Before this tool, there was no good way to see what was wrong if a view just did not show up. This tool can alleviate a lot of that pain.

Also, this tool pauses the app just like the debugger, so you can use LLDB commands in the console to get further information about your view elements. Figure 2-14 shows the console after starting the UI Debugger. I found the memory address of a `UIButton` on the view and then using the LLDB command `po`, I was able to get information about that element. The rest of it shows setting up a variable through the `expression` command aliased to `e` and then printing out the `layer` property of that particular `UIButton`.

> **Note** LLDB is a command-line debugging environment integrated with Xcode.

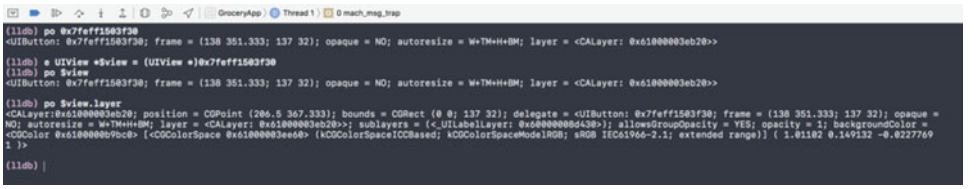


Figure 2-14. The Xcode console and the use of LLDB commands to debug the UI

> **Caution** Any variables declared through LLDB commands with the expression command must start with a $.

The UI Debugger is a very powerful tool that can give you a great idea of what is going on in the view hierarchy. There is also a lot you can do through the LLDB debugger commands and Figure 2-14 highlights some of the basic features. That about does it for this section. I have just scratched the surface of all the new and old features that are worth talking about in Xcode. The next section is going to be really fun. We are going to cover the Thread Sanitizer and the Address Sanitizer.

Sanitizers

The *Thread Sanitizer* is a new feature that was introduced in Xcode 8. The *Address Sanitizer* was released with Xcode 7, but only for Objective-C. Xcode 8 is bringing support for the Address Sanitizer to Swift. This section will take a look at both of these sanitizers and the capabilities they offer us. Figure 2-15 shows how to turn on the Address Sanitizer and Thread Sanitizer through the scheme-editing menu. Let's start with the Address Sanitizer and memory corruption.

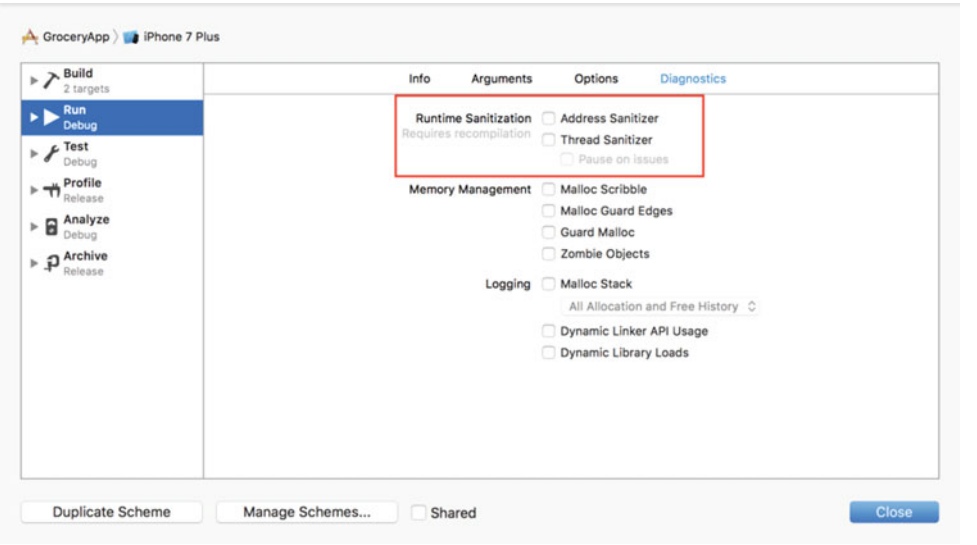


Figure 2-15. Turn on the Address Sanitizer and Thread Sanitizer for your project's scheme

> **Note** Malloc Stack must be turned off in order to enable Address Sanitizer or Thread Sanitizer.

Address Sanitizer

First, let's look at the Address Sanitizer. The purpose of this tool is to help you fix memory corruption errors. In Xcode 7, it only supported C languages. Memory corruption is very easy in C languages. C is very powerful, but it can be very unsafe. In C, you can access an array element that is out of bounds. It then gives you whatever value is at that memory location without an error. This is clearly wrong, but C will let you continue, and this is what the Address Sanitizer is meant to solve.

As of Xcode 8, the Address Sanitizer is compatible with Swift 3. You might be wondering how you could run into a memory corruption error in Swift. It's supposed to be safe, right? It is unless you use the appropriately named `UnsafeMutablePointer` or `UnsafePointer`. It literally has unsafe in the name. This is right from Apple's documentation on `UnsafePointer`:

> *A raw pointer for accessing data of type Pointee. This type provides no automated memory management, and therefore must be handled with great care to ensure safety.*
>
> —Apple, Inc.

You might use these types when interacting with some of the lower-level C APIs that are available. The following code block is meant to illustrate how unsafe these `UnsafePointer` objects can be. You would never write this, but this will cause the Address Sanitizer to yell at you:

```
func unsafelyAdd(one: UnsafePointer<Int>, two: UnsafePointer<Int>) -> Int {
    return one.pointee + two.pointee
}
```

I created this global function to unsafely add two `Int`s. It takes two parameters that are both `UnsafePointer` objects that contain `Int` values. I then use the property `pointee` to access the `Int` the pointer is wrapping. This code will work. The following code block shows how you call this function:

```
var one = 1
var two = 2
let value = unsafelyAdd(one: &one, two: &two)
```

Now, let's make this implode. The `UnsafePointer` is just a pointer to some memory address. We can do some crazy stuff with this object, but it might not be a good thing. The following code block is how you can change the `unsafelyAdd` function so that it will cause the Address Sanitizer to get involved:

```
func unsafelyAdd(one: UnsafePointer<Int>, two: UnsafePointer<Int>) -> Int {
    return one.pointee + two[1]
}
```

The previous code is completely valid. The subscript operator on the `UnsafePointer` struct is meant to access the memory address of the pointer plus an offset. If you look at the documentation for this, it has a precondition that the memory address plus the offset is initialized. In the preceding code, I have no idea what is at that memory address. So, if I run this with the Address Sanitizer turned on, what happens? Well, we get a huge output to the console and Xcode pauses on the line within the `unsafelyAdd` function. Here is the console output:

```
=================================================================
==18694==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff5535d5e8 at pc
0x00010a8a361c bp 0x7fff5535d530 sp 0x7fff5535d528
READ of size 8 at 0x7fff5535d5e8 thread T0
    #0 0x10a8a361b in _TF12ClosureCycle11unsafelyAddFT3oneGSPSi_3twoGSPSi__Si
    ViewController.swift:23
    #1 0x10a8a31e6 in _TFC12ClosureCycle14ViewController11viewDidLoadfT_T_ ViewController.
    swift:18
    #2 0x10a8a36d2 in _TToFC12ClosureCycle14ViewController11viewDidLoadfT_T_ ViewController.
swift

    // TL;DR

Address 0x7fff5535d5e8 is located in stack of thread T0 at offset 72 in frame
    #0 0x10a8a2dbf in _TFC12ClosureCycle14ViewController11viewDidLoadfT_T_ ViewController.
swift:13

  This frame has 3 object(s):
    [32, 40) ''
    [64, 72) '' <== Memory access at offset 72 overflows this variable
    [96, 112) ''
HINT: this may be a false positive if your program uses some custom stack unwind mechanism
or swapcontext
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow ViewController.swift:23 in _TF12ClosureCycl
e11unsafelyAddFT3oneGSPSi_3twoGSPSi__Si
Shadow bytes around the buggy address:
  0x1fffeaa6ba60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

    // TL;DR

==18694==ABORTING
```

There are a lot of memory addresses and really low-level stuff here. Ultimately, the important part is where is it says `This frame has 3 object(s)` and `Memory access at offset 72 overflows this variable`. Essentially, we have gone beyond the size of the `Int`.

This might seem scary at first, but it is very important. If I were to run this app with the Address Sanitizer turned off, nothing happens. It accesses whatever is at that memory address and adds it with the first value. This might not seem like a big deal here, but what if the memory address it accesses and uses is something critical to the app? It could very well corrupt that memory and then the entire app could do some pretty weird stuff. Next up, the Thread Sanitizer.

Thread Sanitizer

The *Thread Sanitizer* is a new addition to Xcode 8 and is used to detect data races. There is no one line of code we can write to cause an issue that will be detected by the Thread Sanitizer. To cause an error that can be detected by the Thread Sanitizer, I am going to use *dispatch async* to put some work on a background thread. Let's start with a `Model` class:

```
class Model {
    var data: Int = 0
}
```

We have the `Model` class that holds the app's data. This next part is where the code creates a race condition by using two background threads:

```
class ViewController: UIViewController {

    var model = Model()

    override func viewDidLoad() {
        super.viewDidLoad()

        DispatchQueue.global().async {
            self.model.data = 3
        }

        DispatchQueue.global().async {
            self.model.data = 4
        }
    }
}
```

In this code, there is a view controller subclass with the `model` property, and in `viewDidLoad` it tries to set the data on `model`. It sets the data twice on two separate background threads. Basically, there is no way to tell which one is going to execute first. Upon running this code, the Thread Sanitizer outputs a warning to the console and a runtime issue shows up in the Issue Navigator. The following block shows the output of the console:

```
==================
WARNING: ThreadSanitizer: data race (pid=19543)
  Write of size 8 at 0x7d08000045f0 by thread T2:
    #0 _TFC12ClosureCycle5Models4dataSi ViewController.swift (ClosureCycle+0x000100002fe7)
    #1 _TFFC12ClosureCycle14ViewController11viewDidLoadFT_T_U0_FT_T_ ViewController.swift:33
    (ClosureCycle+0x0001000003cb5)
    #2 _TPA__TFFC12ClosureCycle14ViewController11viewDidLoadFT_T_U0_FT_T_ ViewController.
    swift (ClosureCycle+0x0001000003d8e)
    #3 _TTRXFo___XFdCb___ ViewController.swift (ClosureCycle+0x0001000003af5)

    // TL;DR

  Thread T2 (tid=7411193, running) created by thread T-1
    [failed to restore the stack]
```

```
  Thread T1 (tid=7411192, running) created by thread T-1
    [failed to restore the stack]

SUMMARY: ThreadSanitizer: data race ViewController.swift in _TFC12ClosureCycle5Models4dataSi
==================
```

Just like the Address Sanitizer, there is a lot here, but it is just trying to trace where the data race issue takes place. Unlike the Address Sanitizer, the Thread Sanitizer will not stop execution of the app on the line that creates the race condition; it just tries to trace it and create a runtime issue. The runtime issue can be seen in Figure 2-16. This Issue Navigator tries to give you as much information as possible. You can see it locates the data race issue to the `Model` class and gives us a memory address. There is also information about the separate threads that caused this issue.

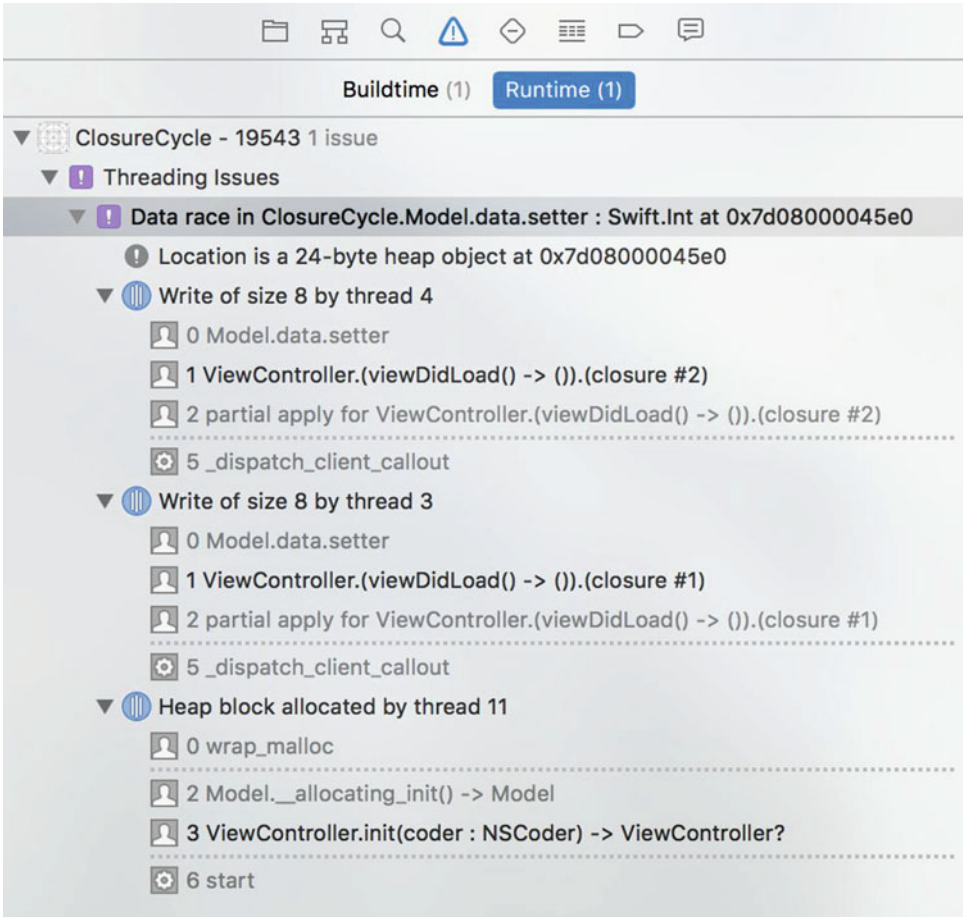


Figure 2-16. *Issue Navigator displaying the data race issue in the app*

Just like the memory corruption errors the Address Sanitizer showed us, the data race issue does not cause the app to crash or anything visually bad happen, so it would be very difficult to trace this error down with the Thread Sanitizer. These tools should be a part of your day-to-day development to make sure these issues do not fester in your codebase.

I hope this section has given you a good introduction to the Address Sanitizer and Thread Sanitizer. They have specific use cases for the `UnsafePointer` and threading issues, but they can highlight issues that you might not know are there. For this reason, they are invaluable tools that should be understood and used. The next and last feature of Xcode 8 we are going to examine is the Memory Graph Debugger.

Memory Graph Debugger

The new *Memory Graph Debugger* is bringing more instrumentation to Xcode 8. In previous versions of Xcode, if you wanted to check for memory leaks, it would require a trip through *Instruments*. I do not know about you, but I usually forgot this step when developing features for my apps. This meant I would usually find memory leaks later and have no context for that section of the app. Now that this tool is integrated into Xcode 8, it will be easy to find memory leaks and fix them earlier and faster.

Let's take a look at how we can use the Memory Graph Debugger. It works the same way as the UI Debugger. It will snapshot your app and gives you a massive amount of information regarding memory allocations and references for your app at that time. Figure 2-17 shows how to enable the Memory Graph Debugger at runtime and the Memory Graph Debugger in action.

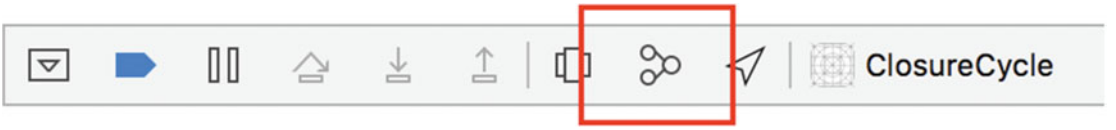


***Figure 2-17.** Enabling the Memory Graph Debugger. You can find this in the bottom bar directly to the right of the UI Debugging button.*

In Figure 2-18, we can see how the memory for the `ViewController` class is allocated. First, there is a `UIWindow` on the left. Then, a `UINavigationController` is allocated that then contains a `NSMutableArray` that holds onto this `ViewController`. You can see this in the API for `UINavigationController`. This is not all of the information we get, though. We can expand and collapse each of these nodes to see more information. If we expand the `UIWindow` node, it shows us what leads to the `UIWindow` being allocated. Pretty cool stuff!

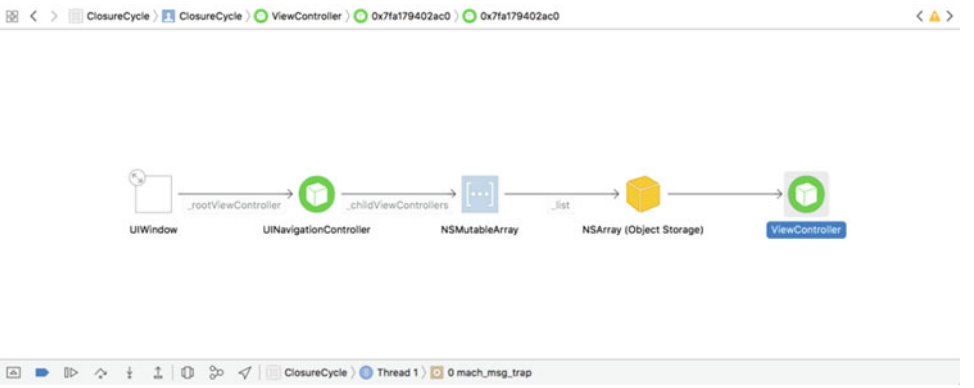


Figure 2-18. Xcode with the Memory Graph Debugger running

Now, that we have some background on this tool, let's see a common memory problem in Swift. Closures in Swift can make our code really simple, but they have some drawbacks, specifically having to do with memory. We have to be careful with the capture lists and strong references that closures can create. The following example is going to intentionally capture self. Let's start with the code that holds onto the closure:

```
class MyClass {
    var closure: ((Void) -> Void)? = nil

    init() {}
}
```

This has the optional closure and that is it. The app I have set up has two view controllers. The first view controller is configured to segue to the second controller through a navigation controller. Then the `viewDidAppear` method for the second view controller looks like the following:

```
class SecondViewController: UIViewController {


    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        let instance = MyClass()

        instance.closure = {
            print(self)
            print(instance)
        }
    }
}
```

`ViewDidAppear` creates an instance of the `MyClass` and then sets the closure to print the view controller and the instance of `MyClass`. This is pretty simple, but once this code executes and we transition back to the first view controller, this leaks memory. Transitioning back to the first view controller should deallocate the second view controller, but the closure has captured self, so the second view controller sticks around. Figure 2-19 shows the Memory Graph Debugger after transitioning away from the second view controller.

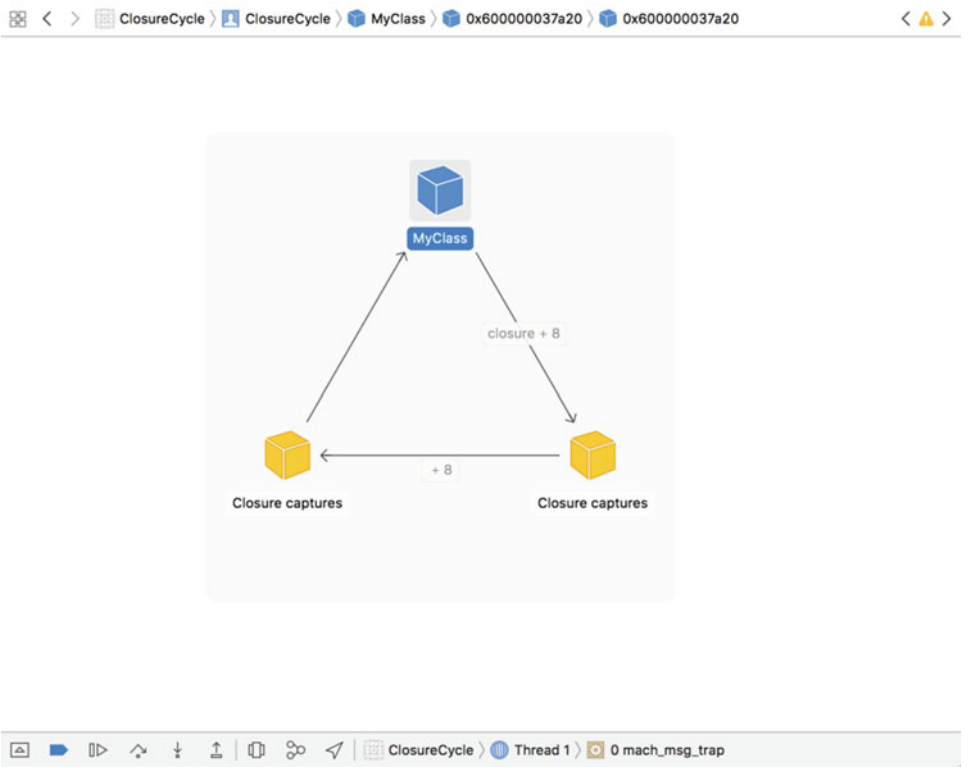


Figure 2-19. *The Memory Graph Debugger after leaking memory by capturing self through the closure on MyClass*

It's pretty obvious that the closure creates a strong reference cycle. This is really easy to do by mistake, and suddenly we are leaking memory. Now, if we captured something with a lot more allocated memory, this could cause a crash quickly. The fix for this is to make sure we create ***weak*** references for the items being captured. The `SecondViewController` will now look like the following:

```
class SecondViewController: UIViewController {


    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        let instance = MyClass()

        instance.closure = { [weak self, weak instance] in
            print(self)
            print(instance)
        }
    }
}
```

The only difference is the capture lists on the closure. We add `[weak self, weak instance]`. By marking these two captured objects as weak references, the reference count for these objects are not increased by the closure. This means that everything will deallocate once the navigation stack pops this view controller from the navigation stack.

That does it for the Memory Graph Debugger. We have seen how to use the debugger and even how to fix common closure errors through weak references. There is so much I have not covered here. The Memory Graph Debugger and all of these tools deserve a lot more research.

Wrap Up

This chapter has covered a lot of new and old features in Xcode. We started with the new color and image literals, the new runtime issues, and the automatic signing features. All of these additions are going to make our lives simpler and make development faster. We then discuss the Xcode Source Editor Extension. This addition has great potential and hopefully leads to a more community-driven approach to Xcode upgrades (it could happen ☺).

We then took a step back to older features that have been in Xcode for a release or two. After walking through this chapter, you should know how to effectively use Xcode's code coverage tool. The reintroduction of Autoresizing Masks is a cool throwback and should make the lives of developers still supporting older interfaces a little better.

After all of the previous features, we examined the Address Sanitizer and Thread Sanitizer. You should now feel comfortable folding these tools into daily development to highlight issues that may have flown under the radar. We covered the concept of memory corruption in C and Swift with `UnsafePointer`. The Thread Sanitizer then showed how easily it could be to create previously undetectable race conditions through the use of background threads.

Finally, there was the Memory Graph Debugger. I am really excited about this tool. It is so much easier to use than Instruments and creates a very fast feedback loop for memory leaks. Previously, memory leaks were so hard to find and required time in Instruments. We can now just flip on the Memory Graph Debugger while running an app and see if there are any leaks. Memory Graph Debugger then tells us where this memory was leaked. It couldn't be easier.

The intention of this chapter was to give you an overview of the IDE you use every day to develop Swift/iOS code. Xcode has a mountain of features, and sometimes it can be hard to find the cool little features that make us faster. Memory leaks, data race conditions, and memory corruption are so obscure and hard to fix; it is important we understand what causes these issues and how to fix them to create cleaner apps. And that's it for this chapter. The next chapter is going to cover a few third-party package managers in Swift as well as the official package manager.

Chapter 3

Package Managers

This chapter is all about package managers in Swift. We are going to discuss two popular managers, *CocoaPods*, and *Carthage*. CocoaPods has been out for years and is heavily used. Since the introduction of Swift, however, Carthage has been on the rise. CocoaPods' response to the rise of Carthage is CocoaPods-Rome, but more on that later.

To talk about these package managers, we are going to need to discuss *dynamic frameworks* and *static libraries*. It is important to understand the differences between these and the direction in which Swift is moving. Once we have an understanding of these concepts, we are going to look at how to use these concepts in Xcode.

Afterward, we are also going to take an in-depth look at the official *Swift Package Manager*. I am really excited about this. The Swift Package Manager is the one of the first steps toward Swift running everywhere. We are going to build a Swift package using this package manager. We are going to look at the frameworks we have available and how to pull in dependencies. Let's see the specifics of what you are going to learn in this chapter.

What You'll Learn

In this chapter, you will learn about two of the most popular and widely used Swift package managers, CocoaPods and Carthage. You will learn the differences between the two and why you might pick one over the other. To discuss how these package managers work, we are going to need to understand the differences between dynamic frameworks and static libraries.

Dynamic frameworks and static libraries are very important when we discuss the future of Swift and what iOS and macOS are moving toward. At the end of this chapter, you will understand the differences and be able to pick which one suits your project. Spoiler alert—dynamic frameworks are the future of iOS and macOS packages.

The last section in this chapter is going to be all about the new Swift Package Manager created by Apple. We are going to build our own Swift package here. Chapter 8 will revisit the package we are going to create here. In Chapter 8, we will examine testing in the Swift Package Manager. The rest of this section will look at how to include external dependencies

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_3

in a Swift package. By the end of this section, you should feel comfortable creating packages with the new Swift Package Manager and including external dependencies to build cool new packages that can run on more than just Macs.

Packaging Code in iOS

This section is going to cover the topics of dynamic frameworks and static libraries. These are the two concepts we have in iOS that allow us to bundle code together so another party can consume it. In Xcode, these concepts are referred to as Cocoa Touch Dynamic Framework and Cocoa Touch Static Library, respectively. Prior to iOS and Swift, static libraries were all that were available to us in iOS, so let's start there.

Static Library

Prior to iOS 8 and the introduction of Swift, the only option Objective-C had for managing and distributing packaged code was the *Cocoa Touch Static Library*. Static libraries are still around, but they have a few limitations. First and foremost, you *cannot* create a static library that contains Swift code, only Objective-C code. Since this is a book on Swift, that seems like a big limitation. Objective-C is still around, however, so it is important to understand how to work with these libraries.

So the first question is what are *static libraries*? They are just what their name suggests. They are libraries of code that are statically linked at compile time. So, let's break down what this means. First, an individual library contains all the header files and implementation files associated with the particular task the library is solving. Then at compile time, the static library is linked with a project. Effectively, this means the library's code and the project's code are sitting right next to each other in the compiled binary. The end result is that the project has access to the code within the library and can use it.

Now, let's discuss the limitations of static libraries. Let's first talk about *static linking*. I said previously that the library's code sits right next to the project's code after it has been linked. This means there could be a situation in which a class within the library is named the same as something in the project's source. Since the code within the library is not separated from the project's code at the link phase, it would be like naming two classes in the project a duplicate name.

This is where prefixing Objective-C code comes from. When a third-party library is created, if all classes are properly prefixed, then there should be no naming collisions. An example would be if a networking library called one of its classes `HTTPRequestSerializer`. This class would have to be prefixed with something representing that library, so in the case of `AFNetworking`, it would be `AFHTTPRequestSerializer`. The `AFNetworking` library prefixed its classes with `AF`. This library can be found on GitHub at https://github.com/AFNetworking/AFNetworking. This is obviously a pain when creating these libraries. Let's move on to the next big limitation.

Another limitation is static libraries are not allowed to contain any images or assets. They cannot contain any PNGs, image catalogs, Nib files, or storyboard files. This is a huge limitation since apps require these assets to be there. It is possible to bundle these assets

together, but they cannot be contained within the library itself. In order to contain images and assets you must create a separate *bundle*. This bundle can then be distributed with the static library. Although this works, it is not ideal to separate the concept of code and a bundle. It would be nice if it could all be packaged together. Cue dynamic frameworks.

Dynamic Framework

Once iOS 8 hit and Swift was released, Apple opened the door for iOS developers to create and manage the Cocoa Touch Dynamic Framework. Just like the static library, the name says it all. Instead of being statically linked, dynamic frameworks are dynamically linked. This means the code is not included with the app's code. Instead, everything is done at runtime. The framework is loaded into memory at runtime, the app can use the pieces of framework it needs, and then the framework can clean up after itself.

As previously mentioned, static libraries do not support Swift code, so dynamic frameworks have to be used. Objective-C, however, has both options available. Before we make a decision on which type of package should be used going forward, let's first look at the benefits of dynamic frameworks. Even though dynamic frameworks work with Objective-C, there are more tangible benefits to using dynamic frameworks in Swift. In Swift, the *access control* model is based on the idea of modules. In Xcode, each app/framework is treated as a separate Swift module. This completely alleviates the pain found in static libraries around naming collisions.

Let's look at the successor to AFNetworking: Alamofire. *Alamofire* is the Swift version of AFNetworking and it is available as a dynamic framework. In the source for Alamofire, there are global functions. An example would be the `request` function responsible for creating a `DataRequest` object. This works because the framework is treated as a module. This means you can include Alamofire in one of your projects and also have a global `request` function without any collisions. This works by using the framework name as the prefix to identify the Alamofire `request` function. The call to this function would look like `Alamofire.request(...` instead of `request(...`. Now, using Swift and dynamic frameworks, prefixing classes is no longer necessary unless you are supporting Objective-C through interpolation.

Dynamic frameworks also come packaged with their own bundles. That is, they can contain images and assets. Using frameworks can keep our PNGs, image catalogs, Nib files, and storyboards all contained within the framework directly with the source code. Therefore, the source within the framework does not need to reference any external bundle. It can just reference the framework's bundle. This is a benefit to any external party consuming the framework. They no longer need to include two separate files in their Xcode project.

So what does this mean for the future? Well, static libraries cannot contain Swift code, so it seems pretty obvious that dynamic frameworks are the winners. At least for Swift code. So, it all depends on your project and what suits you. This section has given you the briefest of introductions to the two concepts and it is a lot more complicated than described here. I would highly recommend more research. Now that we know how Objective-C and Swift code have been packaged for consumption, let's start looking at the package managers. We will start with CocoaPods.

CocoaPods and CocoaPods-Rome

This section is going to cover the popular CocoaPods and the newer response to Carthage, CocoaPods-Rome. *CocoaPods* has been around since late 2011 and is meant to manage dependencies for Objective-C and Swift. CocoaPods is open source and is supported and driven by the community. The source can be found on GitHub at `https://github.com/CocoaPods/CocoaPods`. CocoaPods-Rome is newer and was built in response to Carthage. Let's start with vanilla CocoaPods.

> **Note** Feel free to follow along, but depending on the versions you are using, there might be compatibility issues.

CocoaPods

CocoaPods is built in Ruby and is available as a RubyGem. If you have never used Ruby or RubyGems before, you should definitely check both out. If you look on the wiki page for Swift, under the languages it was influenced by, you will find Ruby. You can see the Ruby influence in certain aspects of Swift. RubyGems.org is a community-driven repository of open source code written in Ruby. This influenced CocoaPods and how their system is designed.

> **Note** The Ruby language can be found at `https://www.ruby-lang.org/en/` and RubyGems can be found at `https://rubygems.org/`.

CocoaPods takes a similar approach and has a public list of available CocoaPods for iOS code. CocoaPods also tries to make the process as easy as possible. CocoaPods can modify an Xcode project and create an Xcode workspace. Once it creates the Xcode workspace, it can then link the packages it has downloaded. This makes it very easy for anyone to set up CocoaPods and start pulling in packages. It then handles all the work and you can immediately start using the packages. In CocoaPods, a package is called a *pod*.

Let's look at how we can use CocoaPods in an example project. There is no need to follow along, but feel free. First, we have to make sure we have Ruby, RubyGems, and the CocoaPods gem installed. Once we have all of these things, we can start by creating a sample Xcode project. I have called my project `CocoapodsTest`. With all of these requirements set up, we move to the Terminal app. Most of this work will take place in Terminal.

To initialize CocoaPods in Terminal, you must navigate to the directory that contains the Xcode project. In my case that directory would be `CocoapodsTest`. Once I am in this directory, I can run the `pod init` command. This will create a file called `Podfile`. This file is where we can define the dependencies for the app. So, in this example, I am going to pull in Alamofire. Before I tell the `Podfile` to pull in Alamofire, here is what it looks like after a fresh `pod init`:

```
# Uncomment this line to define a global platform for your project
# platform :ios, '8.0'
```

```
# Uncomment this line if you're using Swift
# use_frameworks!

target 'CocoapodsTest' do

end
```

You can see the previous code block has a section called `CocoapodsTest` defined with the word `target`. This is the actual target in the Xcode project for `CocoapodsTest`. If I included unit tests or UI tests when I created the project, those targets would be in the `Podfile` as well. By defining different dependencies in different targets within the `Podfile`, we can pull in separate dependencies based on a unit test target or the production app target. Let's add Alamofire to the `CocoapodsTest` target.

If you go to Alamofire's documentation, specifically on the CocoaPods installation, it has you turn the `use_frameworks!` option on and set the platform to iOS 10.0. It also specifies an Alamofire version of 4.x. Adding this pod to the `Podfile`, it now looks like the following:

```
platform :ios, '10.0'
use_frameworks!

target 'CocoapodsTest' do
    pod 'Alamofire', '~> 4.0'
end
```

Now, the `Podfile` is in place. The next step is to install the pod via the `pod install` command. Running `pod install`, I received the following terminal output:

```
Updating local specs repositories


CocoaPods 1.1.0.rc.2 is available.
To update use: `gem install cocoapods --pre`
[!] This is a test version we'd love you to try.

For more information see http://blog.cocoapods.org
and the CHANGELOG for this version http://git.io/BaH8pQ.

Analyzing dependencies
Downloading dependencies
Installing Alamofire (4.0.0)
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use `CocoapodsTest.xcworkspace` for this
project from now on.
Sending stats
Sending stats
Pod installation complete! There is 1 dependency from the Podfile and 1 total
pod installed.
```

Running this command, CocoaPods will resolve all the dependencies between any pods you pull down and their dependencies. You can see on the two lines after it says `Installing Alamofire (4.0.0)` that it generates a new Xcode project and then integrates the `CocoapodsTest` project with the Pods project in an Xcode workspace. After this process is finished, it then states that I must use the new `CocoapodsTest.xcworkspace` instead of the previous `CocoapodsTest.xcodeproj`. Figure 3-1 shows the Project Navigator panel after opening the new `CocoapodsTest.xcworkspace` file. I have the original `CocoapodsTest` project as well as the new `Pods.xcodeproj`. The Pods project then has Alamofire under the pods group.

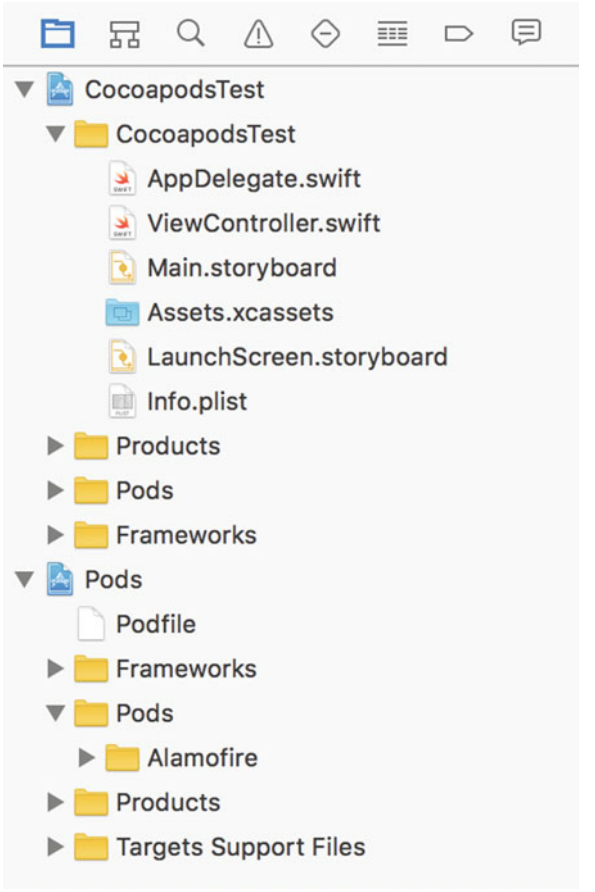


Figure 3-1. *Cocoapods.xcworkspace Project Navigator after running the pod install*

And that's all there is to it. CocoaPods was designed to be very easy to use and it just works. We should be able to import Alamofire now and start making network requests in `CocoapodsTests`. Now, do not be fooled, there is some magic going on in the background here. CocoaPods hides this from you, but it is useful information to know. We will discuss the magic when we get to Carthage. For now, let's move on to CocoaPods-Rome.

CocoaPods-Rome

CocoaPods-Rome is not a different tool from CocoaPods; instead, it is another RubyGem that is built on top of CocoaPods. It can be found on GitHub at https://github.com/CocoaPods/Rome. The purpose of CocoaPods-Rome is to build Cocoa Touch Dynamic Frameworks out of the pods that CocoaPods use. This is useful when you want to use CocoaPods to manage your dependencies, but you want to use the pods outside of Xcode.

So now the question, why is it called CocoaPods-Rome? Well, it comes from the Punic Wars, which took place from 264 BC to 146 BC. The Punic Wars was a series of battles between Rome and Carthage that eventually ended with the defeat of Carthage. So, because of all that, we now get to call our Swift package managers Carthage and Rome. We will discuss Carthage in the next section, but for now, let's focus on CocoaPods-Rome.

CocoaPods-Rome starts off the same way CocoaPods does with a `Podfile` generated via `pod init`. Here is the same `Podfile` as the previous one `CocoapodsTest` used, except this time it is integrating CocoaPods-Rome:

```
platform :ios, '10.0'
use_frameworks!

plugin ‘cocoapods-rome’

target 'CocoapodsTest' do
    pod 'Alamofire', '~> 4.0'
end
```

In the previous block, the only difference in the Podfile is the line that reads plugin ‘cocoapods-rome’. Since CocoaPods-Rome is a gem built on top of CocoaPods, by specifying this plugin here, we can integrate the Rome gem. Now, if all goes well, this should pull down Alamofire and then build a dynamic framework instead of creating and integrating a Pods project in Xcode.

However, at the time of writing this book, there appear to be compatibility issues with Swift 3 and Xcode 8 and this gem. So, let's just discuss what we expected to happen. After running `pod install` with the CocoaPods-Rome plugin successfully, there should be a new directory at the root of the project. The directory should be called `Rome` and contain the pre-built `Alamofire.framework` file. We can then use this in Xcode instead of integrating the Pods project.

The next section is going to delve into Carthage and how to integrate the dynamic frameworks that Carthage builds. Integrating dynamic frameworks managed by Carthage will be the same process as integrating frameworks that are built with CocoaPods-Rome. Let's get to it.

Carthage

> *… we created Carthage because we wanted the simplest tool possible—a dependency manager that gets the job done without taking over the responsibility of Xcode.*
>
> —Carthage

Carthage is the last third-party package manager we are going to discuss in this chapter. Carthage is very different from CocoaPods, but it is similar to CocoaPods-Rome. As the preceding quote suggests, Carthage is meant to manage Swift packages and get out of your way. CocoaPods handles the entire Xcode configuration for you when integrating the `Pods.xcodeproj`. However, when using Carthage, you have to handle the Xcode configuration.

Let's use Carthage to manage Alamofire in this section. If you are following along, before we can use Carthage to integrate Alamofire, we must *deintegrate* the `Pods.xcodeproj`. There is a nifty RubyGem called `cocoapods-deintegrate` that can handle this work. Just like CocoaPods-Rome, it is built on top of CocoaPods. This gem can be found on GitHub at https://github.com/CocoaPods/cocoapods-deintegrate. It is very easy to use and I am not going to cover this here.

I have removed CocoaPods from the project I am using. Now, let's start with Carthage. Carthage can be installed via Homebrew. Homebrew is just another package manager, but it manages packages for macOS, not Xcode projects. It's package manager inception. Homebrew can be found at http://brew.sh/. Carthage can be found on GitHub at https://github.com/Carthage/Carthage. I am not going to go over installation here.

After installing Homebrew and Carthage, we can finally get started. First, we need a *Cartfile*. The purpose of the Cartfile is the same as CocoaPods's *Podfile*—to define the dependencies we want Carthage to manage. The Cartfile is far simpler than the Podfile. A Cartfile only requires a reference to whatever Git repository holds the framework you want to pull down. The following is my Cartfile for managing Alamofire:

```
github "Alamofire/Alamofire" ~> 4.0
```

That's all it needs. Now, all I need to run is the `carthage update` command and it will pull down Alamofire. This process will be a little different than a pod install. When Carthage pulls down Alamofire, it then uses Xcode's command-line tool `xcodebuild` to build the `Alamofire.framework` file. After building the framework there will be a Carthage directory and a new `Cartfile.resolved`. The resolved file is the Carthage equivalent of the `Podfile.lock`. This file describes the dependency graph that is required to build the managed frameworks. If Alamofire depended on other frameworks, it would have to resolve those dependencies and build everything.

Now that the Alamofire framework has been pulled down and built in the Carthage directory, we have to integrate this framework in Xcode. This is where CocoaPods handled the magic for us. Carthage however, does nothing to help us integrate frameworks. Let's look at how I can integrate Alamofire in my project.

First, I navigate to the `Alamofire.framework` file in Carthage à Build à iOS and drag and drop it into my Xcode project. It is very important that I keep the *Copy items if needed* option turned off. If I copied the framework into my project, I would not be able to update the framework via Carthage anymore, so I just need a reference. Figure 3-2 shows the result of this action. In the target's settings on the *General* tab, it puts `Alamofire.framework` under the *Linked Frameworks and Libraries* section.

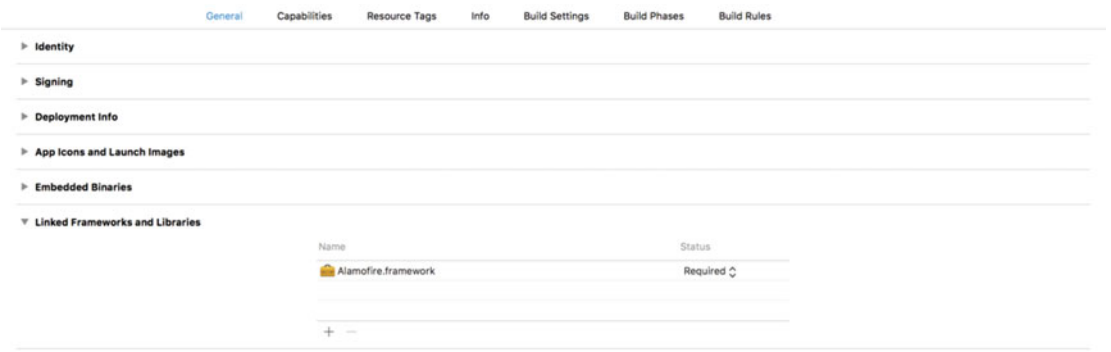


Figure 3-2. *The Alamofire.framework listed under the Linked Frameworks and Libraries section*

After this step, if I `import Alamofire` in one of my files, everything will build and we can run the app. However, we are not done integrating yet. Once the app starts to run, it will crash and give the following error in the console:

```
dyld: Library not loaded: @rpath/Alamofire.framework/Alamofire
  Referenced from: .../CocoapodsTest.app/CocoapodsTest
  Reason: image not found
```

This looks like a crazy error, but we are still missing some pieces, and ultimately the app cannot find Alamofire. In previous versions of Carthage there would still be two steps left. In the recent versions, Carthage handles one of these for us, but I still want to mention it here. Navigate to the *Build Settings* tab under the Target's settings. Make sure the *All* and *Combined* options are turned on and search for **framework search paths**. Figure 3-3 shows the *Build Settings* and *Framework Search Paths*.

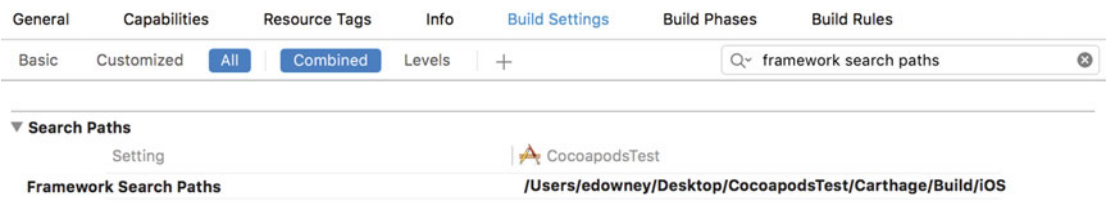


Figure 3-3. *Framework Search Paths under the Build Settings tab*

This setting must be set to the `iOS` directory within the `Carthage/Build` directories so Xcode knows where to find the actual `Alamofire.framework` file. The last step is in the *Build Phases* tab. We are going to add a new phase for the framework. Figure 3-4 shows the steps to adding a new build phase for Alamofire. First, add a new phase by clicking the + button at the top left. Then switch the *Destination* from *Resources* to *Frameworks*. The last step is to drag in the `Alamofire.framework` file from the *Project Navigator* panel.

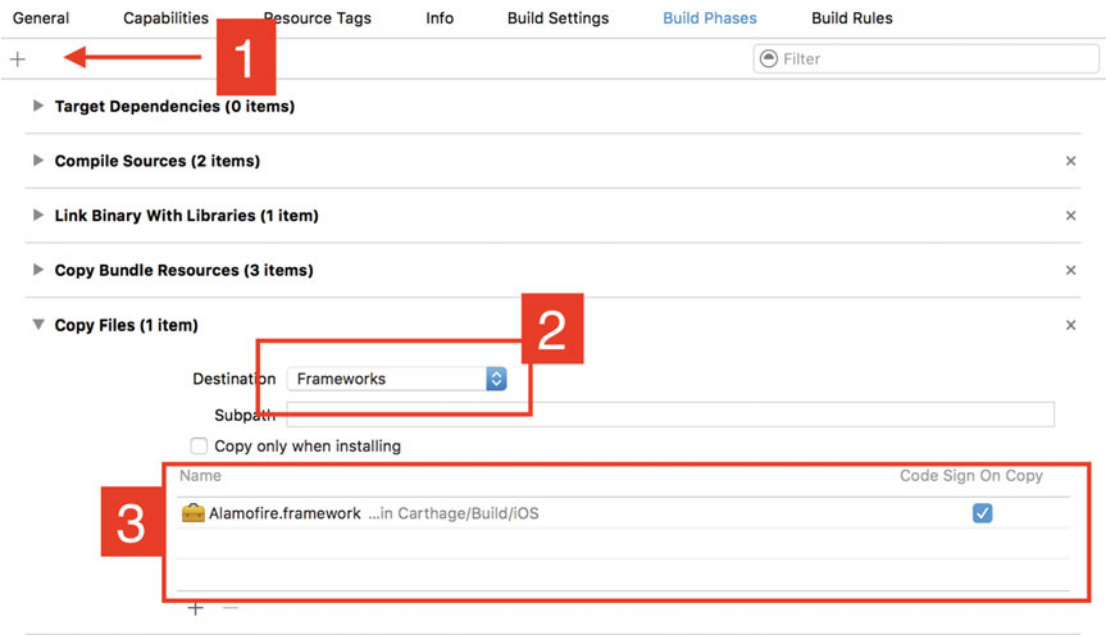


Figure 3-4. The Build Phases tab and the steps to adding a new phase that will copy the Alamofire framework

Now, if I build and run my app, it works! Carthage is a bit more involved than CocoaPods, but we do not have to worry about the configuration CocoaPods does to the Xcode workspace. With Carthage, we get to choose how we manage our project.

That is it for this section. So far, we have seen how to use CocoaPods, CocoaPods-Rome, and Carthage. These package managers are great tools for pulling in third-party libraries for both Swift and Objective-C. Finally, we can talk about the official Swift Package Manager. This package manager is still very new, but the implications are pretty cool. In the last section, we are going to see the Swift Package Manager in action.

Swift Package Manager

This chapter has been all about package managers, but so far all of them have been third-party managers that have been developed by people outside of Apple. The *Swift Package Manager*, which can be found at `https://swift.org/package-manager/`, is the official package manager created by Apple. One of the greatest features with which this package manager provides developers is the ability to write and run Swift code on Linux.

The iOS ecosystem has always been locked down and any code has always required Xcode and a Mac. Releasing Swift as open source and now releasing the Swift Package Manager has changed these restrictions. In this section, we are going to build a Swift package of our own. If you have not been following along, now is the time to start. Chapter 8 is going to

expand on the package we create here. The package we are going to build here is going to be a simple *logger*. Before we jump right in, let's make sure we are set up correctly. Start by opening the Terminal application and typing the following command:

```
$ swift package --version
```

When I run this command, I see the following response:

```
Apple Swift Package Manager - Swift 3.0.0 (swiftpm-19)
```

If you receive any error like the following, you most likely do not have the necessary Swift version or environment set up correctly:

```
<unknown>:0: error: no such file or directory: 'package'
```

You can find the Swift Package Manager on GitHub with installation instructions at https://github.com/apple/swift-package-manager#installation. Once you can successfully run the `swift package --version` command in Terminal, we can start to build our Logger package.

First thing is to create a directory to hold our package. I have created a `Logger` directory on my desktop. In Terminal, navigate to the `Logger` directory and type **swift package --help**. This will output the following:

```
OVERVIEW: Perform operations on Swift packages

USAGE: swift package [command] [options]

COMMANDS:
  init [--type <type>]                    Initialize a new package
      (type: empty|library|executable|system-module)
  fetch                                   Fetch package dependencies
  update                                  Update package dependencies
  generate-xcodeproj [--output <path>]    Generates an Xcode project
  show-dependencies [--format <format>]   Print the resolved dependency graph
      (format: text|dot|json)
  dump-package [--input <path>]           Print parsed Package.swift as JSON

OPTIONS:
  -C, --chdir <path>        Change working directory before any other operation
  --color <mode>            Specify color mode (auto|always|never)
  --enable-code-coverage    Enable code coverage in generated Xcode projects
  -v, --verbose             Increase verbosity of informational output
  --version                 Print the Swift Package Manager version
  -Xcc <flag>               Pass flag through to all C compiler invocations
  -Xlinker <flag>           Pass flag through to all linker invocations
  -Xswiftc <flag>           Pass flag through to all Swift compiler invocations

NOTE: Use `swift build` to build packages, and `swift test` to test packages
```

These are all the commands and options we can feed to the `swift package` command. The very first command is where we are going to start. Let's initialize a new package by running `swift package init --type library`. After running this command, our Logger package should initialized. Let's look at what the previous command created for us:

```
.
├── Package.swift
├── Sources
│   └── Logger.swift
└── Tests
    ├── LinuxMain.swift
    └── LoggerTests
        └── LoggerTests.swift

3 directories, 4 files
```

> **Note** I installed the tree command via Homebrew to get this output from my `Logger` directory.

You can see that the Swift Package Manager uses the name of the containing directory to name the library we created. I chose the `library` type for this package. The `library` option creates a Swift module that we can import in other code. This is not something we are going to build and execute by itself. Initializing our library also gave us two directories, `Sources` and `Tests`. Every Swift package must have these two directories to expose public code and to run tests.

Before we start coding, let's discuss the `Package.swift` file. In the Swift Package Manager, this file is the manifest file and it uses the `PackageDescription` module provided by Apple. This file defines the name of the package, any targets the current package contains, and the dependencies of this package. Here is what our package file looks like for Logger:

```
import PackageDescription

let package = Package(
    name: "Logger"
)
```

Our `Package.swift` file is very simple, but let's look at a more complicated use of this file. The following example is from the documentation on the package manager from `swift.org`. The `Package.swift` file in this example defines a module called `DeckOfPlayingCards` with zero targets, and two external dependencies:

```
import PackageDescription

let package = Package(
    name: "DeckOfPlayingCards",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/apple/example-package-fisheryates.git",
                 majorVersion: 1),
```

```
        .Package(url: "https://github.com/apple/example-package-playingcard.git",
                 majorVersion: 1),
    ]
)
```

The dependencies are defined through the use of this `.Package` type that has `URL` and `majorVersion` parameters. Just like Carthage, the `URL` must resolve to a Git repository. Let's jump back to our Logger package and start writing some code. If you looked closely at the available commands in `swift package`, you would see that there is a `generate-xcodeproj` command. Run the following command in Terminal at the root of the `Logger` directory:

```
$ swift package generate-xcodeproj
```

After running this command, you should see the following response in your Terminal:

```
generated: ./Logger.xcodeproj
```

We now have an actual Xcode project that is configured to build our Logger package. Open this up and let's start coding. In the `sources` directory, there is already a file and struct defined for us. The generated code looks like the following:

```
struct Logger {


    var text = "Hello, World!"
}
```

Let's leave this as a `struct`. If you are unclear on the difference between a `class` and `struct`, do not worry. We are going to cover this topic in Chapter 5. The first thing we can do is add a `LogFilter enum` to the `Logger.swift` file:

```
public enum LogFilter {
    case Info
    case Severe
}

public struct Logger {

    var text = "Hello, World!"
}
```

I have also marked the `enum` and the `Logger struct` as `public`. Way back in the "Dynamic Framework" section we talked about Swift modules and their access control. If we want to use these outside the logger module, we have to use `public`. Let's fill out the rest of the implementation for our logger. We need an initializer that defaults a severity for the log filter. We then need to expose some way for the users to actually log something. Here is the code:

```
public enum LogFilter {
    case Info
    case Severe
}
```

```
public struct Logger {

    public var severity: LogFilter

    public init() {
        severity = .Info
    }

    public func log(item: String?, withSeverity severity: LogFilter) {
        if self.severity == severity {
            print(item)
        }
     }
}
```

I know this is extremely simple, but we are going to expand on this later in the book. The next thing we need to do is to build our package. Let's go back into Terminal and run the following command:

```
$ swift build
```

This will generate a `.build` directory at the root of `Logger`. This command builds our package, generating a lot of files. Here is the tree of the `.build` directory:

```
.
├── build.db
├── debug
|   ├── Logger.build
|   |   ├── Logger.d
|   |   ├── Logger.swift.o
|   |   ├── Logger.swiftdeps
|   |   ├── Logger~partial.swiftdoc
|   |   ├── Logger~partial.swiftmodule
|   |   ├── master.swiftdeps
|   |   └── output-file-map.json
|   ├── Logger.swiftdoc
|   ├── Logger.swiftmodule
|   ├── LoggerPackageTests.xctest
|   |   └── Contents
|   |       ├── Info.plist
|   |       └── MacOS
|   └── ModuleCache
|       ├── CGC1TPJT37OH
|       |   └── SwiftShims-1HJGLIW7H35BO.pcm
|       └── modules.timestamp
└── debug.yaml

7 directories, 14 files
```

Buried in this tree is the `Logger.swiftmodule` file. This is the file that represents our built Swift module. If we switch back over to Xcode, it will build a `.framework` file. The framework is just another Swift module, but it works with Xcode. Figure 3-5 shows Xcode's Project Navigator panel for our Logger package. Under the products group, we have a `Logger.framework` file.

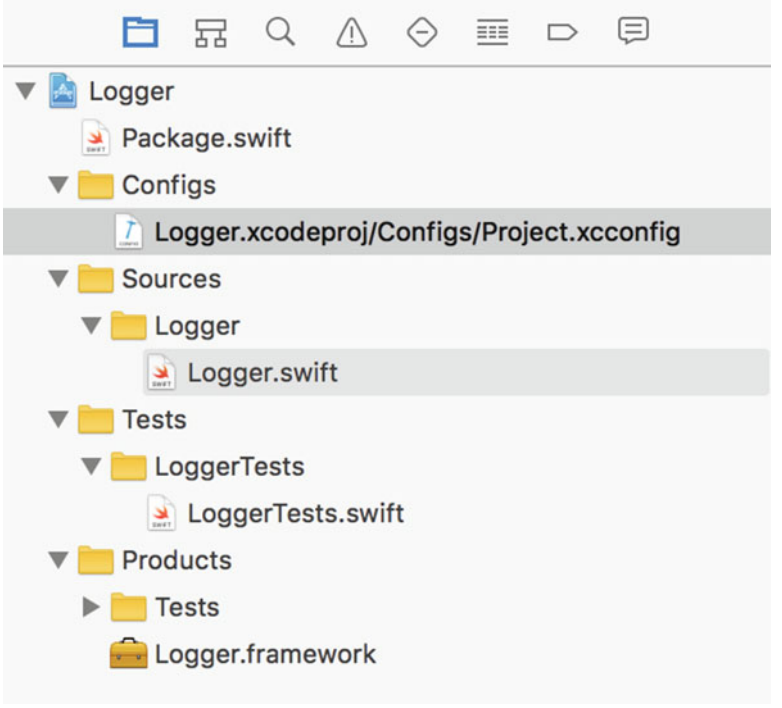


Figure 3-5. *The Project Navigator panel for the Logger package in Xcode*

And that's it. We have just built our first package with the official Swift Package Manager. This section has been a quick introduction to the new package manager, but hopefully it gives you some idea of what the future of Swift might be like. If you want to keep working on this package, we will be adding more code and even tests in Chapter 8, so stay tuned.

Wrap Up

That's a wrap for Chapter 3. We have discussed many different topics in this chapter including static libraries and dynamic frameworks, and the third-party package managers CocoaPods, CocoaPods-Rome, and Carthage. We then built our very own Swift module with the official Swift Package Manager by Apple.

After you read the first section on static libraries and dynamic frameworks, I hope you know the differences between the two. We discussed the idea of Swift modules here, which is how all Swift apps and frameworks are treated. Afterward, we started to look at package managers including CocoaPods, CocoaPods-Rome, and Carthage. If you are using third-party frameworks/code in your project, I would highly recommend evaluating these tools to see if they can benefit you. After this chapter, I hope you can pick the right package manager for your project.

And finally, we have the Swift Package Manager. We built a small Logger package that we are going to revisit in Chapter 8 after you have some more chapters under your belt. Hopefully, you can see the power of the Swift Package Manager and how it can be the start of Swift running everywhere. The idea of Swift code in more areas than iOS and macOS is a really awesome idea that I hope is the future of Swift. Up next, we are going to discuss design patterns specifically in iOS.

Chapter 4

iOS Architecture

And… we're back. This chapter will discuss software architecture and design patterns in iOS. This is the first chapter where we will use Xcode *playgrounds*. Playgrounds are amazing! If you have not used them yet, this is going to be a good introduction. If you have, then you know what I am talking about. Playgrounds can have multiple pages and include a markup system for rendering comments and annotations. They have the potential to be interactive books.

I started a playground about a year ago and I have been adding more to it since. I have about 30 pages of code and markup. It helps me keep my examples, ideas, and the problems I have figured out in one central location. This is invaluable because before playgrounds, any time I wanted to try new code, I had to start a new project, and I always lost track of it. Starting a new project was also kind of heavy handed or the simplicity of the code. Playgrounds are much more lightweight and quick. This is why we will be building a playground reference throughout the first section of this book. We will then refer to this playground in the second half when building our app. I also hope you not only create this reference from the book, but do more research and expand on it.

Now, entire books have been written on the subject of architecture. There is so much to this topic that it cannot all fit in this chapter. So, this chapter is going to look at a small set of design patterns that will be used later. I have found that these patterns provide huge benefits for their simplicity. We will then look at the patterns that we should avoid, called *AntiPatterns*. Let's get started.

What You'll Learn

This chapter will discuss Xcode's playground system along with the playground markup syntax. I would highly recommend more research into this topic. Providing comments for the code in playgrounds can be invaluable when you are keeping a long-running playground alive. We are just going to examine the basics.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_4

Afterward, we will use our playground to create examples of a few design patterns: dependency injection, MVC, MVVM, presenter, and singleton. These patterns are very simple in nature, but the benefits can be clean and concise code. These design patterns can be categorized as structural patterns. This means they describe the overall structure of an application. There are many other design patterns that can be used for smaller problems. We will not examine these design patterns since they are so situational.

After examining the structural design patterns we can use, we will examine how these patterns can be used effectively and how to determine if a pattern is necessary or over-engineering. Over-engineering can be a struggle. It is so hard to determine when enough is enough. We are going to examine this problem in a more theoretical way rather than through code.

After examining all of these design patterns, we will look at the patterns that we want to avoid. These patterns are called *AntiPatterns* and they can destroy a codebase. There's a lot to cover so let's get started!

> **Caution** Avoid AntiPatterns like the plague. ☺

Playgrounds and Markup Syntax

This section is going to give you an overview of Xcode's playground markup syntax. There is a lot to the markup syntax and we will not cover everything here. First, let's talk the about the basics—the single-line comments versus the block comments.

> **Note** Everything you want to know about playgrounds' markup syntax can be found at the following link: https://developer.apple.com/library/ios/documentation/Xcode/Reference/xcode_markup_formatting_ref/index.html.

```
//: # This is a single line Header

/*:

 # Hello
    - First Point
    - Second Point

*/
```

The first comment is a single-line comment, whereas the second is the block syntax. The # symbol describes a header. The more # symbols, the smaller the header, like the `h1`, `h2`, and `h3` tags in HTML. As you might have guessed, in the block comment, we have a list, but playgrounds add a little extra for us. Figure 4-1 shows how these two comments are rendered.

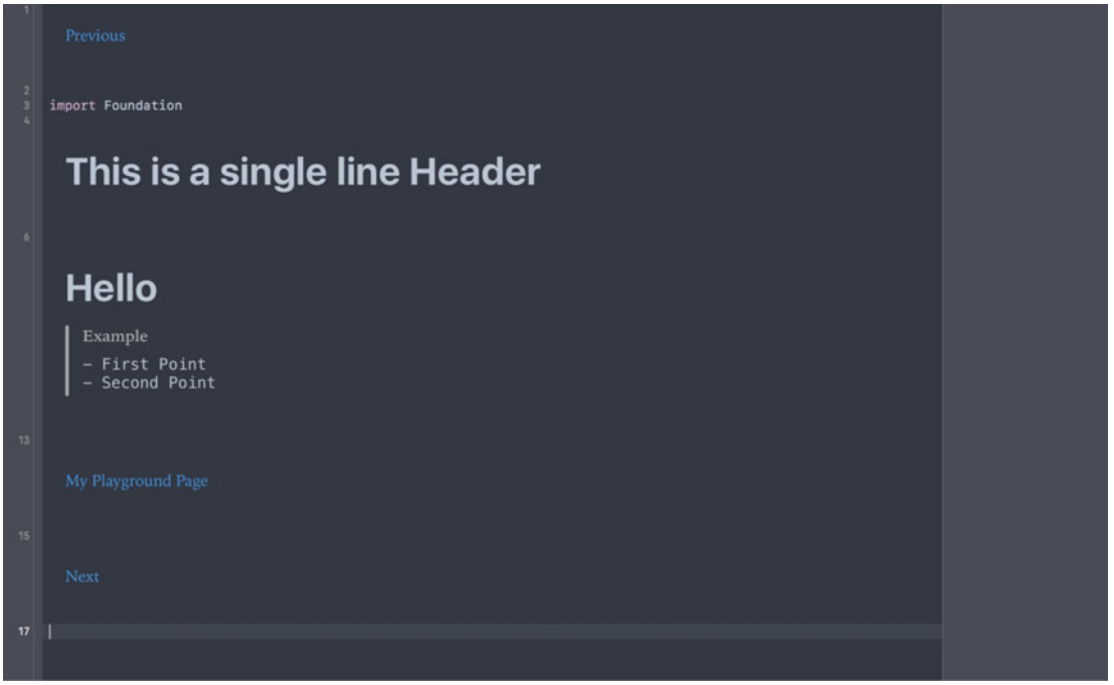


Figure 4-1. *The rendered markup from the previous two comments*

You can see that the playground renders this with two headers and our list. It also denotes this section as an "Example." There is a lot more you can add to this. You can add *callouts,* which are described here: https://developer.apple.com/library/ios/documentation/Xcode/Reference/xcode_markup_formatting_ref/Attention.html. The next thing you will want to know about is *links*. You can add links in your markup to other pages within your playground. This is really cool! The syntax for a playground page link is as follows:

```
//: [My Playground Page](@PlaygroundPage)
```

First, the brackets denote the display name for the link and the parentheses (using the `@` symbol) denote the actual name of the playground page. You can also create `next` and `previous` links for the next and previous pages that do not need the specific name of the playground page.

The last features we will talk about in this section are image and video assets. Playground pages can contain images and videos directly in the source. In the left pane of the playground, you can see a folder reference called *Resources*. You can put images and videos here and reference them in markup. The syntax for resources is very similar to the page link syntax. Square brackets contain the display name for the resource and the parentheses contain the actual name of the resource. The parentheses can also contain HTTP links to external resources.

The last thing we need to discuss is the format this book is going to use for our playground reference. The section header is going to contain the name of the corresponding playground page. The following text details a sample header and playground page.

Sample Header

In the previous example, the section "Sample Header" would correspond to the playground page named *Ch04—Playground Reference*. This section would either add more code to that page or it would mean you needed to create the new playground page for this section. Sections that do not correspond with a playground page will have the following subheading: *Playground Page: N/A*. We might use the same playground page for more than one section, so be sure to pay attention to the name of the page.

Also, the code blocks that we are going to add to our playground will be marked with the name of the page like the following:

```
var message: String = "Yay Playgrounds!"
print(message)
```

If the code box appears without a playground page, that means the code is meant for reference but should not be added to your playground page.

Xcode playgrounds are very useful and have the potential to be interactive books. As Swift grows in popularity, I hope to see full books published using playgrounds. Next up, we will start our discussion of design patterns by first describing what a design pattern is.

Design Patterns

> *A design pattern is a general repeatable solution to a commonly occurring problem in software design.*
>
> —sourcemaking.com

The preceding quote describes the purpose of *design patterns* in software. They are meant to be a common solution that is not tied to any particular system or technology. There are many different types of problems in software development, so there are many different types of design patterns. We are going to cover four of the most common design patterns: dependency injection, MVC, MVVM, presenter, and singleton.

The goal of this chapter is to set us up for success when we go to build our app, but also to set you up for making architectural decisions in your everyday development. We can be the best Swift developers ever, but if we do not understand when to use design patterns and more importantly, when ***not*** to use design patterns, we will not succeed. Let's first start with dependency injection.

Dependency Injection

First up, *dependency injection.* This pattern can sound kind of scary at first. When I first heard it used in a discussion, it sounded like some crazy, complicated thing that I would never understand. This could not be further from the truth. In fact, you have used dependency injection before. Anytime you pass a value to a method or function, you are using dependency injection.

The first thing you will learn in researching this pattern is that it can take a few different forms. First, let's examine what your code would look like without dependency injection. This is where we will start adding code to our playground. If you have not done so, please create a playground now. Start a new page, and add the following code:

```
//: ## Dependency Injection

//: #### Without D.I.

class Service {
    func doSomething() {
        print("hello")
    }
}
```

Now, in this first example, we are going to *not* use dependency injection and see what problems arise. In the previous code block we have a `Service` class that has a `doSomething` method. Next, we are going to define the consumer of this class:

```
class Client {
    let service: Service

    init() {
        service = Service()
    }

    func startSomething() {
        service.doSomething()
    }
}

let client = Client()
client.startSomething()
```

Now, we have declared a `Client` class, which has a dependency on our `Service` class. You might notice that the `service` member is declared with the `let` keyword. Once we set our `service` in the initializer, we cannot assign it again. The initializer also does not allow a `service` object to be injected.

This might not seem like that big of a deal, but imagine you are writing a framework. If someone wanted to use your `Client` class, but subclass and override your `Service` class, they would be unable to do so. The problem here is that we have written code that is not extensible. That is the ultimate goal of design patterns. When we are using design patterns properly, we open our code up to be extensible and scalable. The previous example is neither.

Let's reexamine this with dependency injection:

```
//: #### With D.I.

class DIClient {
    var service: Service
```

```
    init(service: Service = Service()) {
        self.service = service
    }

    func startSomething() {
        service.doSomething()
    }
}
```

Here, we are using the same exact `Service` class as in the earlier example. Since we are on the same page in our playground, we cannot redeclare the `Client` class, as doing so would generate a compile error. Now we have a dependency injection–capable class called `DIClient`. There are two differences here that open up the implementation.

First, we have changed our declaration of `service` to be a *variable* instead of a *constant*. This allows a user of `DIClient` to inject a `service` object via the `service` variable. This is called *setter injection*. The second form of dependency injection we have is in the `initializer`, called *constructor injection*. Before, we just instantiated a `service` object; now, we are allowing one to be injected as a parameter and we have given this a default parameter. The purpose of the default parameter is to allow the class to instantiate its own `service` object to save the user the trouble if it uses the base `Service` class. Therefore, instantiating one of these objects is the same as before, but we have opened ourselves to be extensible.

One of the more complicated forms of dependency injection is called *Inversion of Control (IoC)*. This is a really fancy name and it's what most dependency injection frameworks are doing. An IoC container's purpose is to hold a dependency and the knowledge of how to create said dependency. You can then use the container to create your dependencies where you need them. This can be very useful because a class can register itself as a dependency and it knows how to instantiate itself and what dependencies it requires. That means that all the logic associated with a particular object is all contained within the object itself, instead of spread across your app. Let's look at some code:

```
//: #### Inversion of Control Container

class Container {
    typealias Closure = (Void) -> AnyObject
    var registry: [String: Closure] = [:]

    func register(name: String, for type: @escaping Closure) {
        registry[name] = type
    }
}
```

The previous code block declares a class called `Container`. This class holds a dictionary of closures that can be registered on the class by name. These closures then have a return type of `AnyObject`, which is the dependency. We have our container, so how can we use this? Well, we are going to need one more function. We need our container to be able resolve dependencies by name.

```
//: #### Inversion of Control Container

class Container {
```

```
    typealias Closure = (Void) -> AnyObject
    var registry: [String: Closure] = [:]

    func register(name: String, for type: @escaping Closure) {
        registry[name] = type
    }

    func resolve(name: String) -> AnyObject? {
        return registry[name]?()
    }
}
```

We now can register dependencies and then resolve them all through this `Container` class. Here is an example of usage for this class:

```
let container = Container()
container.register(name: "Service") {
    Service()
}
container.register(name: "Client") {
    DIClient(service: container.resolve(name: "Service") as! Service)
}
let result = container.resolve(name: "Client") as? DIClient
result?.startSomething()
```

All right, we instantiate a container and register our `service` class. We then register the `client` class and we also resolve the `service` class. After everything has been registered, we then resolve the client. This action should run the closure for the `client` object, which then runs the closure for the `service` object. Our dependencies are now resolved and we can then use the result to `startSomething`.

We have just created an IoC container. This is a really cool class, but it can be hard to see its power here. Imagine you had a class that required a bunch of other properties to initialize, but they were not readily available at the correct point in time. You can register a closure that instantiates your type, then use that closure elsewhere when you need it.

Awesome! Dependency injection is a really cool pattern. We have talked about the many forms it can take like setter injection, constructor injection, and even an IoC container. Next up, we are going to discuss MVC.

MVC

MVC is the most widely used design pattern; you have already used it before because it is built right into iOS. MVC has three categories of objects: model, view, and controller. UIKit is packaged with `UIViewController`, Xcode has Storyboard/Nib files as your view, and the model is just the object that represents your data. Your view could also be a subclass of `UIView` depending on how you have set things up. The general structure and communication of MVC is described in Figure 4-2. It has the model and view on opposite ends with the controller set directly in the middle coordinating the communication.

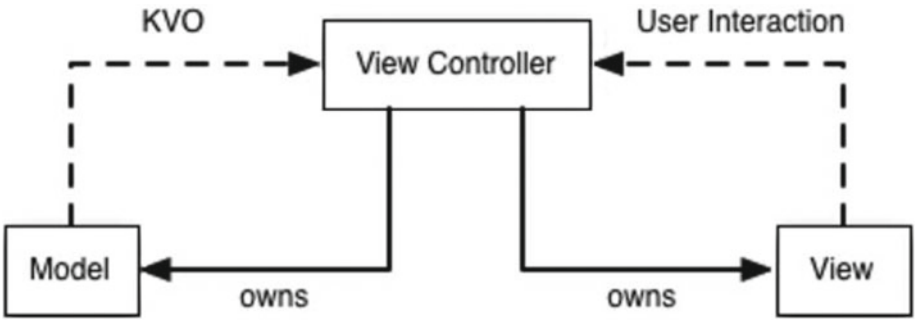


Figure 4-2. *Describes the communication in a system built with MVC architecture. Figure retrieved from Ash Furrow, Objc.io at* `www.objc.io/issues/13-architecture/mvvm/`

The benefit of MVC is reusability and extensibility. In iOS, you should be able to represent any component as these three parts. Each view has a controller and a model behind it. This separation can allow you to switch out a model and reuse the same exact view. This pattern is also extendible because your app is just a repetition of these three components over and over again. There is not much more to say about MVC. It is widely adopted, there is a lot of documentation online, and you have most likely already used it if you have built any iOS applications before. Next up, we have MVVM, a very cool pattern with roots in MVC.

MVVM

> *Ever heard of MVC? Massive View Controller, some call it.*
>
> —Ash Furrow, objc.io

As the quote above suggests, there can be some problems with MVC. This is where we can start to talk about when not to use patterns. MVC is basically built into iOS, but this does not mean it is the end all, be all pattern. I like to think of MVVM as an evolution of MVC. So, what is *MVVM?* It stands for model, view, view model. Just like MVC, MVVM is describing the layers of the design. However, one key point is left out. Remember, `UIViewController` has to be used in iOS to communicate with the view layer. Figure 4-3 describes how MVVM can look in an iOS application.

The preceding quote is from an article by Ash Furrow at Objc.io called 'Introduction to MVVM." Figure 4-2 and 4-3 are also from that article. Let's break down both the quote and the diagram in Figure 4-3. First let's talk about the quote. Massive view controllers are a problem. You may have seen them or even created a few in your time. I know I have. Eventually, you might get to a point where you are reimplementing something from another view controller, or you are breaking code out into helpers to manage the size and complexity. This is only a Band-Aid though, because there is a structural issue going on in your app. The structural issue is that there is not enough structure.

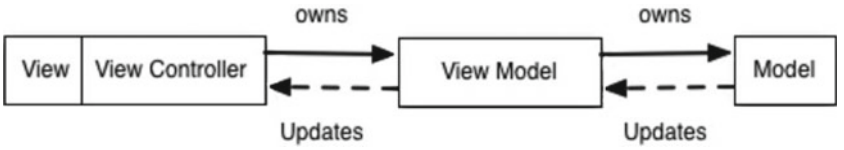


Figure 4-3. *Describes the interaction and communication in the MVVM design pattern. Figure retrieved from Ash Furrow, Objc.io at* www.objc.io/issues/13-architecture/mvvm/

Now, let's discuss the diagram. You can see Ash has coupled the view layer and the view controller layer together. He has done this because, all too often, view controllers and the views they control go hand in hand. Theoretically, you should be capable of switching a view controller with another view, depending on the data behind the view. This is hardly ever the case, which is why he has coupled these two pieces together. The next part of the diagram is the view model. This is in between the view controller and the model. The job of the view model is to take the model and transform it into something the view controller can use. The example Ash uses is a model that contains a `Date` object. When the view controller asks for the model's `date`, the view model transforms it into a properly date-formatted `string`. Let's see how this code looks in action:

```
class MyModel {
    var date: Date = Date()
}

class ViewModel {
    var model: MyModel

    init(model: MyModel) {
        self.model = model
    }
}
```

All right, we have created a `MyModel` class and a `ViewModel` class. Notice we are using dependency injection here with the model. Now, let's add the `date` example Ash was talking about:

```
class ViewModel {
    var model: MyModel
    var dateFormatter: DateFormatter

    init(model: MyModel) {
        self.model = model
        self.dateFormatter = DateFormatter()
        self.dateFormatter.dateStyle = .short
    }

    var modelDate: String {
        return dateFormatter.string(from: model.date)
    }
}
```

This is a fairly simple example, but a good illustration of this pattern. We create a `dateFormatter` in the `init`, and then in a computed property at the bottom, we `return` a formatted date string. Now, let's see what our `UIViewController` would look like with this `ViewModel` attached.

> **Note** You will have to import UIKit in your playground to use `UIViewController`.

```
import UIKit

class MyViewController: UIViewController {
    var viewModel: ViewModel?
    var model = MyModel()

    @IBOutlet var dateLabel: UILabel?

    override func viewDidLoad() {
        super.viewDidLoad()

        viewModel = ViewModel(model: model)
    }

    @IBAction func setDate() {
        dateLabel?.text = viewModel?.modelDate
    }
}
```

Let's walk through this view controller. First, we have an optional `ViewModel` and a `MyModel` defined. Next, we have a label called `dateLabel`. In the `viewDidLoad` method, we then set up our `ViewModel` and inject the model. Finally, we have some `@IBAction` that is hooked up in our storyboard and it just sets the `dateLabel`'s text to the formatted date string from our model. You can see how the controller layer does not communicate with the model layer. Instead the controller layer communicates with `ViewModel`.

Now, the idea is we can switch out the `ViewModel` and the `MyModel` without disturbing the controller and the view. Now, it might not always work out this way, so here is how I like to think about my view models. The view model is responsible for the business logic. The business logic is anything that needs to happen to your data that is not associated with your view. This is what I personally associate with the `ViewModel` layer. Then you are capable of breaking out your presentation logic.

These are the basics of MVVM. It is a simple pattern, but it can really clean up your codebase and reduce massive view controllers to a manageable size. Next up, we are going to look at the presenter pattern. This is a fun pattern that can be used with MVC or MVVM.

Presenter

The previous two patterns have been more about the base structure for an application. The *presenter* pattern can be a little bit different. This pattern has been associated with MVP and VIPER. MVP stands for *model-view-presenter*. VIPER stands for *view-interactor-presenter-entity-routing*. The presenter pattern has also shown up in MVVM as MVVMP.

When described in articles, the presenter shows up in between the controller layer and the model layer. It can also show up in between the view controller and the view model in the case of MVVMP. It can be very similar to the `ViewModel`. Its purpose is supposed to be to convert the model data into a presentable form. This seems very similar to the purpose of the `ViewModel`. This is why I like to think about the presenter as a sort of modifier.

In iOS, things can be very heavily tied to the view and controller. This is what Figure 4-3 described. In the section on MVVM, I described how I like to think of the `ViewModel` as *business* logic and not *presentation* logic. I like to think the presenter is meant to be the yin to the `ViewModel`'s yang. The presenter will house the presentation logic, whereas your `ViewModel` contains the business logic. These two layers would then never talk directly to each other but instead through the view controller acting as a coordinator. The presenter pattern can take different forms based on other patterns being used. This makes the pattern a bit tricky. Now that we have gone through what the pattern is, let's see how it looks in code.

First, let's create a very simple presenter class to illustrate the pattern:

```
//: ## Presenter

class Presenter {
    weak var view: UIView?

    init(view: UIView) {
        self.view = view
    }
}
```

This is about as simple as it gets. All this class contains is a weak reference to a `UIView`. The idea is this `UIView` is a reference to the `UIViewController`'s `view` property. You can see we have used dependency injection here with the `UIView`. Now, let's look at a view controller that uses one of these:

```
class ViewController: UIViewController {
    var presenter: Presenter?

    override func viewDidLoad() {
        super.viewDidLoad()
        presenter = Presenter(view: view)
    }
}
```

This is very similar to our `ViewModel` example. We just instantiate a presenter and inject the view controller's `view` property. Now, let's add a `UIAlertController` to display some information. Our presenter class will now look like the following:

```
class Presenter {
    weak var view: UIView?

    init(view: UIView) {
        self.view = view
    }
```

```
    func displayAlert(on viewController: UIViewController, withTitle title: String?,
    message: String?, andActions actions: [UIAlertAction]) {
        let alert = createAlert(withTitle: title, message: message, andActions: actions)
        viewController.present(alert, animated: true, completion: nil)
    }

    func createAlert(withTitle title: String?, message: String?, andActions actions:
    [UIAlertAction]) -> UIAlertController {
        let alert = UIAlertController(title: title, message: message, preferredStyle:
        .alert)

        actions.forEach { alert.addAction($0) }

        return alert
    }
}
```

We have added a `createAlert` method that returns a `UIAlertController` and a `displayAlert` method that takes the view controller and displays the alert. These methods are responsible for creating and displaying an alert based on the given information. Next, let's use this functionality in our controller:

```
class ViewController: UIViewController {
    var presenter: Presenter?

    override func viewDidLoad() {
        super.viewDidLoad()
        presenter = Presenter(view: view)
    }

    @IBAction func userDoesSomething() {
        let action = UIAlertAction(title: "OK", style: .destructive, handler: nil)
        presenter?.displayAlert(on: self, withTitle: "Error", message: "You did something
        bad", andActions: [action])
    }
}
```

The new `@IBAction` in our controller uses this functionality to display an alert. You can see how simply displaying an alert is using this pattern. Not to get ahead of ourselves, but you might also be able to see how testable this pattern can make our code, but more on that in Chapter 8. That is pretty much it for the presenter pattern. It can be very simple, but it can also take multiple forms depending on the structure around it.

Singleton

The singleton pattern is a very common pattern in iOS development. Singletons are used extensively in Apple's frameworks such as the `UIApplication`, `NSFileManager`, and `NSNotificationCenter`. The purpose of a singleton is to provide a global point of access to the object and to guarantee there is only ever one instance of the object. The `UIApplication` can be accessed anywhere in our apps by using `UIApplication.shared`.

These objects make sense to be singletons because they hold global application state. Therein lies the problems that can arise with singletons, though. Global state can be very difficult to manage and even harder to debug. This is why many people avoid creating singletons in their applications. This section is going to cover how to build a singleton in code, and then we will continue our discussion of advantages and disadvantages.

Let's flip back over to our playground reference and create a singleton:

```
//: ## Singleton

class MyAppState {
    static let instance: MyAppState = MyAppState()
}
```

In the preceding code block, we have defined a new class called `MyAppState`. This class has a static variable called `instance`. The static keyword on a variable creates a type property. *Type properties* are global values that are associated with the specific type. It might not seem like much, but that is all that is required to define a singleton in Swift. Let's add a method to our class to print a string:

```
class MyAppState {
    static let instance: MyAppState = MyAppState()

    func handleSomething() {
        print("Singleton working here...")
    }
}
```

Now, let's use our singleton and its new functionality:

```
MyAppState.instance.handleSomething()
```

Instead of instantiating a new object of type `MyAppState`, we just access the `instance` property and we can call `handleSomething`. If you have ever used singletons before, you are probably familiar with this style. You can see how easy they are to define. The only difference between a singleton and a normal class is the singleton is always accessible.

This is why we have to be careful with this pattern. It is very easy to define singletons and then access/use them everywhere. We could use this pattern instead of MVVM. If we encapsulated all of our data logic in a singleton, there would be no need to create a view model. The view controller could then access the app's data through the singleton. There is nothing necessarily wrong with this pattern, but we must consider the downside.

First, we are coupling code with the singleton, which makes the architecture rigid. We cannot extend any code that uses a singleton, because there will only be one singleton. We also have to consider testing. Since a singleton controls its own lifecycle, we run the risk of the singleton influencing our tests differently for different runs. This is called *idompotency*.

This term in the context of testing means our tests should produce the same results for every run. Since the purpose of singletons is to hold global state, it is possible for the state of a singleton to change during a test and cause a previously passing test to fail even though the underlying code being tested has not changed.

I do not mean for this to deter you from using singletons altogether. They still have their place, especially in iOS. This section has looked at how we can define singletons and some common pitfalls to using them. That will do it for our discussion on design patterns. I highly recommend more research into these patterns as their definitions and responsibilities can be a bit murky. The entire goal of design patterns is to clean up code. As developers, we want to write scalable, extensible code. Design patterns have been proven to accomplish this task. However, when are you using too many design patterns; how can you know? Read on!

When Is It Too Much?

> *The KISS principle states that most systems work best if they are kept simple rather than made complex; therefore simplicity should be a key goal in design and unnecessary complexity should be avoided.*
>
> —effectivesoftwaredesign.com

We just discussed four great design patterns and how they can really clean up your code. Now, let's talk about how they can be a detriment to your project. It can be very easy to fall into situations in which design patterns can be used and you say, "Why not?" You might start your iOS app out using MVC, but then start to have big view controllers, so you decide to refactor to MVVM. However, this might be overkill. You might be able to get away with just using a presenter or some other, smaller pattern.

This is the problem with design patterns. How can you know when to use them and when not to use them? When too many design patterns start to get used in your application, it can be hard to spot. Each pattern has its purpose and, logically, it can make sense. However, when you are combining multiple patterns, applications can grow to nearly unmanageable sizes. I believe it is possible to see when too many design patterns are used in an application, but the argument against them can be difficult to formulate. Logically, it might make sense for certain design patterns to be used, so this is how I like to approach this problem. Keep it simple. If this sounds familiar it should. There is a principle of software development called KISS. It stands for "Keep it simple, stupid."

AntiPatterns

This will be the last major section in this chapter and it is very important. This section will be describing *AntiPatterns*. As you can imagine, AntiPatterns are the opposite of design patterns. They take your scalable, extendable code and ensure it is difficult to understand and change. I have selected two AntiPatterns to discuss, the God class/Blob and Poltergeists. This section is not going to contain any code. I will not help you code AntiPatterns. Instead, it will be a discussion about the AntiPatterns and how to spot them in the wild. Let's get started.

The God Class and the Blob

> *[A God/Blob object]… leads to one object with a lion's share of the responsibilities… solution includes refactoring the design to distribute responsibilities more uniformly…*
>
> —sourcemaking.com

This AntiPattern is tied into a principle of software design called the *single-responsibility principle*. This principle is the first in the SOLID principles that were created by Robert Martin. The principle states, "A class should have only one reason to change." This AntiPattern is the opposite of this principle. The quote above states this God/Blob object can have a bunch of responsibilities and this can lead to very bad design. This type of object can be very difficult to reuse and can be so complex and disjointed, it is hard to understand.

So, how can you be sure you do not create any blobs and how can you spot them if they already exist? Most of the time, an object such as this is hard to name. If you were to create a `UIViewController` that lists a set of data, you might call it `DataListViewController`. Imagine what you would call an object that has 20 different responsibilities. This is the first sign and they usually wind up being called `helper` or `utilit`(y)(`ies`)

> **Note** Not all objects called `helper` or `utility` are implementations of this AntiPattern. Targeted, simple `helpers` and `utility` classes can be created and used and can be very effective. The problem is when there is one `helper` or `utility` object to rule them all. Don't be Frodo.

This AntiPattern does not just appear as `helpers` or `utilities`. A base class can just as easily become a Blob/God class. In objected-oriented design (OOD), inheritance is used heavily. There can be issues with OOD, which we will discuss in the next chapter. When you start merging base classes to satisfy inheritance, most likely, that object is becoming a God class. The next AntiPattern we will discuss is Poltergeists.

Poltergeists

> *Poltergeists are classes with very limited roles and effective life cycles.*
>
> —sourcemaking.com

As the quote and name of this AntiPattern suggests, it is all about the life cycle and responsibilities of the object. They are very limited and you might wonder what the object is even doing. Let's examine a form of this AntiPattern in terms of building a networking layer. Let's say you break out your networking code in two halves. One half is responsible for coordination with the rest of the app, while the other half is responsible for actually making the networking requests.

You can imagine the first half talks with the second half and vice versa to complete one whole network transaction. A poltergeist that could appear in this scenario is the communication between the two objects. What would the communication look like

now? The first half instantiates an object, whose entire responsibility is to handle the communication back to the rest of the app. The first half then sends this object to the second half along with the original request. The job of the second half is still the same and makes the network request. When the network request completes, instead of communicating back to the first object, it uses the poltergeist to communicate back. What exactly is the responsibility of the poltergeist object? It is taking over a small portion of the first network object's job. The object is also only alive for the single network request. This is a lot of overhead for something the original object should be capable of doing.

Poltergeists are small, can add unnecessary complexity, and violate the principle of single responsibility, as they have no real responsibility. AntiPatterns can be hard to track down. They can also be difficult to fix. Moreover, the problem is not just a code issue. AntiPatterns are hints at deeper problems. Please be aware that there are a lot of other AntiPatterns out there and, just like design patterns, they can be subjective in nature.

Wrap Up

This has been a really fun chapter to write. I have a love/hate relationship with architecture. It is often more subjective than objective and this can make it difficult to discuss. Please note that this chapter is based on architecture I have used and what I have seen work and not work. Take this with a grain of salt and figure out what is best for you. Experiment with architecture and figure out your best practices and approaches. I hope you have gained some knowledge in this chapter and you continue to do your own research. Architecture will continue to be a theme in the first half of this book and will be heavily focused on when we go to build our app. The next chapter is going to be about protocol-oriented programming.

Articles

1. iOS Architecture Patterns
   - https://medium.com/ios-os-x-development/ios-architecture-patterns-ecba4c38de52#.ne1su4f9w
2. Introduction to MVVM
   - www.objc.io/issues/13-architecture/mvvm/
3. From MVC to MVVM in Swift
   - http://rasic.info/from-mvc-to-mvvm-in-swift/
4. Design Patterns
   - https://sourcemaking.com/design_patterns
5. AntiPatterns
   - https://sourcemaking.com/antipatterns
6. Dependency Injection in Swift
   - https://medium.com/ios-os-x-development/dependency-injection-in-swift-a959c6eee0ab#.me0i6nloq

Chapter 5

Protocol-Oriented Programming

This chapter is going to walk you through *protocol-oriented programming,* which is my favorite paradigm in the Swift language. We will go over what protocols are, how they work, and how you have already used protocols. We will look at interfaces and protocols to see how these concepts have been implemented in Swift and other languages, and this will lead us into the concepts of *traits* and *abilities*.

Next, we will look at an object-oriented example that is better suited to be protocol oriented. This objected-oriented example will touch on a few problems that are common in an objected-oriented world. We will also take a look at Swift game development using `SpriteKit`. Game development is a really fun topic and we will really be able to see the power of protocol-oriented programming here.

Finally, we will wrap up the chapter talking about testing. Testing our code can be a whole lot easier when we use protocols properly. I am also going to introduce a concept for testing that I will go into depth on in Chapter 8. Let's get started!

What You'll Learn

This chapter is going to discuss Swift protocols. We are going to examine the difference between interfaces and protocols. If you think there is no difference, you're in the right place. The core of this chapter is about the concepts of traits and abilities. Once we have an understanding of these concepts, we will move on to how we can think in a protocol-oriented manner.

Thinking in a protocol-oriented way is the biggest hurdle when you're first starting protocol-oriented programming. It can be difficult to transition from pure inheritance with objected-oriented programming to protocols. Apple has also added the discussion of reference types and value types, but more on this later. We will extend our Xcode playground reference with an object-oriented problem and discuss how we can rework our solution to achieve a protocol-oriented result.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_5

In the last section of this chapter, we are going to cover testing with protocols. The idea of this section is abstraction. We will look at how we can test code that uses the `UIApplication` singleton. Adding protocols will make this previously impossible-to test-code, testable. Protocols can make testing easier, when used correctly. Let's start this chapter by discussing what protocols are.

What Are Protocols?

> *A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.*
>
> —The Swift Programming Language

"A blueprint of methods, properties, and other requirements…." This is taken directly from Apple's documentation on the Swift language from their Protocols section. The link to their documentation is https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html. I want you to read this entire quote a few times. This next part took me a few readings. Also, skim through the code provided by Apple at this link. Based on this quote and the code provided by Apple, I treat Swift protocols as traits. The definition of a *trait is a distinguishing quality or characteristic*. All too often, I hear Swift protocol being equated to interfaces and I believe this is not necessarily correct. To fully understand this, let's examine an interface.

Interfaces

> *A point where two systems, subjects, organizations, etc., meet and interact.*
>
> —Google

This is the definition of an *interface* provided by Google. This difference between protocols and interfaces boils down to how two things interact. Interfaces would be how two objects talk with one another in objected-oriented programming. Each object has an interface to the outside world, the public methods, and properties. Let's examine an interface in Java. The following code sample is from Oracle's documentation on interfaces at https://docs.oracle.com/javase/tutorial/java/concepts/interface.html. The following code defines an interface called `Bicycle`:

```
interface Bicycle {

    //  wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

This interface provides four methods that all describe how something can interact with a `Bicycle`. The rest of this article creates a class that implements this interface and this is a normal pattern in Java and other languages. The problem here is that the interface is called `Bicycle` and provides methods for interacting with a bicycle concept. You can see it has a set of brakes, pedals with which to speed up, multiple gears to switch between, and the ability to change what it calls the cadence. Next, let's examine a trait and how we can change this example to be more inline with Swift protocols.

Traits

> *A distinguishing quality or characteristic.*
>
> —Google

This is the definition for a *trait* and how Swift protocols should be treated. If we reexamine the `Bicycle` interface from the last section in terms of traits, what would it look like?

> **Note** I have omitted any properties of each protocol for clarity. Most likely, each protocol would have a set of properties associated with them.

```
//: # Bicycle Object with Traits

//: #### 1. An object can have a cadence
protocol hasCadence {
    func changeCadence(by value: Int)
}

//: #### 2. An object can have gears
protocol hasGears {
    func changeGears(to value: Int)
}

//: #### 3. An object can have pedals
protocol hasPedals {
    func speedUp(by value: Int)
}

//: #### 4. An object can have brakes
protocol hasBrakes {
    func applyBrakes(by amount: Int)
}
```

Here, I have defined four separate protocols. Each of these protocols corresponds with one of the methods defined in the previous interface. Each one represents a trait a bicycle might have. However, this does not have to be the case. In the previous interface, the idea of

pedals, brakes, gears, and so on are all associated with a bicycle concept. This is not how it would work in the real world. Brakes could also apply to cars, planes, or even tricycles. This means that in the previous interface, we would then need to duplicate code and concepts to achieve the other objects.

This is not the case with our trait example. The following is an implementation of a `Bicycle` class using the previously defined traits:

```
class Bicycle: hasCadence, hasGears, hasPedals, hasBrakes {

    // MARK: - Properties for a bike

    // 1
    func changeCadence(by value: Int) { }

    // 2
    func changeGears(to value: Int) { }

    // 3
    func speedUp(by value: Int) { }

    // 4
    func applyBrakes(by amount: Int) { }
}
```

This models our bicycle object perfectly. You can imagine, we would implement our methods and our bicycle could operate exactly as it should. We can then take our protocols and immediately create a `plane` class that conforms to `hasPedals` and `hasBrakes`. We would then have all the information required without the baggage of the other two protocols.

Using Swift protocols in this way alleviates any potential for a Blob or God Class (refer back to Chapter 4, where these are discussed in more detail). This is the power of treating Swift protocols as traits. We can model many different types of objects in our system and avoid common AntiPatterns that bog down other systems. Now, with all of this being said, interfaces still have their place in Swift. Developing to an interface is a valid objected-oriented pattern. Even though I would say Swift protocols are more in line with traits, Swift is still an object-oriented language. It is just protocol oriented as well. We will talk about this more once we go through the concept of abilities.

This section has walked through the differences between interfaces and traits. This difference is key to understanding this chapter and protocol-oriented programming. Essentially, it boils down to shifting your thinking to a different programming paradigm. The next section is going to delve into this shift.

Protocol-Oriented Thinking

We have already discussed the difference between an interface and a trait. This section is going to cover how we can start thinking differently to take full advantage of Swift protocols. This section is going to examine the idea of *abilities*. We are going to learn what this means in terms of code and our way of thinking.

If you're subclassing, you're doing it wrong.

—Hector Matos

The preceding quote is from an article by Hector Matos. His article discusses many of the same concepts we will touch on in this chapter. The article can be found at http://krakendev.io/blog/subclassing-can-suck-and-heres-why. In his article he talks about a lot of important concepts and ideas. He discusses AntiPatterns, abstractions and simplicity, and functional programming. He discusses all of these topics and centers the article on preferring composition and protocols instead of inheritance. I would highly recommend this article; it is a great read.

Moreover, I believe the title of his article says it all. The concepts from his article and those in this chapter allow us to move away from heavy inheritance and subclassing. These concepts then help us avoid the AntiPatterns we discussed in Chapter 4.

Let's take a step back and examine where you have used protocols before. If you have ever used a `UITableView`, `UICollectionView`, or `UIWebView`, then you have most likely used protocols in iOS. The `UITableViewDataSource` is an Objective-C protocol that requires a certain subset of methods to be implemented on the conforming object. These methods include the following:

```
// Asks the data source for a cell to insert in a particular location of the table view.

func tableView(_ tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell

// Tells the data source to return the number of rows in a given section of a table view.

func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int
```

The previous code is pulled from Apple's `UITableViewDataSource` reference. Once your object conforms to this protocol, it is required to have these methods along with others. The results of these methods are then used to fill out a table view. These methods are used via the `dataSource` property on a table view. If you look at the type of the `dataSource` property, it is a `UITableViewDataSource?`.

These methods can be implemented on any object. Swift has structs. We could conform to the `UITableViewDataSource` protocol on a struct and then hold onto it in our view controller. Normally, you would conform to this protocol on your view controller class, but with the power of protocols, this is not required. This is one of the keys to thinking in a protocol-oriented way.

Before we go any further, let's discuss what problem protocol-oriented programming is trying to solve. I argue the problem is pure objected-oriented programming. I use the word *pure* because Swift is still an object-oriented programming language. We still want to use objected-oriented programming paradigms where it makes sense. We can then use protocol-oriented programming where appropriate. Many languages are objected oriented and it will mostly likely also be in our programming languages in one form or another until the end of time.

I also used the word *pure* because languages without other paradigms do suffer from problems. I talked about how easy it can be to fall into AntiPatterns in Chapter 4. Using just inheritance can easily create a God Class. Let's examine this problem in depth in the next section.

The Problem: Object-Oriented Programming

Object-oriented programming can have its issues. It can also solve a lot of problems. The example we are going to look at in this section will be a simple one, but it illustrates this problem perfectly. We are going to model people and animals. I know this has been overdone a million times. Well, buckle up, because this will be a million and one.

> **Note** It is still possible to solve this example with object-oriented programming, but this chapter is about protocols, so tough noogies.

So, say you are a consultant. You get a new client and they want to build an awesome mobile app. They describe the problem to you as follows: "We want to build an app that can display different categories of creatures including animals and people. Users can then take pictures and upload them, and we want to classify their pictures based on specific types 'cause big data and all that cool stuff."

All right, so what you heard was "blah, blah, blah, take pictures, and classify into specific categories." So you get to work, deciding to start with the data models. For this project, we just need a model for `Creature`, `Animal`, and `Person`. Your first instinct might be object-oriented programming and inheritance, so let's see how this would look:

```
//: # Protocol-Oriented Thinking

//: ## The Problem: Object-Oriented Programming

class Creature {
    var alive: Bool

    init(alive: Bool) {
        self.alive = alive
    }
}
```

The previous code defines a `Creature` class that has one Boolean property: `alive`. We then have an initializer that injects this property. Pretty simple, and you are off to a good start. The next class you would then create would be an `Animal` class:

```
class Animal: Creature {
    var eyes: Int
    var nose: Int
    var ears: Int
```

```
    init(alive: Bool, eyes: Int, nose: Int, ears: Int) {
        self.eyes = eyes
        self.nose = nose
        self.ears = ears
        super.init(alive: alive)
    }
}
```

Here we have defined our `Animal` class that has three more properties: `eyes`, `nose`, and `ears`. All of these properties are `Int` properties because animals can have more than one eye, ear, or nose. Our initializer makes sure all of these properties are initialized properly. The next and last class to implement is the `Person` class:

```
class Person: Animal {
    var name: String
    var age: Int

    init(name: String, age: Int) {
        self.name = name
        self.age = age
        super.init(alive: true, eyes: 2, nose: 1, ears: 2)
    }
}
```

The `Person` class inherits from the `Animal` class, which inherited from the `Creature` class. It then adds two more properties for the `name` and `age` of the person. Now, pat yourself on the back because all of your data models are written and you can move on to implementing the meat of the app, except… the client then changes requirements on you. It turns out, aliens just made first contact.

So, now we have to be able to model an `Alien`, but technically, aliens are not animals. They might be classifiable as a `Creature`, but these aliens have eyes, ears, and noses. We would have to reimplement our functionality from the `Animal` class in our `Alien` class; we could also move the `Animal` functionality into the `Creature` class, which is not a good idea either.

You can see the issues with this approach. We have locked ourselves into an inheritance structure that would require refactoring to adapt to our new requirements. This is where protocols can come to the rescue. Let's reimplement the previous example in a protocol-oriented way. Before we start implementing, let's create a new playground page called *Ch05—Protocol-Oriented Thinking Pt2*. We can start with the trait the `Creature` class had:

```
//: ## The Solution: Protocols


protocol Living {
    var alive: Bool { get set }
}
```

We have defined a `Living` protocol that has our `alive` property on it. In protocols, we have to mark any variables as `get` and `set`. The `set` is not required as some properties might not be settable, but the `get` is always required. So far, so good; now let's finish this example with the traits from our `Creature` class and our `Person` class:

```
protocol Vision {
    var eyes: Int { get set }
}

protocol CanSmellThings {
    var nose: Int { get set }
}

protocol Evesdropper {
    var ears: Int { get set }
}

protocol Identifiable {
    var name: String { get set }
}

protocol CanGrowUp {
    var age: Int { get set }
}
```

We have all of our possible traits defined as protocols. We are almost done. I selected an example where it would be easy to identify the traits from our objects. Most likely, in the real world, you would not split create protocols that only contain one property. You would most likely create protocols that contain groups of properties and methods that all correspond to one possible trait. Now, let's finish this example by defining all four classes with our protocols instead of inheritance.

```
//: #### Creature, Animal, Person, & Alien with Protocols


class Creature: Living {
    var alive: Bool
    init(alive: Bool) {
        self.alive = alive
    }
}

class Animal: Living, Vision, CanSmellThings, Evesdropper {
    var alive: Bool
    var eyes: Int
    var nose: Int
    var ears: Int

    init(alive: Bool, eyes: Int, nose: Int, ears: Int) { ... }
}
```

```
class Person: Living, Vision, CanSmellThings, Evesdropper, Identifiable, CanGrowUp {
    var name: String
    var age: Int

    var eyes: Int
    var nose: Int
    var ears: Int

    var alive: Bool

    init(name: String, age: Int) { ... }
}

class Alien: Living, Vision, Evesdropper, Identifiable {
    var name: String

    var eyes: Int
    var ears: Int

    var alive: Bool

    init(name: String, eyes: Int, ears: Int) { ... }
}
```

You can see with the previous code, we have created a little more overhead for ourselves. Now, every class must define each property instead of just inheriting it from a superclass. However, imagine this was how we implemented our models the first time. When the client came back to us and asked to add aliens, we would have said "Sure," and it would have been done in no time.

If you are saying to yourself," This is great and all, but where is the new paradigm?" know that what I have shown you so far is just part of the equation. The next section is going to expand on this example with protocol extensions. This is where abilities come into play. The next section is going to expand on our previous example. It will then show where protocol-oriented programming really shines in SpriteKit and how we can use it in UIKit. It's going to be a big section, so I will see you there.

Abilities

We have a lot to cover. First, we are going to expand on our previous example and start by explaining abilities using extensions and constraints. After we finish this example, we are going to get into some iOS game development through SpriteKit. This should be a really fun part. This will lead us into our discussion on value types and reference types. Finally, we will finish this section up with a more practical look at where we can take advantage of protocol-oriented programming while using UIKit. I want to start by explaining what I mean with I say *abilities*.

This entire chapter, I have been using the word *ability.* This word is not from any Swift documentation. I use this word to describe the concept of extending protocols to give them default implementations. The `extension` keyword and the word *extend* are a bit overloaded, so I wanted a different word to explain the concept. I also believe the term *ability* drives home the purpose of the concept.

So here it is—the full explanation of *trait* and *ability*. We want to create protocols that represent traits our objects can have. We then want to give our traits abilities through the use of protocol extensions and constraints. I will still be using the terms *protocol extension* and *constraints* to explain the specific code. The term *ability* is great when describing the concept and purpose of the code.

Now, I want to take a step back and talk about interfaces one more time. Earlier we discussed the differences between an interface and a trait. Treating Swift protocols as traits is key to protocol-oriented programming. Remember, though, that Swift is still an object-oriented language. Interfaces have their place, and they should describe a common interface to a type of object. Here's the drawback to interfaces, though. I have found that when you create an interface, it is not always appropriate to give it abilities. Traits, however, can and should have abilities. Protocols can act as a traits or an interface, but it is up to us to distinguish between the two and use them appropriately.

This is not a hard and fast rule. I believe that by following this rule, we can write better code. This is where you can take this information and make your own decision. Now that we have a handle on this whole idea of traits and abilities, let's expand on our Creature/Person/Alien example.

Creatures, Animals, People, and Aliens

Let's start this section by jumping right into our code. First, we need to add methods to three of our protocols:

> **Tip** I have modified and added to the protocols and classes in my playground reference. It might be better for you to keep the old page and create a new page if you want to see the progression. However, I am still using the old playground page name.

```
protocol Vision {
    var eyes: Int { get set }
    func look(for something: Any)
}

protocol CanSmellThings {
    var nose: Int { get set }
    func smell(thing: Any)
}

protocol Evesdropper {
    var ears: Int { get set }
    func listen(for aSound: Any)
}
```

I have edited the `Vision`, `CanSmellThings`, and `Evesdropper` protocols. I have added `look`,

smell, and listen methods to the protocols respectively. Now the playground page will not compile because the conforming classes do not have any of these methods. Before we add these methods to our classes, let's start with an extension on our Vision protocol:

> **Caution** I have placed the protocol extensions in-between the protocols and the conforming classes' definitions.

```
//: #### Abilities for Vision, CanSmellThings, and Evesdropper

extension Vision {
    func look(for something: Any) {
        print("Look for \(something)")
    }
}
```

In the previous code, we have added an implementation to the Vision protocol. Now, our protocol has an implementation and our classes no longer need to provide one. It is important to know that our classes can still provided an implementation, but it will not have any connection to the protocol's implementation. Protocol extensions do not work the same way inheritance works. Before we continue, let's implement this method in our Animal class:

```
class Animal: Living, Vision, CanSmellThings, Evesdropper {
    var alive: Bool
    var eyes: Int
    var nose: Int
    var ears: Int

    init(alive: Bool, eyes: Int, nose: Int, ears: Int) {
        self.eyes = eyes
        self.nose = nose
        self.ears = ears
        self.alive = alive
    }

    func look(for something: Any) {
        // Do something here
    }
}
```

Here is the entire Animal class. When you start to type look, if the playground completes the code for you, you will notice it does not contain the override keyword. This is because the override keyword signifies this method is already implemented and we are going to add an overriding implementation, but we can still call super. This is a common inheritance and polymorphism. However, for protocol extensions, there is no super and there is no inheritance. The protocol extension implementation is mixed into our object. This is very important, so I wanted to make sure I was very clear about this subject. Now, here are all of our protocol extensions:

```
//: #### Abilities for Vision, CanSmellThings, and Evesdropper

extension Vision {
    func look(for something: Any) {
        print("Look for \(something)")
    }
}

extension canSmellThings {
    func smell(thing: Any) {
        print("Smelling: \(thing)")
    }
}

extension Evesdropper {
    func listen(for aSound: Any) {
        print("Listen for \(aSound)")
    }
}
```

These extensions are very simple, but that is all there is to it. Now, with all of our protocol methods implemented in extensions, our code should compile. Also, be sure to remove the `look` method's implementation from our `Animal` class. Let's instantiate our classes and see what happens when we use these APIs:

> **Note** I have placed this section at the bottom of the playground.

```
//: #### Using our classes and protocol extensions


var bob = Person(name: "Bob", age: 30)
bob.listen(for: "Music")

var anteater = Animal(alive: true, eyes: 2, nose: 1, ears: 2)
anteater.smell(thing: "Ants")

var ET = Alien(name: "ET", eyes: 2, ears: 2)
ET.look(for: "Home")
```

We have created three objects in the previous example. We created `bob`, `anteater`, and `ET`. Each object then uses one of our protocol methods. Take a look at the method list on `ET`; the list should not contain a smell method. Our `Alien` class does not conform to the `CanSmellThings` protocol, so `ET` cannot smell anything. Here is the entire playground page *Ch05—Protocol-Oriented Thinking Pt2*. I have omitted the class implementations to conserve space:

```
//: ## The Solution: Protocols

protocol Living {
    var alive: Bool { get set }
}
```

```
protocol Vision {
    var eyes: Int { get set }
    func look(for something: Any)
}

protocol CanSmellThings {
    var nose: Int { get set }
    func smell(thing: Any)
}

protocol Evesdropper {
    var ears: Int { get set }
    func listen(for aSound: Any)
}

protocol Identifiable {
    var name: String { get set }
}

protocol CanGrowUp {
    var age: Int { get set }
}

//: #### Abilities for Vision, CanSmellThings, and Evesdropper

extension Vision {
    func look(for something: Any) {
        print("Look for \(something)")
    }
}

extension CanSmellThings {
    func smell(thing: Any) {
        print("Smelling: \(thing)")
    }
}

extension Evesdropper {
    func listen(for aSound: Any) {
        print("Listen for \(aSound)")
    }
}

//: #### Creature, Animal, Person, & Alien with Protocols

class Creature: Living {
    // ...
}

class Animal: Living, Vision, CanSmellThings, Evesdropper {
    // ...
}
```

```
class Person: Living, Vision, CanSmellThings, Evesdropper, Identifiable, CanGrowUp {
    // ...
}

class Alien: Living, Vision, Evesdropper, Identifiable, SuperVision {
    // ...
}
```

This is pretty awesome stuff and this is the power behind protocol-oriented programming. Now, let's take this up a notch. In the next code block, we are going to add another protocol and another extension. We are then going to use constraints to modify our `Alien`'s behavior all through protocols. Let's see how this works:

```
//: The SuperVision protocol is more of a decorator
protocol SuperVision {}

//: Extension on the SuperVision protocol where the conforming type also conforms to Vision
extension Vision where Self: SuperVision {
    func look(for something: Any) {
        print("Really look for \(something)")
    }
}
```

I have grouped the protocol and the extension here for clarity, but in my playground reference, they are in their appropriate sections. All right, we have defined a new protocol called `SuperVision`. This protocol does not have any methods or properties. In this example, I am using it more as a decorator. So, the really interesting piece here is our new protocol extension. We extend our `Vision` protocol where the conforming type also conforms to the `SuperVision` protocol. This means we will have two extensions for the `Vision` protocol. One extension is where we have no constraints and then we have our second that is constrained to `SuperVision`.

Now, instead of me just explaining this, conform to the `SuperVision` protocol on the `Alien` class and see what happens in the console. Before `ET` was just looking for home. Now, he is *really* looking for home. Because he's an alien, he has superhuman powers. However, if you were to call `look` on `bob`, he is just able to look for things.

By defining our `SuperVision` constraint on the `Vision` protocol, we have added more functionality to any conforming type that has both protocols, but only where both protocols are present. Using these constraints can be very useful when we have traits that we want to add abilities to, but only in certain circumstances. We can also use classes as constraints on our protocols and we will see this more in the next section.

You have seen the basics of protocols, protocol extensions, and constraints. Let's make this a little more complicated by looking at game development. We are going to see how protocol-oriented programming can make our SpriteKit games very easy to write and maintain.

SpriteKit Game Development

If you have ever done iOS game development with SpriteKit before, you know the framework is really easy to use, but there are a few things that can present problems. If you have never used SpriteKit, it is a 2D game development framework built by Apple. It was released with

iOS 7. I wanted to focus on a couple of the issues and how we can solve them through the use of protocol-oriented programming. I believe protocol-oriented programming can really shine in game development, which is why it is included in this chapter.

Let's first describe the problem. It is not just a problem with SpriteKit, but the same problem is described in the object-oriented programming section. SpriteKit has an object called `SKNode`. This object is the base building block for all visual objects in the SpriteKit framework. Then, to draw content, such as an image, we need to use the `SKSpriteNode` class. `SKSpriteNode` inherits from the `SKNode` class.

This works great, but these objects do not have game behavior to them such as scrolling across the screen. These objects just provide the mechanisms available to developers to create such an action. This makes sense too because these objects are the foundation for our games and they should not contain game-specific logic, just facilitate our logic. This makes it difficult on us though.

Let's examine the scrolling problem. Figure 5-1 shows the problem with adding scrolling to our game. Just like many other game frameworks, SpriteKit uses a node hierarchy structure to represent game scenes where `SKNodes` can contain other `SKNodes`. So let's say we create a `Background` class that needs to scroll. We also want to create a `Platform` class that the player of our game can interact with. Both of these objects will need the ability to scroll across the screen. However, in Figure 5-1 you can see we cannot have our `Background` class inherit from `SKNode` and a `Scrolling` class. The same issue exists with `SKSpriteNode` and the `Platform` class.

Using our knowledge of protocol-oriented programming, think about what the solution to this would be. We are going to define a trait and give that trait the ability to scroll, but this one is going to be a bit tricky. We need `SKNode` functionality to scroll a node, but all we are going to have is a trait and an ability. The following code is how I have implemented the solution to the problem described in Figure 5-1:

```
//: # SpriteKit example to illustrate POP techniques in Game Development

//: #### ScrollDirection defines the direction a node can scroll

enum ScrollDirection {
    case Forward
    case Backward
}

//: #### The Scrolling protocol defines the necessary properties and actions
//: #### to properly scroll a node

protocol Scrolling {
    var scrollDirection: ScrollDirection { get set }
    var scrollSpeed: CGFloat { get set }

    func scroll()
    func scrollForever()
    func scrollAction() -> SKAction
    func scrollForeverAction() -> SKAction
}
```

I have defined an `enum` and a protocol called `Scrolling` here. The `enum` defines the two directions we can scroll a node, `Forward` and `Backward`. This protocol defines four methods and two properties meant to control how the node scrolls. Now, let's add two separate extensions for our `Scrolling` protocol. Let's start with the first extension on our `Scrolling` protocol:

```
extension Scrolling {
    /// Using the scrollDirection and the scrollSpeed to describe the direction and speed
used for scrolling
    var scrollDelta: CGVector {
        return CGVector(dx: 1.0, dy: 0.0)
    }

    /// Defines the SKAction used to scroll
    func scrollAction() -> SKAction {
        return SKAction.move(by: scrollDelta, duration: 1.0/60.0)
    }

    /// Repeats the scrollAction's SKAction forever
    func scrollForeverAction() -> SKAction {
        return SKAction.repeatForever( scrollAction() )
    }
}
```

The first extension on `Scrolling` defines the implementation for the `scrollAction` and `scrollForeverAction` methods. I have also added a computed property for a `CGVector`. This describes the vector used for the scrolling direction and speed. I have hardcoded this value, but in a real implementation, it would combine the `scrollDirection` and `scrollSpeed` properties to calculate the vector.

`SKAction` is a class that allows you to change a node's properties and the structure within the game scene. In our case, we just want to scroll a node by creating a move `SKAction`. The previous protocol extension is just half of our implementation, though. We still need to implement the other two methods, `scroll` and `scrollForever`. Let's define the second protocol extension where we implemented these methods:

```
extension Scrolling where Self: SKNode {
    /// Uses the run method on SKNode to execute the SKAction object from scrollAction
    func scroll() {
        run( scrollAction() )
    }

    /// Uses the run method on SKNode to execute the SKAction object from
scrollForeverAction
    func scrollForever(){
        run( scrollForeverAction() )
    }
}
```

So, in our first protocol, we defined `SKAction` objects. The only way we can run these actions is by using an `SKNode` object. This is where the constraints on the second protocol extension come into play. We constrain the two methods `scroll` and `scrollForever` to be implemented where the conforming type is an `SKNode` type. We can then call `self` as if it was an `SKNode` type and this gives us the `run` method. We then just call the other two protocol methods to get our `SKAction` objects to run, and we run them.

There was not a lot of code in this section, but what we have created is pretty awesome. We can now use this protocol on any `SKNode` subclass and it will automatically be capable of scrolling across our scene. Let's go back to our original problem described in Figure 5-1. We want a `Background` class that is a subclass of `SKNode` and a `Platform` class that inherits from `SKSpriteNode` and both need the ability to scroll.

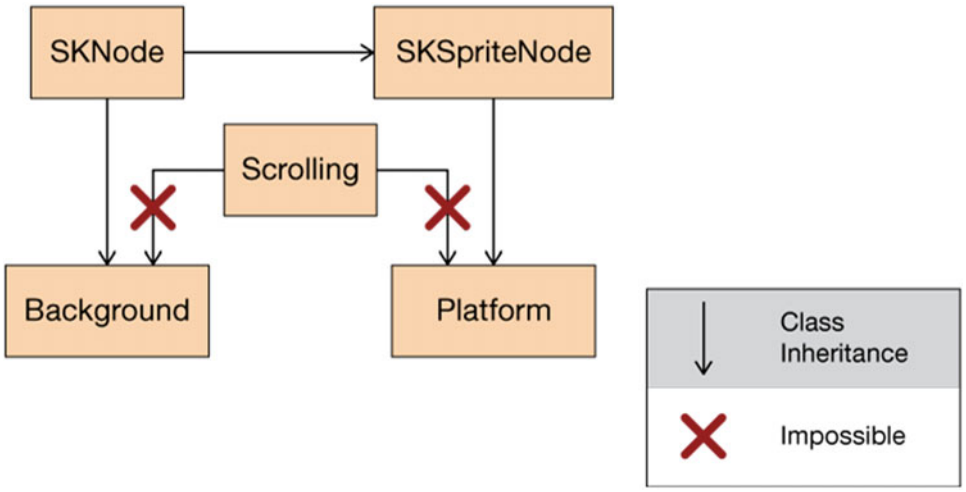


Figure 5-1. *Shows the issue with creating a Background class and a Platform class that inherits from SKNode and SKSpriteNode, respectively, that both need scrolling logic*

So, in our solution, we built a `Scrolling` protocol that described how scrolling can work. We then built two separate extensions. Using these two extensions, any `SKNode` subclass we create can conform to the `Scrolling` protocol and then have the ability to scroll. It is important to understand that if a type conforms to our `Scrolling` protocol that is not an `SKNode` type, we will not have an implementation for the `scroll` and `scrollForever` methods. However, we will still retain the implementation from the first protocol extension with no constraints. We can see how this solution looks in Figure 5-2.

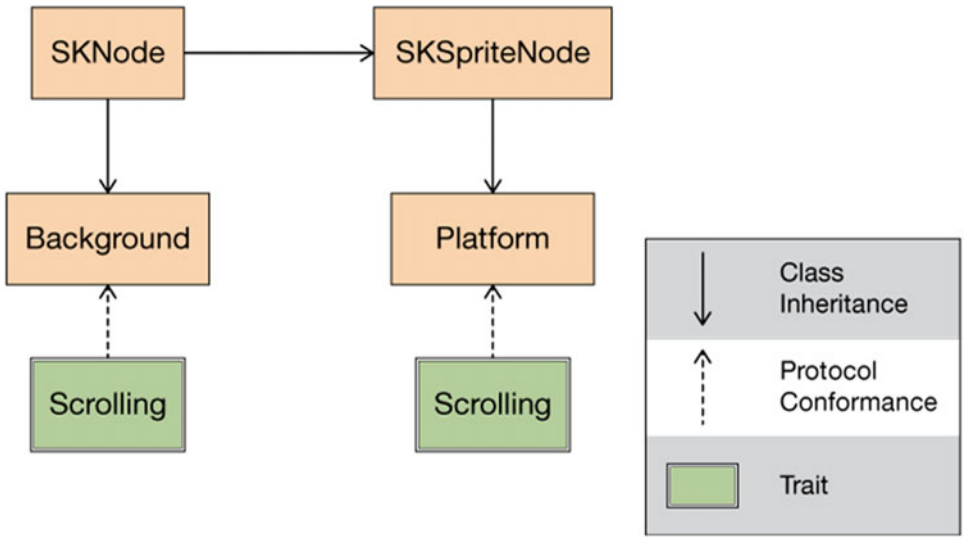


***Figure 5-2.** Shows the solution we created with a Scrolling trait*

In Figure 5-2, we can see our two classes inherit from their proper classes, `SKNode` and `SKSpriteNode`. We can then give our `Background` and `Platform` classes the trait: `Scrolling`. I chose to have scrolling represented with two boxes. We still only have one protocol and one set of extensions. This is to illustrate the idea that protocols are lightweight and they are a trait given to classes rather than an object in space we can inherit from. We can also have each class implement their own version of scrolling logic if need be.

This section has examined using protocol-oriented programming in iOS game development with the SpriteKit framework. We have seen how employing protocols as traits can clean up the implementation of game behavior. In the next section, we are going to cover value types and reference types. This is another aspect to protocol-oriented programming and a very important aspect of Swift.

Value and Reference Types

In Swift, there are two categories of types: value types and reference types. This section is going to cover both categories, with a focus on value types. Then we are going to look at how and when to use each category.

Let's start with the first category, value types. In Swift, value types are structs, enums, and tuples. Swift's `String`, `Array`, `Dictionary`, `Int`, and so on are all structs. Structs do not have the ability to inherit from other structs. They can however, conform to protocols. Protocols for the win! This means if we need to give multiple structs a common set of functionality, the only mechanism we have is through protocols. Structs are also very safe.

This is why structs are favored in Swift. Each instance of a Swift struct holds onto a unique copy of its own value. Therefore anytime we set a variable to a struct, that struct is copied into the variable and we have two copies. Let's see this in code:

```
//: # Basics of Value Types

struct Thing {
    var id: Int
}

let thing1 = Thing(id: 1)
var thing2 = thing1
thing2.id += 1

thing1.id
thing2.id
```

In the previous code block, we have defined a struct called `Thing`. This struct has an `id` variable and that's it. We then declare an instance of this struct called `thing1` and then another variable called `thing2`, which is just set to `thing1`. We then increment the `id` of `thing2` by 1. The last two lines will just print out the value of `id` on `thing1` and `thing2`. We are dealing with value types, so `thing1` will still have the correct `id` of 1, while `thing2` has an `id` of 2.

This works because the value of `thing1` was copied to our variable `thing2`. If we were using reference types here, the result would have been much different. The last thing you will notice is that we have used the `let` keyword with `thing1`, but the `var` keyword with `thing2`. This is because we mutated the value of `thing2`. Structs are safe because they will prevent you from mutating their values if it is declared as a constant with `let`. Structs also force you to mark methods that change the internal values as mutating. Let's see this in action:

```
struct Thing {
    var id: Int

    mutating func change(id: Int) {
        self.id = id
    }
}

let thing1 = Thing(id: 1)
var thing2 = thing1
thing2.change(id: 2)

thing1.id
thing2.id
```

We have added a mutating method to the `Thing` struct called `change`. The purpose of this method is to mutate the value of `id` and we still cannot declare `thing2` as a constant because it is being mutated. This is not the case with reference types. If a constant is declared with a reference type, the constant itself cannot be mutated, but the internals of the reference *can* be mutated. So, we declare a constant set to a `UIViewController`. We cannot set that constant to any other `UIViewController`, but we can set the `view` property on `UIViewController` to another `UIView`.

You can see how safe this type of coding can be. The compiler forces us to think about the mutation of this object. We can only mutate this object if it is a variable. We also cannot run into a situation where we have two references trying to update the value simultaneously. This is a huge win in a multithreaded application. If we were working with reference types, we would have to worry about all of those problems.

References types in Swift are classes. Anything defined as a class, is a reference type. This includes most of UIKit and SpriteKit and many other iOS frameworks. Reference types are not copied, hence their name. Let's reexamine our last code using reference types:

```
 //: # Basics of Reference types


class Object {
    var id: Int

    init(id: Int) {
        self.id = id
    }
}

let object1 = Object(id: 1)
let object2 = object1
object2.id = 2

object1.id
object2.id
```

The differences here should be very apparent. Our `Object` class is just like our `Thing` struct. However, when we declare `object2`, it can be a constant. We then set the `id` of `object2` to 2 and see what happens. Because we are dealing with reference types here, the `id` of both `object1` and `object2` is now 2.

Sometimes, this is what we want and can be an advantage, but if we are in a multithreaded system, this can create unwanted side effects that would be very difficult to track down. We have to be careful and understand that reference types are not copied when set to another variable like the preceding example.

So, what does this all mean and why is it in the protocol-oriented programming chapter? Apple has said that Swift developers should favor value types because of their safety. This safety makes our lives easier. In Chapter 1, I talked about *optionals*. Optionals were a strange new concept to iOS introduced in Swift 1. I then argued that when optionals are used correctly, our apps become safer. I argue the same thing here. As iOS developers, if we use value types and protocols over classes and inheritance, our apps become cleaner and safer. The next section is going to cover a practical look at protocols in UIKit.

Protocols in UIKit

UIKit and protocols can be a bit tricky. UIKit is comprised mostly of classes. This framework houses our `UIViewController`, `UITableViewController`, and so on. This can make protocol-oriented programming hard to think about at first. UIKit is where you might think value types

are great, but if we are dealing with UIViewControllers and mostly classes, we will have to use reference types. That is not necessarily true. Yes, we will have to use classes for certain things to work, but if we start to think in a more protocol-oriented way, we can use protocols and value types to do some pretty cool stuff.

This section is where we bring everything we have talked about together. We will see how to use traits and abilities and how to use value types all in UIKit. We will rely heavily on this section in future chapters, especially when building our app. Let's get started!

> **Note** Let's start a new page in our playground reference. Also, be sure to import UIKit at the top of the new playground page.

We are going to start with three protocols. The three protocols are going to describe an object that can have a view, an object that can present view controllers, and an object that can block the UI. These three protocols are defined as follows:

```
//: # Protocols for working with UIKit


protocol hasView {
    var view: UIView! { get set }
}

protocol canPresentViewControllers {
    func present(_ viewControllerToPresent: UIViewController, animated flag: Bool,
completion: (() -> Void)?)
}

protocol canBlockView {
    func block()
}
```

The previous code block has defined three protocols: hasView, canPresentViewControllers, and canBlockView. A UIViewController subclass can conform to the first two protocols without any more code being required. These protocols are meant to replace references to UIViewController. Let's start with the canBlockView protocol and add an extension with a constraint. The idea is that we want to block the view of a particular UIViewController without being constrained to the UIViewController type:

```
extension canBlockView where Self: hasView {
    func block() {
        // block the view
    }
}
```

I am not going to write the implementation of this method here for clarity. We will go in-depth into the functionality of this protocol in Chapter 7. So, we have defined an extension for `canBlockView` with an implementation of `block` and this method will only be available to types that conform to `hasView`. If we did not have the `hasView` protocol, we most likely would have constrained our implementation to the `UIViewController` type. This does not scale very well. So, now with the `canBlockView` and `hasView` protocols, we can block the view on a `UIViewController` or any other object that has a view property.

Let's now examine the third protocol. This is again going to free us from using a `UIViewController` type. All right, we want to be good Swift developers, and we know we should favor value types and protocols. We already have our protocol, so let's define the value type. We want to be able to present `UIAlertControllers` in a clean way.

To do this, let's define a struct called `AlertDisplayer`. This struct is then going to hold a reference to something that can present view controllers. Then all we need to do is present the alert controller. Again, the implementation is going to be omitted for clarity:

```
//: #### Alert Displayer using struct and protocol


struct AlertDisplayer {
    var canPresentControllers: canPresentViewControllers

    init(canPresentControllers: canPresentViewControllers) {
        self.canPresentControllers = canPresentControllers
    }

    func displayAlert(withTitle title: String?, andMessage message: String?) {
        // present UIAlertController
    }
}
```

This code block defines the `AlertDisplayer` struct. The constructor takes a type that conforms to our protocol and holds onto it. We then have a `displayAlert` method that would create a `UIAlertController` and present it. Pretty cool, huh? We have completely freed ourselves of the `UIViewController` type here. Let's see how we can now use this:

```
extension UIViewController: canPresentViewControllers {}


let viewCtrl = UIViewController()
let alertDisplayer = AlertDisplayer(canPresentControllers: viewCtrl)
alertDisplayer.canPresentControllers

let secondAD = alertDisplayer
secondAD.canPresentControllers
```

The first line is key here. We extend the `UIViewController` type to conform to our `canPresentViewControllers` protocol. All right, I am going to get a bit technical on you. This first line is an extension on `UIViewController` that makes it conform to our protocol. This is called *retroactive modeling*. This is another aspect to protocol-oriented programming.

We can have any class, struct, or enum that we define easily conform to our protocols. Extensions, though, allow us to have classes, structs, and enums outside our control conform to our protocols as well. So, anything we have access to in UIKit can conform to our protocols. In the previous code block, we extend `UIViewController` to conform to our `canPresentViewControllers` protocol. Without subclassing, all view controllers can now be used as this protocol type.

Let's now look at the rest of the code from the previous block. It creates a view controller (`viewCtrl`), then an alert displayer (`alertDisplayer`) with that view controller. I also added in the last three lines to show that the alert displayer object will be copied, but the reference to the view controller is still intact.

Wow! This has been a gigantic section and it has been really fun to write. So what have we learned in this section? We have learned about the concept of *traits* and *abilities* working together. We define our protocols as a trait an object can have. We then give abilities to this trait. This section also described my theory that Swift protocols distinguish between two concepts: the first being *traits* and *abilities* and the second being *interfaces*. I think these two concepts working together start to create the paradigm shift in Swift that is protocol-oriented programming. I think this is one of the greatest takeaways from this section as well as the entire chapter and book.

We expanded on our knowledge of traits and how much of a difference they can make in iOS game development. We took a common problem in SpriteKit and made a very simple solution that scales for any SpriteKit game. After SpriteKit, we examined the differences of value types and reference types. Finally, we brought all this all knowledge together and took a practical look at how we can use traits, abilities, and value types in our normal everyday UIKit work. This will be very important when we start building our app. The last section is going to finish things off by describing how protocols and our newfound techniques can make testing easy.

Testing with Protocols

This is the last section in this chapter. We are going to discuss unit testing using protocols. Chapter 8 will go more in-depth on the subject of testing in iOS and Swift, but I wanted to make sure protocols are fresh in your mind when discussing this section.

This section is going to look at how we can test the previous section's code, specifically, the `AlertDisplayer` struct. We will then look at how we can test another aspect of UIKit. In general, singletons can be very hard to test, and we will examine how we might test the `UIApplication` singleton. Let's get started!

Testing AlertDisplayer

We are going to look at how we can test the `AlertDisplayer` struct we created in the previous section. Testing Swift code, especially when you mix in the UIKit framework, can sometimes be difficult. By using protocols, we can make this process easy, if not trivial. Here is all of the code we are going to test:

> **Note** I have posted the code here for clarity. I would recommend starting a new playground page and putting this code at the top.

```
protocol canPresentViewControllers {
    func present(_ viewControllerToPresent: UIViewController, animated flag: Bool,
    completion: (() -> Void)?)
}

struct AlertDisplayer {
    var canPresentControllers: canPresentViewControllers

    init(canPresentControllers: canPresentViewControllers) {
        self.canPresentControllers = canPresentControllers
    }

    func displayAlert(withTitle title: String?, andMessage message: String?) {
        // present UIAlertController
    }
}
```

We are going to write test code here, but we are still in a playground, so we will not be able to run our test. Here is the start of the test code:

```
//: # Testing with protocols

class ProtocolTest {

}
```

This class will contain the test method for our `AlertDisplayer` struct. Before we write the test, let's see what we are going to test. The `AlertDisplayer` struct does not contain a view controller, but a reference to the protocol `canPresentViewControllers`. Even though we are going to use this struct in an app to display a `UIAlertController` on a `UIViewController`, we do not need this to occur in the test. Instead, we can have anything conform to this protocol. Let's set this up now:

```
class ProtocolTest {

    // Fake test object that conforms to canPresentViewControllers

    class TestObject: canPresentViewControllers {
        var controller: UIViewController? = nil
        var flag: Bool? = nil
        var completion: (() -> Void)? = nil

        func present(_ viewControllerToPresent: UIViewController, animated flag: Bool,
completion: (() -> Void)?) {
            self.controller = viewControllerToPresent
            self.flag = flag
            self.completion = completion
        }
    }
}
```

Now, the class TestObject does not do anything to present a view controller; it just records the parameters. Now, let's write the test method using this TestObject class:

```
class ProtocolTest {

    func testDisplayAlert() {
        let testObj = TestObject()
        let subject = AlertDisplayer(canPresentControllers: testObj)

        subject.displayAlert(withTitle: "Title", andMessage: "Message")

        testObj.controller != nil
        testObj.flag == true
        testObj.completion == nil
    }

    class TestObject: canPresentViewControllers {
        // Test Object Code ...
    }
}
```

In the test for the AlertDisplayer, we create an instance of the TestObject class and we create an instance of the AlertDisplayer called subject. This is the key to the test: we inject the instance of TestObject. Then, we call displayAlert and expect the testObj properties to prove our code actually displays a UIAlertController.

As stated previously, this code is not a valid test that will run but the concept is what is important. By using protocols, in our implementation, we were able to avoid using view controllers and we successfully isolated the test for the AlertDisplayer. Let's look at a slightly more complicated example using the UIApplication.

Testing UIApplication

The title for this section is a bit of a misnomer. We are not going to test any code within the UIApplication. We are going to test code that was previously untestable because the UIApplication was used. First, I cannot take credit for this example. It is an amazing example and it is from Eli Perkins' blog post http://blog.eliperkins.me/mocks-in-swift-via-protocols. It is such a good example I had to include it here.

All right, the problem is that we need to register our app for push notifications. If you do not know how this is done, that's okay, I don't know either. This example is going to focus on one method: registerForRemoteNotifications. This method is on the UIApplication singleton and feels like a black hole. We call it, a lot of things happen, and at the end, the app is registered for remote push notifications. How can we possibly test this? Well, there's this function on UIApplication: isRegisteredForRemoteNotifications. That won't work though, because this is a singleton and once we are registered, that's it, we are registered. By using this property, we would create a test that actually tries to register us for remote notifications. It would also be a very brittle test since we do not know what any of this code does.

This is the problem and this is what we are going to test, but let's take a step back and write the code behind the problem. We are going to assume we are in an app using the MVVM design pattern from Chapter 4. So, we will have a `ViewModel` class. This is also about registering a user for remote notifications, so we will also need a `User` object. Let's see this in code:

```
//: # Testing UIApplication with Protocols
struct User {}

class ViewModel {
    var user: User? = nil {
        didSet {
            print("Registered")
            UIApplication.shared.registerForRemoteNotifications()
        }
    }
}
```

The previous code block defines a struct for our user. We then have a `ViewModel` class that contains the user. The `ViewModel` class then uses Swift's property observer `didSet` on the `User` property to register for remote notifications when the `user` object is set. Now, with the previous code, imagine what your test would look like. The following code block shows the beginning of a test for this code. However, once we reach the assertions, what do we do?

```
class TestViewModel {
    func testShouldRegisterForRemoteNotifications() {
        // Setup
        let subject = ViewModel()
        let user = User()

        // Action
        subject.user =  user

        // Assert
        // ??????
    }
}
```

So, here is the problem. Now, let's look at how we can make this testable with protocols. Thinking about how the `AlertDisplayer` struct works is the key to this problem. If we could have a protocol that could represent the `UIApplication`, this would become instantly testable. Let's define this protocol:

```
// Protocol used to describe functionality from the UIApplication
protocol PushNotificationRegistrar {
    func registerForRemoteNotifications()
}
```

This protocol only has one function on it and it might look familiar. Next, let's modify our `ViewModel` to inject an object that conforms to this protocol as a dependency:

```
class ViewModel {
    var user: User? = nil {
```

```
        didSet {
            print("Registered")
            registrar.registerForRemoteNotifications()
        }
    }
    // This registrar holds the functionality for registering for remote notifications
    var registrar: PushNotificationRegistrar

    // Dependency injection allows us to inject anything that conform to this protocol
    init(registrar: PushNotificationRegistrar) {
        self.registrar = registrar
    }
}
```

The `ViewModel` class now does not use `UIApplication` at all. It just uses the registrar. Through dependency injection, this can be any object. Now, let's bring this all together:

```
extension UIApplication: PushNotificationRegistrar {}
```

Earlier in this chapter you might remember extending `UIViewController` with a protocol and we can do the same to `UIApplication`. We are using retroactive modeling on `UIApplication` because it already has the method from our protocol. This means the production code can use `UIApplication`, while our test can look like the following:

```
class TestViewModel {
    func testShouldRegisterForRemoteNotifications() {
        // Setup
        let registrar = FakeRegistrar()
        let subject = ViewModel(registrar: registrar)
        let user = User()

        // Action
        subject.user =  user

        // Assertion
        registrar.registered == true
    }

    // This class conforms to our protocol for testing
    class FakeRegistrar: PushNotificationRegistrar {
        var registered = false
        func registerForRemoteNotifications() {
            registered = true
        }
    }
}
```

Now, the test contains a `FakeRegistrar` class that conforms to the `PushNotificationRegistrar` protocol. An instance of this object is then injected into the `ViewModel` that is under test. Now, the test is easy and we can just assert the registered property is true at the end. No need for any black magic voodoo to figure out if the `UIApplication` is registered, which makes our test isolated, reliable, and easy to understand.

This has been a pretty cool section (at least I hope so). This section has covered how to test code using protocols and protocol-oriented programming. There is a huge reason iOS and Swift developers should start thinking this way. We can write all the awesome code we want, but if it is not testable, it will not be used. Swift also does not have all the nifty tricks of Objective-C, so we will have to think in different ways when testing Swift code. We will discuss this more in Chapter 8. I wanted to end this section with huge thank you to Eli Perkins' blog post at `http://blog.eliperkins.me/mocks-in-swift-via-protocols` for coming up with this example.

Wrap Up

Wow, this has been a big chapter! We have discussed many aspects of protocol-oriented programming. I introduced the concept of traits and abilities and explained the differences with nterfaces. I want to reiterate this concept, since it is a cornerstone of this chapter/book. Swift protocols lend themselves more to the idea of a *trait* (a characteristic or quality of an object) rather than just to an *interface* (a contract for common communication for an object). Swift then allows us to give these traits *abilities* through the use of protocol extensions. With all of this being said, interfaces still have their place in Swift.

Using this concept, we then looked at how we can use traits and abilities in iOS game development. This concept can clean up game code and create smaller building blocks for faster game prototyping. We also examined the differences between value and reference types. Using value types in combination with protocols in UIKit can create safer and scalable apps. Finally, the last concept we discussed what retroactive modeling. This can take our protocol functionality to the next level since we are not limited to objects within our control. Retroactive modeling allows us to extend objects outside our control with our protocols and functionality.

I think all of these concepts working together describe the protocol-oriented programming paradigm. This new paradigm requires a greater shift in thinking required for modern day development. In modern day development, we cannot silo ourselves into one way of thinking. We need to think differently and use different paradigms to create scalable applications. I think this is what Swift is trying to do and as Swift matures, I believe these paradigms will allow Swift to be used in more than just iOS and macOS apps.

The chapter then finished by using these concepts to write tests for our code. Chapter 8 is going to be an in-depth look at testing Swift code, but this was a glimpse at how easy it can be to test Swift code using protocols. Retroactive modeling also made the last example using `UIApplication` possible and made untestable code testable. This is a huge win for protocol-oriented programming. And that is it for this chapter. Next up, we are going to explore Swift generics. Rest assured, we are not done with protocols. We will use protocols as much as we can when we build our app in the second half of this book. The next chapter is also going to have a section on protocols. See you there.

Articles

1. Protocol-Oriented Programming in Swift
   - https://developer.apple.com/videos/play/wwdc2015-408/
2. If You're Subclassing, You're Doing It Wrong.
   - http://krakendev.io/blog/subclassing-can-suck-and-heres-why
3. Parametric (Compile-Time) Polymorphism in Swift
   - http://nsomar.com/parametric-compile-time-polymorphism-in-swift/
4. The Ghost of Swift Bugs Future
   - http://nomothetis.svbtle.com/the-ghost-of-swift-bugs-future
5. Mixins and Traits in Swift 2.0
   - http://matthijshollemans.com/2015/07/22/mixins-and-traits-in-swift-2/
6. Mocks in Swift via Protocols
   - http://blog.eliperkins.me/mocks-in-swift-via-protocols

Chapter 6

Generics

I hope everyone enjoyed the protocol-oriented programming chapter, and I hope you have your playground ready. This chapter is going to cover Swift generics and we are only going to scratch the surface. In the first section, we will discuss the basics of Swift generics. This will include the functional paradigms in Swift with generics. We will then end the section with a discussion of generic type constraints.

The second half of this chapter will discuss more protocol-oriented goodness with associated types. Associated types are how we can achieve generic members through the use of protocols. These associated types have their advantages and disadvantages.

What You'll Learn

Generics are essential to coding in Swift, albeit a bit arcane. You have already worked with generics in Swift; you just might not have known it. We are going to examine the basics of Swift's generics by creating our own generic classes and structs. Swift is very functional by nature, and this functionality works through the use of generics. We are going to focus on functionality for sequences here. Afterward, we are going to use generics with type constraints. Type constraints are similar to protocol extension constraints in functionality.

We will then transition back to Swift protocols, specifically, associated types. *Associated types* are abstract members that, essentially, create generic protocols. After walking through this section, you should feel comfortable creating and using associated types in your protocols. Let's get started!

Swift Generics

> *Generic code enables you to write flexible, reusable functions and types… and expresses its intent in a clear, abstracted manner.*
>
> —Swift Programming Language

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_6

I think the preceding quote says it all about generics. There are times when we want our code to be very specific and precise. However, more often, we want to write our code once and just have it work. Generics give us the ability to accomplish this. Generics are different from the common `Any` or `AnyObject` types. These two types are actually protocols and just represent the concept of anything or anything that is a reference type.

These two protocols, however, do not persist type information. That means, once we have a variable that is of type `Any`, the compiler only see this variable as an `Any` and we cannot use string functionality, even if it was instantiated as a string. Let's start our new playground page and reexamine this problem with code:

```
//: # Generics

//: #### Ability to print a string and only a string

func printString(str: String) {
    print(str)
}

let someString = "hello"

// This works!
printString(str: someString)
```

The previous code should work just fine; so let's break this code. In the next code block, we have set the type of `someString` to be an `Any` type:

```
//: # Generics

//: #### Ability to print a string and only a string

func printString(str: String) {
    print(str)
}

let someString: Any = "hello"

// This does not work
printString(str: someString)
```

If you change this in your playground, it will not compile. Please remove this and make sure your playground page compiles before moving on. So, why does this break? Well, in the first code block, the compiler inferred our `someString` variable as a string type. The `printString` function then only allows string types. When we explicitly set `someString`'s type in the second block, we set it to `Any`. The `Any` type is not the same as a string, and the compiler cannot see `someString` as a string type anymore. Therefore, it cannot be treated as a string type.

This might seem awfully simple, but this is the problem generics are solving. We want the ability to treat something generically, but we don't want to lose functionality. Now before we

start building generic structures, how would we change the previous example to work by using generics? Here is the previous example with a new generic method:

```
//: #### A generic function that can print any type

func printSome<Type>(item: Type) {
    print(item)
}

let myInteger = 1
let myString = "goodbye"

printSome(item: myInteger)
printSome(item: myString)
```

Now we have this `printSome` function and it has a parameter named `item`. The type of the `item` parameter is defined within the signature of the function. This is how generics are defined in functions, classes, structs, and methods. In Swift, the generic type is required to appear as parameter or a return value. Below our function, we create two variables that have two different types, but our function still works. This is a generic function. It is important to note that we could have solved this without generics, but this is just a quick introduction. The next section is going to build our very own generic structure.

Classes and Structs

Let's start with where you have used generics before. If you have ever used an array or dictionary in Swift, you have used generics. Many of the underlying data structures provided by Apple are generic. It makes sense for these to be generic since we want an array to work for any type, not just strings or integers. We are going to build our own generic `Stack` struct in this section. Previously you saw how to define a generic function. The syntax is the same for a generic class or struct. Let's define a `Stack` struct and give it a generic type called `Type`:

```
//: # Generic Stack

struct Stack<Type> {
    private var list: Array<Type> = []
}
```

You will usually see `T`, `U`, or `S` used for generic types, but I used `Type` to be more explicit about its purpose. The struct then has an array called `list`. Since Swift's `Array` is a generic type, we must specify the type of the `Array`. This line can also be written as follows:

```
private var list: [Type] = []
```

Next, let's add an initializer for our struct. We want to allow a developer to inject the items for the stack on initialization. Let's also make the `init` variadic, so we can inject more than one object:

```
struct Stack<Type> {
    private var list: [Type] = []

    init(items: Type...) {
        self.list = items
    }
}
```

You will notice, we do not need to specify the type using angle brackets. The struct itself is generic, so there is a struct level generic type and we do not need to specify the type again. Next, let's give this struct `add` and `pop` methods.

```
struct Stack<Type> {
    private var list: [Type] = []

    init(items: Type...) {
        self.list = items
    }

    mutating func add(items: Type...) {
        list += items
    }

    mutating func pop() -> Type? {
        guard list.count > 0 else { return nil }
        return list.removeLast()
    }
}
```

These two methods modify the structure, so we must declare them as `mutating`. We use the generic `Type` throughout the code where we usually expect a concrete type. This might not seem very cool, but wait until we start using this struct. Let's define three separate `Stack` structs in our playground, giving each one a separate type like the following:

```
var stack1 = Stack<String>()
var stack2 = Stack<Int>()
var stack3 = Stack<Array<Int>>()

stack1.add(items: "hello", "goodbye")
stack2.add(items: 1, 2, 3)
stack3.add(items: [1, 2], [3, 4])

stack1.pop() // Returns "Goodbye"
stack2.pop() // Returns 3
stack3.pop() // Returns [3, 4]
```

We just created generic data structure. The power behind this struct lies in the usage. We could have declared our struct with an array that holds onto `Any` types. However, when we go to retrieve any items, we would have to cast that value to the specific type we want. We then have to deal with optionals. By using generics, the `Stack` already knows what type it holds and then we can directly access any `item` as the correct type.

Many languages, some of which are very functional in nature, have influenced Swift. These functional paradigms would be very difficult to use if Swift did not have support for generics. The next section is going to expand on our generic knowledge and examine these paradigms in the collection types in Swift's Foundation Framework.

Functional Paradigms with Generics

Let's start this section by creating our new playground page: *Ch06—Functional Generics*. The idea behind functional programming is to avoid changing state. This can be accomplished through the use of functions/methods that take in data and spit it back out in a transformed state. This is not to be confused with modifying state. The internal states of the objects being used by the function are not modified. Let's go to our playground and see how these work on the `array` collection type in Swift.

First, create a new variable that is a `[String?]` type:

```
let array1: [String?] = ["hello", nil, "goodbye"]
```

Let's first examine the `map` method on array. This method has one parameter, a closure called `Transform`. The purpose of the `map` method is to transform a sequence of types. The closure parameter describes the conversion. We can transform our `array1` so all strings are prefixed with `Transform`:

```
let newArray1 = array1.map { str in
    return "Transform \(str ?? "")"
}

newArray1 // ["Transform hello", "Transform", "Transform goodbye"]
```

You will notice that the second string in the `newArray1` says, "`Transform`" with no value. The original value in `array1` was `nil`, so there should be no value. This is not what we want. We want there to be no `nil` values in the final result. There is another functional method called `filter`; let's try that:

```
let newArray2 = array1.filter { str in
    return str != nil
}
.map { str in
    return "Transform \(str ?? "")"
}

newArray2 // ["Transform hello", "Transform goodbye"]
```

Awesome! We have only two values in the previous result `newArray2`. So how does this work? Each of these methods is generic. The array already contains the type information, and in this case, it is `String?`. This type is then injected into the `map` or `filter` methods and in the case of `map`, it is expecting another type out. That means we can chain multiple maps or filters and we retain our type information.

Now, if you have used functional programming before, you are probably yelling at me that there is a better way than using `filter` and `map`. And you would be correct. There is another nifty method called `flatMap`. This method will actually omit any `nil` values and the result is a nonoptional sequence. Try out `flatMap` for yourself and see what happens.

Functional programming can be really cool. It also lends itself very well to Swift's value types and generics. We can chain `filter`s and `map`s and other functional methods to quickly transform a collection or perform complex tasks with very few lines of code. This would not be possible without the use of generics.

Generic Type Constraints

We have seen the basic usage of generics. We have seen how to utilize generics for functional programming. There is still one last problem to talk about. We want to be generic, but not so generic we cannot accomplish our task. This is where generic type constraints come into play. Through the use of type constraints similar to the ones used in Chapter 5, we can keep our generic type, but constrain the object to a specific type.

Let's examine type constraints by creating a new playground page and a new struct called `Pair`. This struct is actually going to hold onto two different generic types. The implementation is as follows:

```
struct Pair<T, U> {
    let key: T
    let value: U
}

let p1 = Pair(key: "hello", value: 1)
```

This struct holds a key value pair, like a dictionary. If we created an array of these objects, we would achieve an implementation similar to a dictionary. Now, let's add two generic type constraints:

```
struct Pair<T: Equatable, U: Equatable> {
    let key: T
    let value: U
}
```

We are now constraining our two generic types, `T` and `U`, to conform to the `Equatable` protocol. This means, whatever type `T` and `U` are, they have to conform to this protocol. So, what's the benefit of these constraints? Well, let's make our `Pair` struct conform to this same protocol:

```
struct Pair<T: Equatable, U: Equatable>: Equatable {
    let key: T
    let value: U
}
```

This will cause a compiler error until we have the following code in place:

```
func ==<T, U>(lhs: Pair<T, U>, rhs: Pair<T, U>) -> Bool {
    return true
}
```

Here we are overloading the double equals (==) operator and making it a generic function. We then inject two different `Pair` objects whose generic types are `T` and `U`. Currently, we are just returning `true`, so this will compile, but let's think about what the implementation would be if we had not added the previous type constraints. We cannot compare a generic type because the compiler will not know if it is possible. The generic type constraint clears this ambiguity up, thus making it possible to implement an equality operator for this struct:

```
func ==<T, U>(lhs: Pair<T, U>, rhs: Pair<T, U>) -> Bool {
    return lhs.key == rhs.key && lhs.value == rhs.value
}
```

Putting all of this together, we have a generic `Pair` struct that can only contain `Equatable` types and itself is `Equatable`. These constraints allow us to write a generic comparison function. All of the code for the playground page *Ch06—Generic Type Constraints* is as follows:

```
import Foundation

struct Pair<T: Equatable, U: Equatable>: Equatable {
    let key: T
    let value: U
}

func ==<T, U>(lhs: Pair<T, U>, rhs: Pair<T, U>) -> Bool {
    return lhs.key == rhs.key && lhs.value == rhs.value
}

let p1 = Pair(key: "hello", value: 1)
let p2 = Pair(key: "hello", value: 1)

let p3 = Pair(key: "goodbye", value: 2)

p1 == p2 // True
p1 == p3 // False
```

So what have we learned in this section? We have gone over the basics of Swift generics, we have examined some functional paradigms that are implemented with generics, and we have discussed generic type constraints. Next up, we are going to discuss protocol associated types.

Protocol Associated Types

> *Unlike classes, structs, and enums, protocols don't support generic type parameters. Instead they support abstract type members; in Swift terminology Associated Types.*
>
> —Russ Bishop

As the preceding quote says, protocols cannot be *generic* in the traditional sense. Instead, they can contain generic members. This section describes how to create protocols with these generic members, which is a core piece to protocol-oriented programming. I wanted to include it in this chapter so you had some background on generics beforehand. The following code describes a `ViewModelContainer`:

```
import Foundation

class ViewModel {
    func doSomething() {
        print("A")
    }
}

protocol ViewModelContainer {
    associatedtype VM: ViewModel
    var viewModel: VM { get set }
}
```

First, we have declared a `ViewModel` class. We then declared our protocol and gave it an associated type. The associated type is called `VM` and the right-hand side is telling the Swift compiler this associated type is constrained to types that inherit from our base `ViewModel` class. This type information will then be preserved on any object that conforms to this protocol. Then, the `viewModel` property is declared with type `VM`. Next, let's add a `set` method so our protocol will now look like the following:

```
protocol ViewModelContainer {

    associatedtype VM: ViewModel
    var viewModel: VM { get set }

    mutating func set(viewModel vm: VM)
}
```

We have encountered the `mutating` keyword before. We have to use it here because our protocol is going to modify internal state. Since value types can conform to protocols, this is required on the protocol method declaration. The way to circumvent this safety is to mark a protocol as a `class` protocol. In the protocol declaration, if you use the `class` keyword like so—`protocol Trait: class {...}`—it is not necessary to mark methods as `mutating`, as

classes are not value types, but reference types. Now that that's out of the way, back to your regularly scheduled programming. The next step is to declare an extension and give it the functionality we want:

```
extension ViewModelContainer {
    mutating func set(viewModel vm: VM) {
        viewModel = vm
    }
}
```

Now, why the associated type? We could just have our property be a base `ViewModel`. Read on:

```
class SomethingViewModel: ViewModel {}

class Something: ViewModelContainer {
    var viewModel: SomethingViewModel

    init() {
        viewModel = SomethingViewModel()
    }
}

let s = Something()
s.viewModel
```

And presto! The power of the associated type is that we do not need to hold onto our `ViewModel` as a base `ViewModel`. We can have the specific type we want. In the previous case, it was a `SomethingViewModel`, which, in a real app, should add more functionality specific to this class.

This is why associated types are so powerful, but why you might run into some arcane errors using them. The downside of this power comes when you want to reference an associated type protocol without specific type information. Do not add the following code to your playground because it will not compile:

```
let container: ViewModelContainer = Something() // This will not compile
```

This line will give you this error: '*ViewModelContainer*' *can only be used as a generic constraint because it has Self or associated type requirements*. Let's examine why this is the case. First, we are using our `Something` class that is a `ViewModelContainer`. All we want to do is reference this container's `viewModel` property because we do not care what the concrete type really is, just that it is a `ViewModelContainer`.

This is pretty normal for protocols, but guess what, that associated type member makes this impossible. The protocol essentially does not have enough type information because the associated type is relying on a concrete implementation. We take away the concrete implementation of our `Something` class and the `VM` type is unknown to the compiler.

If we look back at the error from the previous line of code, it tells us the protocol `ViewModelContainer` can only be used as a generic constraint. We cannot declare a type with this protocol, but we can create generic functions with the type constrained to this protocol. Let's add one of these functions to our playground:

```
let s = Something()
s.viewModel

func viewModelDoSomething<T: ViewModelContainer>(subject: T) {
    subject.viewModel.doSomething()
}

viewModelDoSomething(subject: s)
```

This function uses the view model that is contained within the parameter subject. In this case, the compiler has enough type information and can give us access to the base view model functionality. We can also use this protocol in other protocol extension type constraints. That does it for this section. We have seen how to implement associated types in protocols and the constraints they put on our code.

Wrap Up

Another chapter bites the dust. We have just scratched the surface of Swift's generics. There is a lot more to know about generics, but it is out of scope for this book. We have gone through the basics of Swift generics and some slightly more complex usages. We have seen functional paradigms in Swift and how they use generics to create some pretty powerful features.

We then discussed generic type constraints. The constraints can keep our structures and functions generic but at the same time give us enough information to create useful features. We saw this when we create a generic `Pair` struct. This originally had zero constraints, which made it impossible to write an equality operator for it. Once we constrained the generic types to the `Equatable` protocol, we were able to implement the equality operator. Because there is so much more to generics, I hope you do more research and keep learning about it. I know I am still learning about generics. The next chapter is going to discuss Autolayout and constraints in iOS. We will look at the coding side of Autolayout as well. See you there.

Articles

1. Generics
   - https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Generics.html
2. Type Variance in Swift
   - https://nomothetis.svbtle.com/type-variance-in-swift
3. Swift: Associated Types
   - www.russbishop.net/swift-associated-types

Chapter 7

iOS UI and Storyboards

Welcome to Chapter 7. You are almost finished with the first section of this book. Only two chapters remaining! This chapter is going to be a bit different than our previous chapters. So far, we have been able to write code in our playground and that's been it, but in this chapter, we are going to be working, partially, in Interface Builder! Don't worry, we will not be starting a new Xcode project and fragmenting our references. We are going to keep our storyboard file in our playground, but more on that later.

What You'll Learn

This chapter will be discussing Auto Layout and constraints. We are going to look at how to set up constraints in our storyboard and how to set up constraints in our code. If you have never coded constraints, you may find that it can be difficult to understand at first; in fact, when I began, I found it to be a nightmare! My hope for this chapter is to save you from that. We are going to examine the trait variations in Xcode 8 and how we can set up different constraints based on different devices, orientations, and other UI differences. We will finish up the chapter discussing some storyboard tips and tricks to keep in mind to make your development go faster.

Auto Layout and Constraints

> *Auto Layout dynamically calculates the size and position of all the views in your view hierarchy, based on constraints placed on those views.*
>
> —Apple, Inc.

This section is going to be half in Interface Builder and half in code. We are going to set up our storyboard in our playground and build two sample interfaces. The first sample interface we are going to build cannot be created with the *Add Missing Constraints* button. The second half of this section will look at how we can set up views with constraints via Swift

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_7

code. We will examine the API and just how easy it can be to create complicated views. I also cannot pass up an opportunity to talk about protocols, so we will take a closer look at a protocol we have already seen in Chapter 5. First, what is *Auto Layout*?

Auto Layout is a huge topic and there is a lot of math involved. The important part of the math behind Auto Layout is that it represents each constraint as a separate equation. According to Apple Inc. at `https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/AnatomyofaConstraint.html`, the goal is to create a view that has a series of equations that have only one possible solution. When there are multiple possible solutions, you will get an ambiguous view. Figure 7-1 is an image from the preceding link at Apple showing a sample constraint equation.

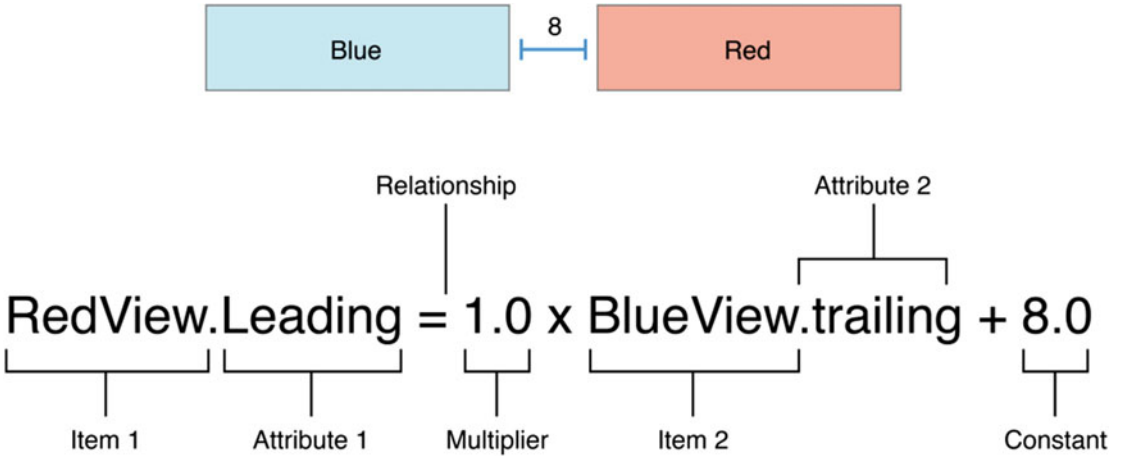


***Figure 7-1.** An image from Apple Inc. showing a sample constraint equation*

Using these equations, Auto Layout can then define the size and position for every element. Ultimately, this is its entire purpose, to specify the X, Y, width, and height of a particular element. It can be really difficult to do this when you factor in all the different devices, orientations, and other UI differences. In Figure 7-1, you can see all the information constraints have on Interface Builder. If you select a constraint on a view, you will see a menu with this information. Later in this chapter, the constraint menu can be seen in Figure 7-10. This has been the briefest of introductions to the inner workings of Auto Layout. Now let's start building our first sample view in Interface Builder.

Interface Builder

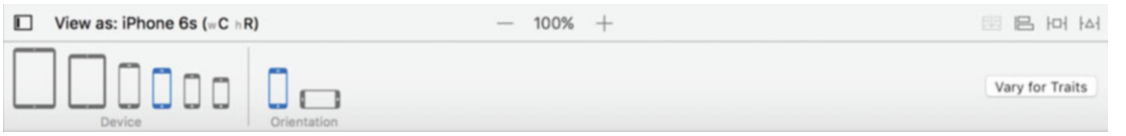
Let's start by adding our storyboard file to our playground. It's not as easy as right clicking our Resources folder and tapping *New File*. Start by opening Xcode and closing out any existing playgrounds, projects, or workspaces. Once you see the Xcode launch screen, in the top menu bar, tap *File* ➤ *New* ➤ *New File*. You will be able to create a new storyboard file and just put it on your Desktop or anywhere that is easily accessible. Now, click and drag the new storyboard file into your playground's Resources folder. You will not be able to open the storyboard and add view controllers and UI elements.

> **Caution** This is not a compiled storyboard file. You will not be able to link view controllers/UI elements to classes. If you want to compile your storyboard, follow this link: http://ericasadun.com/2015/03/25/swift-todays-wow-moment-adding-menus-to-playgrounds/.

I mentioned earlier that we would be building sample interfaces. The first sample I have for you is from the future app we are going to start building in Chapter 9. Figure 7-2 shows one of the interfaces for our Grocery List app.

Figure 7-2. *The interface for adding a grocery list in our future app*

Our interface in Figure 7-2 has a label, a text field, and two buttons. The buttons are fixed at the bottom of the screen and the title and text field are fixed to the top of the screen. The two buttons are also the same width and height. This is the first interface we are going to build and, as you can see, it is a portrait design. We will not worry about landscape just yet. Hop on over to our playground storyboard and drag a new view controller onto the storyboard if you have not done so. Way back in Chapter 2, Figure 2-1 showed the new pane at the bottom of storyboards. I have provided the figure here a second time, so you do not have to flip back. I have selected the smallest size available to me to make the figures easier to see, but feel free to play with these settings as we go.



New pane available at the bottom of a storyboard file for seeing your layout in all devices and orientations

Let's start building the interface from Figure 7-2. First we need to drag a new label and text field onto our view. Figure 7-3 shows the result of this action. I have also changed the settings for the label and text field. Figures 7-4a through 7-4d show the attribute and size inspectors for our label and text field.

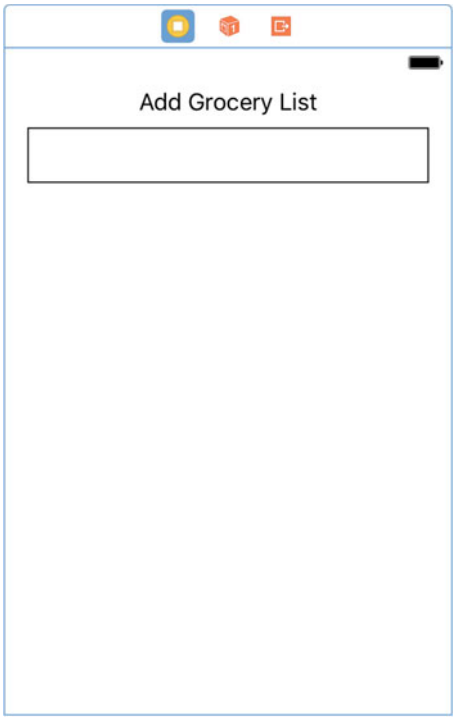


Figure 7-3. *The view controller with our new label and text field added*

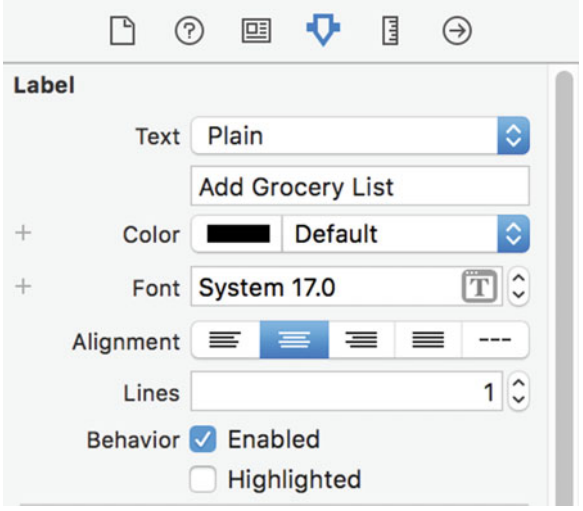


Figure 7-4a. The Attribute Inspector for the label

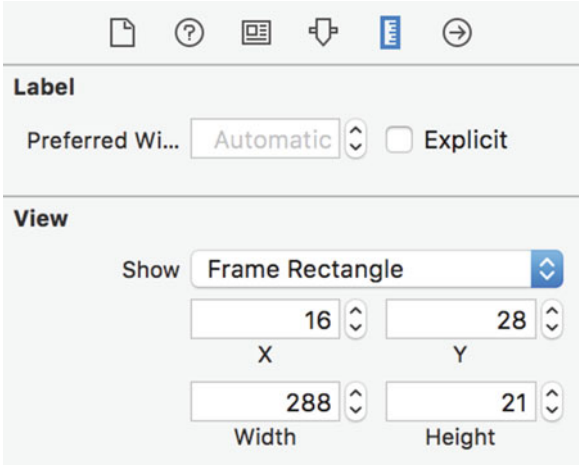


Figure 7-4b. The Size Inspector for the label

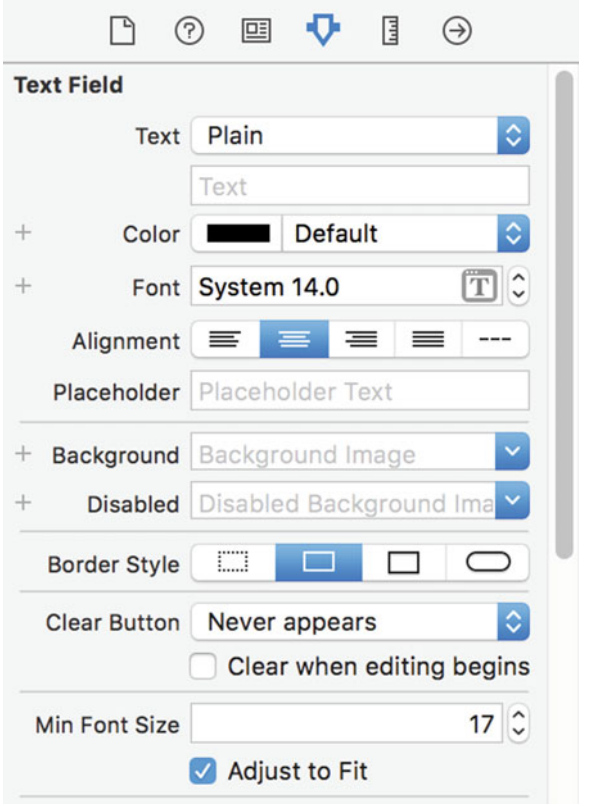


Figure 7-4c. *The Attributes Inspector for the text field*

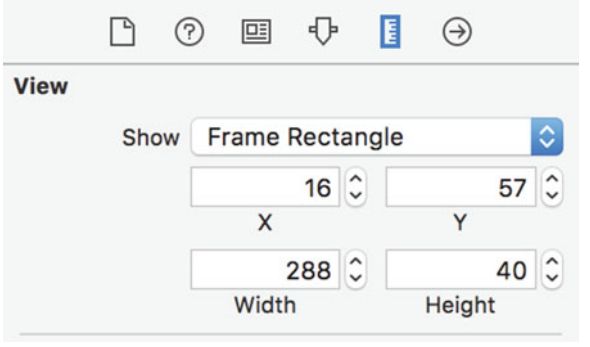


Figure 7-4d. *The Size Inspector for the text field*

Now, that we have our first elements, let's start adding constraints. Your first instinct might be to tap *Add Missing Constraints*. This action can be really helpful when you're starting out and give you some idea how the constraints are going to look. I want you to refrain from using this action, though. I also want you to try adding constraints by hand before I show you the constraints I used. You will not be able to run your storyboard since we are in a playground, but you can flip between the different device sizes, and this can give you a good idea of what your constraints are doing. Figure 7-5 shows our view after tapping *Add Missing Constraints* and moving from the smallest device to an iPhone 6+.

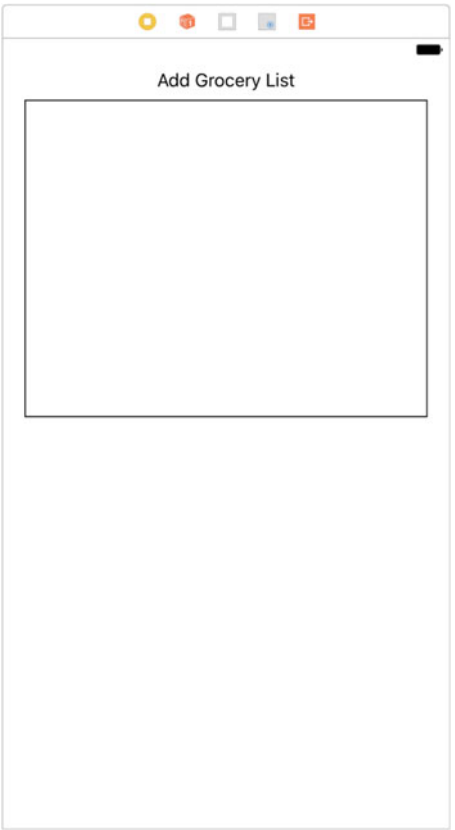


Figure 7-5. *Our view after choosing Add Missing Constraints and changing the display to iPhone 6+ instead of iPhone 4s*

You can see the constraints that were added were close, but are not exactly what we want, and this just gets more complicated as we add more elements to our views. Hopefully, you have tried to add the constraints yourself by hand. Now, I am going to step through the constraints I used. I added four constraints to my title label from the Pin menu shown in Figure 7-6. Now, if you resize your view, you can see the label reacting correctly to the screen resizing. The text field, however, does not, so let's fix that next.

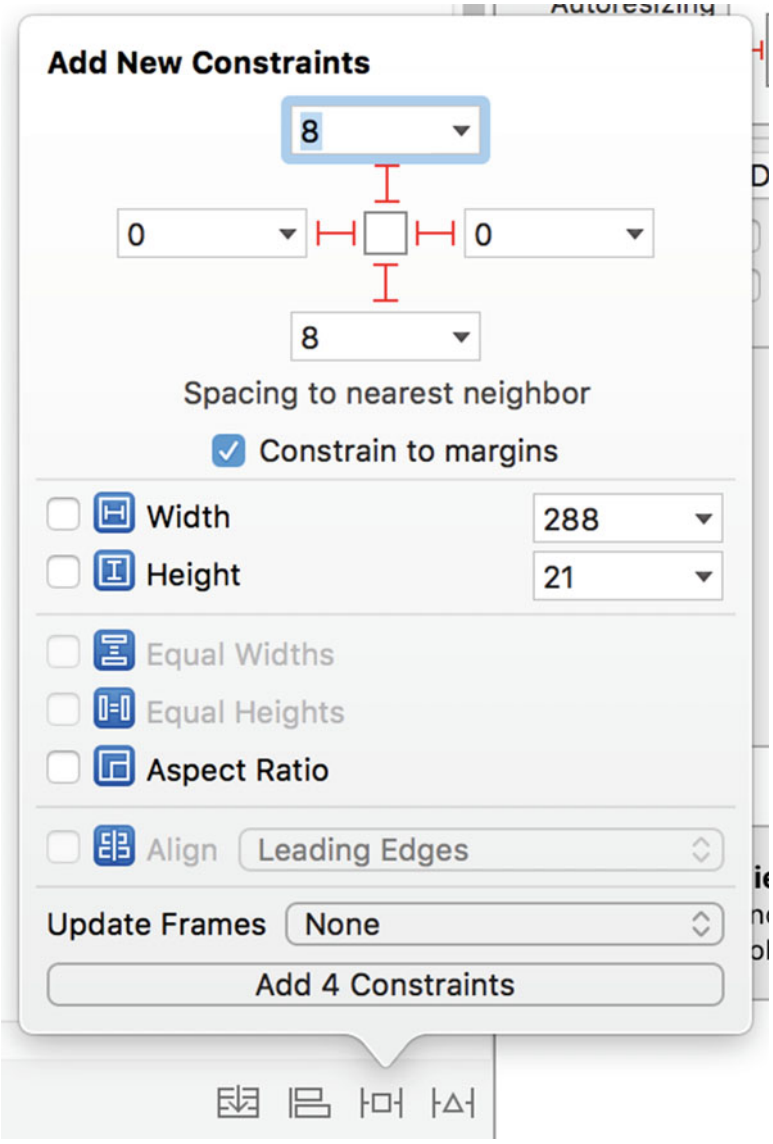


Figure 7-6. The Pin menu and the four constraints I selected to add to the title label

Now, we should have enough constraints on our title label. Figure 7-7 shows the Size Inspector panel and the four constraints on our title label. Our title label has a Trailing Space to Superview constraint, a Leading Space to Superview constraint, a Bottom Space to our text field with a constant of 8 constraint, and a Top Space to the bottom of the Top Layout Guide with a constant of 8 constraint.

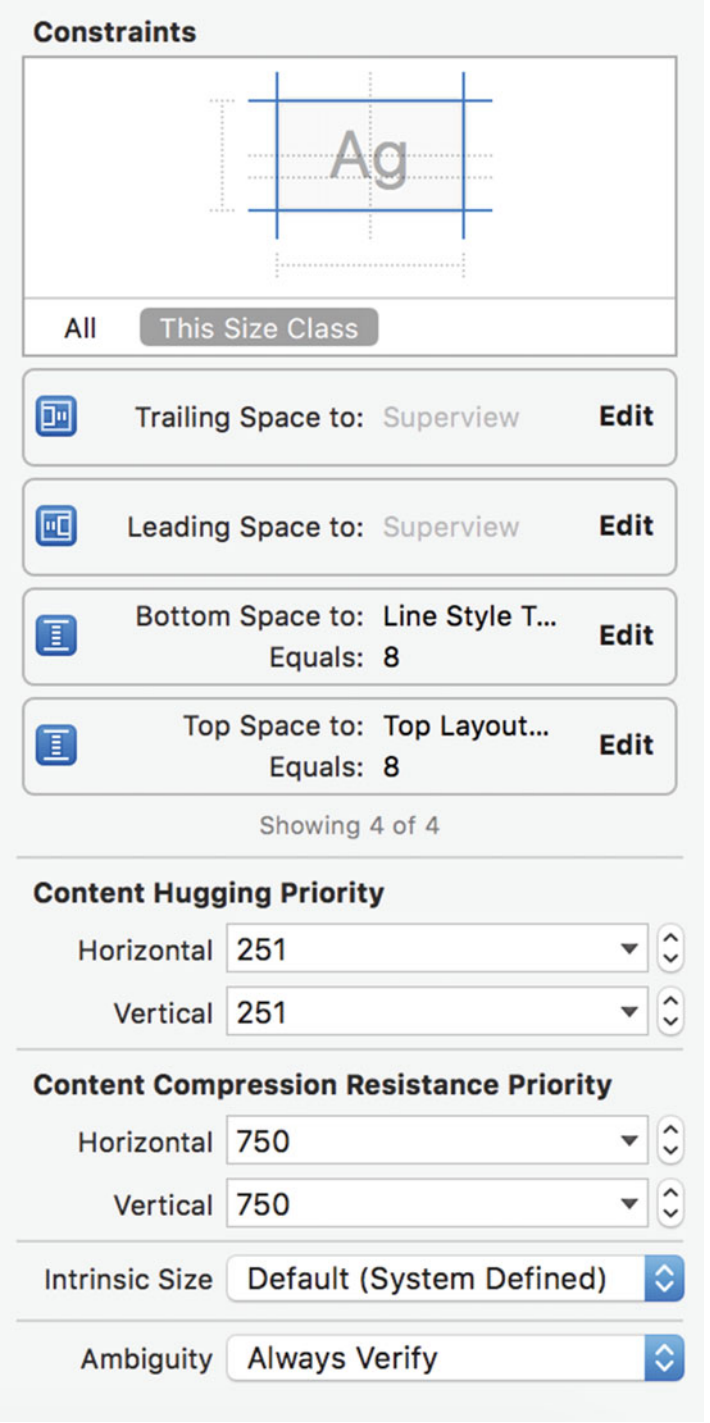


Figure 7-7. All four constraints on our title label

Constraints can be difficult to talk about. I just listed out the constraints and you can see them here, but what does all of this information mean? Remember when we were discussing the purpose of Auto Layout? The only purpose of constraints is to specify the X, Y, width, and height of a particular UI element. The Leading Space and Top Space constraints specify the X and Y coordinates. The Trailing Space constraint then specifies the width of our element and the Bottom Space constraint then handles the height of the element.

Knowing this information, let's move on to the text field. If we look at the Pin menu again, in Figure 7-8, we can see that the top, left, and right all look like the same constraints, but the bottom constraint is giving me the value 383. You might see something different depending on the device you have selected. If I were to use this constraint, the text field would always be 383 pixels away from the bottom of the view. Therefore, if our view grows, so does our text field. This is not what we want, so let's select the three other constraints.

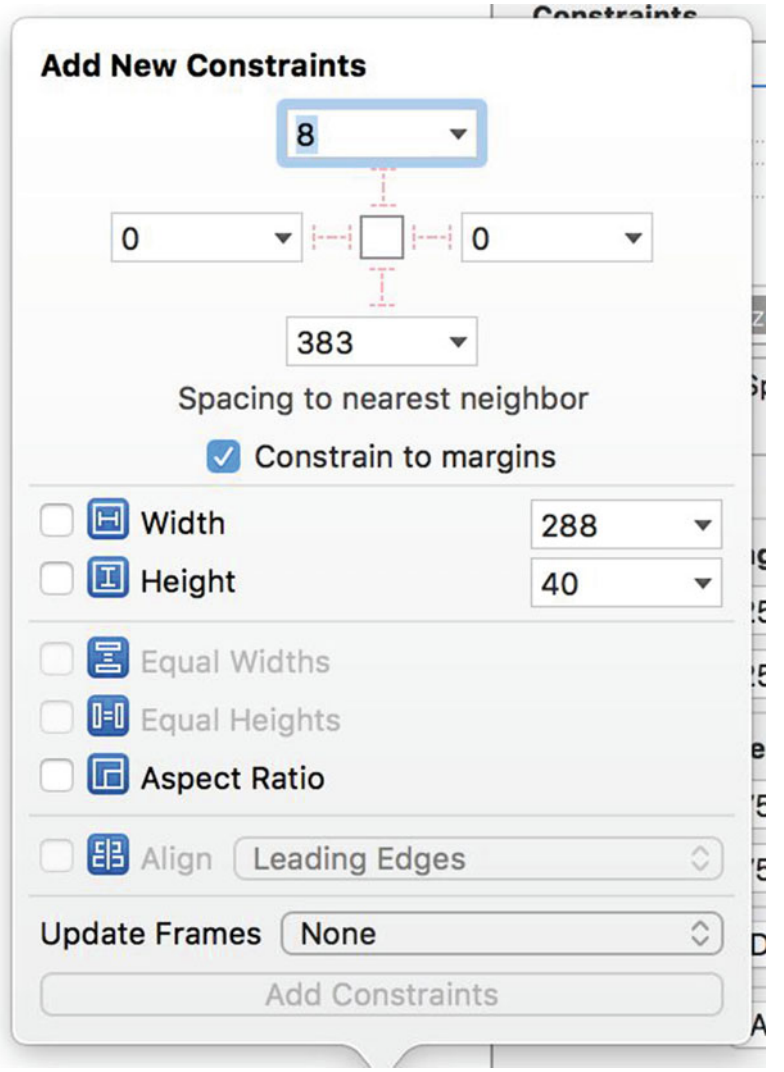


Figure 7-8. The Pin menu while our text field is selected

We now have three constraints on our text field, but this is not enough information for our text field to know its X, Y, width, and height. Let's examine what constraints we have applied. We have a Trailing Space to Superview constraint, a Leading Space to Superview constraint, and a Top Space to our title label with a constant of 8 constraint. Basically, we have the X, Y, and width defined, but no height is defined. The last constraint in Figure 7-8 with a constant value of 383 would have defined the height, but in the wrong way. We can also define the height for our text field by defining a Height constraint. Figure 7-8 shows this option directly below the *Constrain to Margins* option. Let's add this constraint. Figure 7-9 shows all of the constraints on our text field after adding the Height constraint. Now, there is no more ambiguity on our view. If you choose different devices in the bottom pane, you can see both of our UI elements resizing correctly.

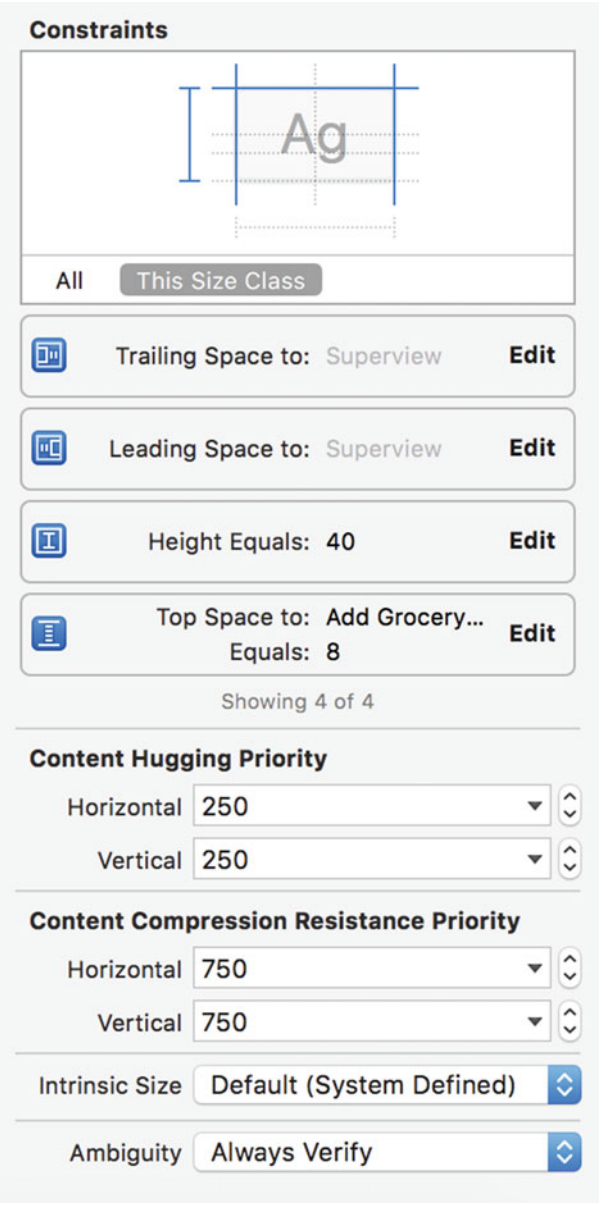


Figure 7-9. *All four of our constraints on our text field*

We are now going to add the last two elements from Figure 7-2, the two buttons. The requirements for the two buttons are (1) they will be fixed to the bottom of the view, (2) the widths are the same, and (3) the heights are the same. Let's start by adding a new button to our view and setting the text to ***Cancel***. We can then press Alt+Option to duplicate the element. Set this new button's text to ***Add List***. This next part is going to be really cool. Select the *Cancel* button, open up the Pin menu, add a Leading Space to Superview constraint, and add a Bottom Space to the top of the Bottom Layout Guide with a constant of 20 constraint.

Now, add the mirror constraints to our *Add List* button. Those constraints would be the Trailing Space to Superview constraint and another Bottom Space constraint. Now, select both buttons, select the Pin menu, and add an Equal Widths constraint. We have one constraint left. Ctrl-drag from the *Add List* button to the *Cancel* button and add a Horizontal Space constraint. Nothing is going to change on our view, so select the constraint by clicking on the blue line between the two buttons. Figure 7-10 shows the Horizontal Space constraint.

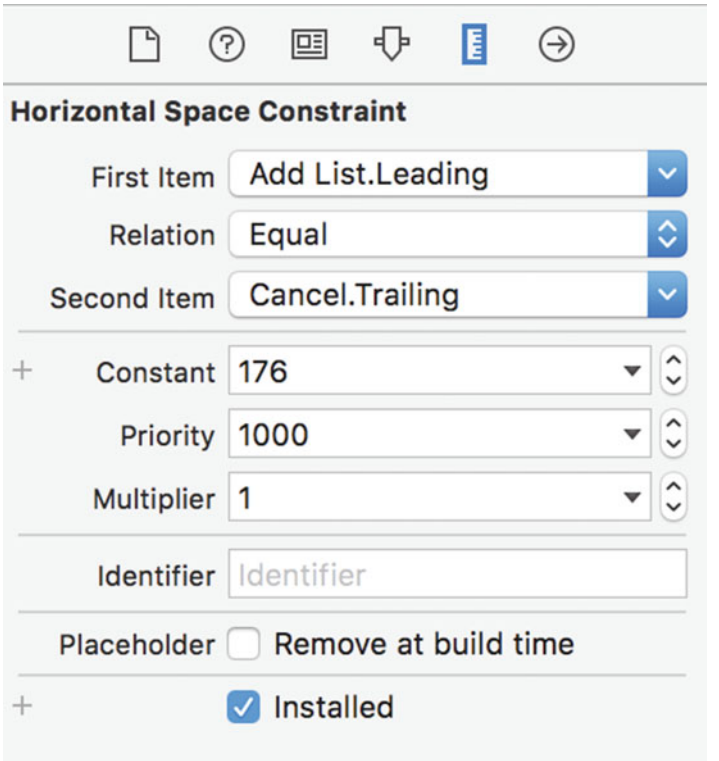


Figure 7-10. *The Horizontal Space constraint*

The *Constant* on my view is set to 176. Let's set this to something more reasonable like 15. Once we set the *Constant* property, our view should update, and due to the Equal Widths constraint and our two buttons, it should resize appropriately. That will about do it for our view. There are more configurations we can do, but the important parts are done. Let's move on to the next section, where we will code constraints.

Constraints in Code

There is going to be a lot of code in this section. We are going to build the view from Figure 7-11 in our playground using constraints instead of frames. This section will also look back and expand on a protocol I introduced in Chapter 5, the `canBlockView` protocol. This protocol will also show up again when we are building our app. Let's start with our first view. Figure 7-11 shows the view we are going to build in our playground.

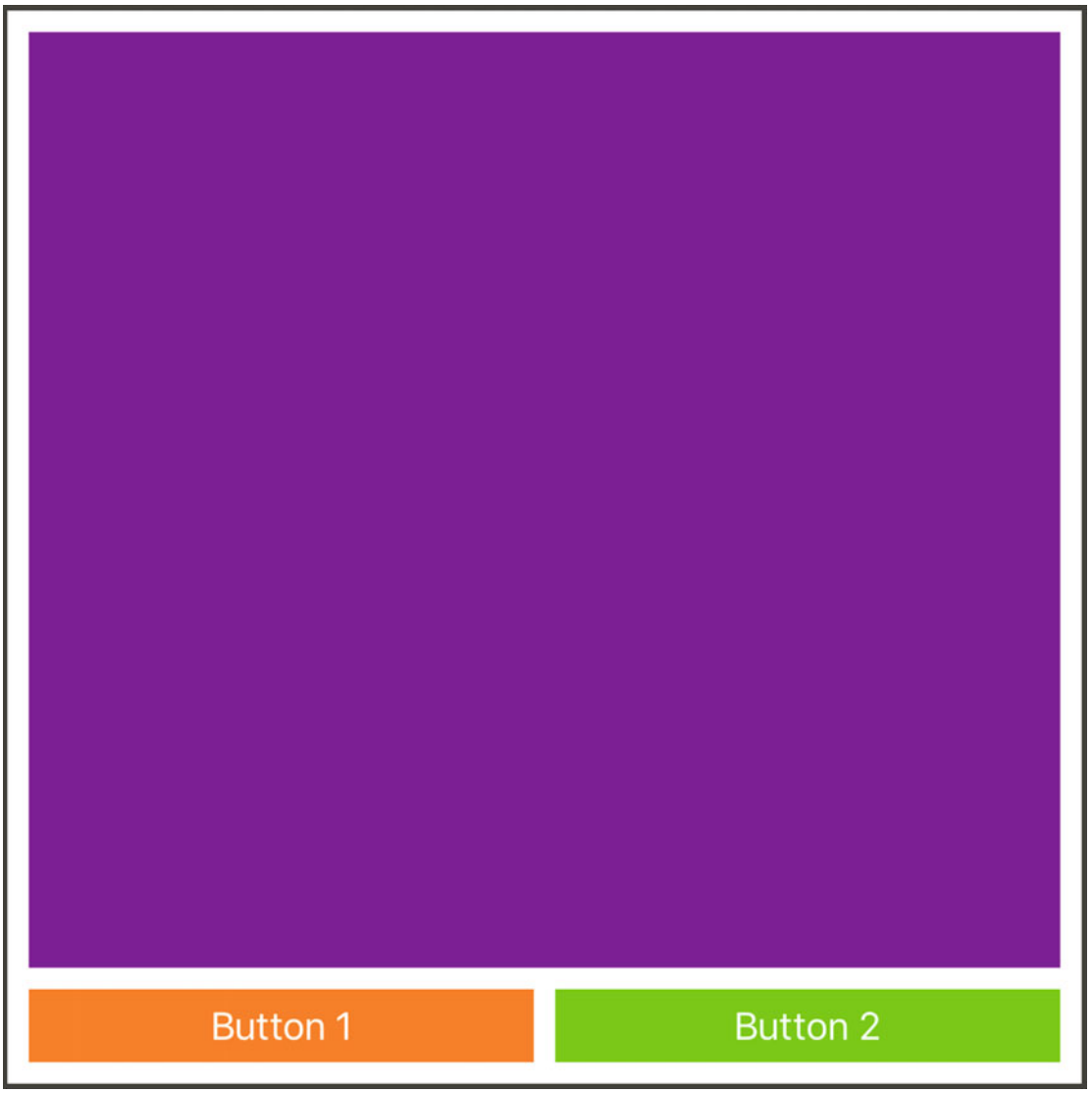


Figure 7-11. *The next view we are going to build via code instead of Interface Builder. We are going to use constraints to build to view instead of frames.*

This is a pretty simple view, but there is a lot of code behind it. Start a new playground page and let's start coding. First, make sure you have the frameworks, UIKit and PlaygroundSupport imported. Let's add the following code to start out view:

```
import UIKit
import PlaygroundSupport

let view = UIView(frame: CGRect(x: 0, y: 0, width: 500, height: 500))
view.backgroundColor = .white
PlaygroundPage.current.liveView = view
```

We have a square view with a white background and we then set the playground's live view to this view. This is how we can see and create views in our playgrounds. Figure 7-12 shows how to turn the live view on in our playground. Open the Assistant Editor, which is represented by the two interlocking circles at the top right, then make sure *Timeline* is selected instead of *Manual*.

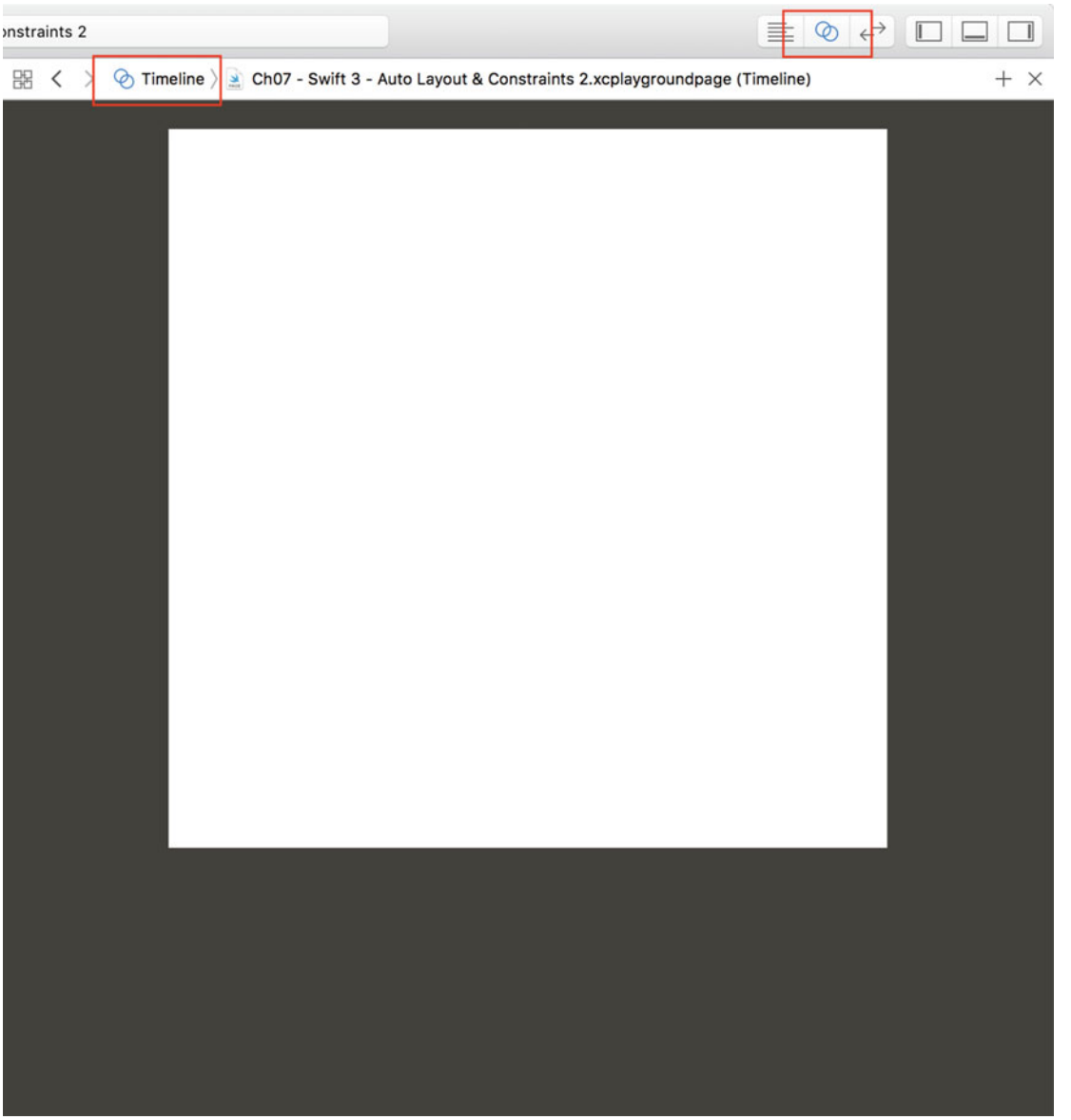


Figure 7-12. *The Assistant Editor open and our Timeline selected instead of Manual to display the live view in our playground*

Now, that our playground is set up, let's add our three UI elements. The following code is adding our two `UIButtons` and one `UIView`:

```
let orangeButton = UIButton(frame: .zero)
orangeButton.backgroundColor = .orange
orangeButton.setTitle("Button 1", for: .normal)

let greenButton = UIButton(frame: .zero)
greenButton.backgroundColor = .green
greenButton.setTitle("Button 2", for: .normal)

let purpleView = UIView(frame: .zero)
purpleView.backgroundColor = .purple

view.addSubview(orangeButton)
view.addSubview(greenButton)
view.addSubview(purpleView)
```

We have created our two buttons and view and we have given them all a frame of `zero`. Once this code is added, nothing is going to happen on our view because all of their frames are `zero`. The next step is critical. We must set the property `translatesAutoresizingMaskIntoConstraints` to `false` for all three of our elements. Here is the documentation for this property:

> *A Boolean value that determines whether the view's autoresizing mask is translated into Auto Layout constraints.*
>
> —UIView Class Reference

Remember in Chapter 2 when I said the old springs and struts system never left? Well, here it is. If we set this property to `true`, the old system will take priority and any Auto Layout constraints we add will create conflicts and not work. I always forget this step, so I wanted to make sure I called it out for you. You have to set this property as `false` any time you want to create constraints in code:

```
orangeButton.translatesAutoresizingMaskIntoConstraints = false
greenButton.translatesAutoresizingMaskIntoConstraints = false
purpleView.translatesAutoresizingMaskIntoConstraints = false
```

All right, we have all our elements set up and added to our live view. Let's add constraints to the `orangeButton`. Figure 7-11 is the view configuration we are shooting for, so you can try adding the constraints yourself first, or just follow along. If you want to try it yourself, a

good place to start might be Interface Builder. When I started writing constraints in code, I would create my view in Interface Builder and then translate the constraints into code from there. Just an suggestion. Let's add the leading, bottom, top, and trailing constraints to the `orangeButton`:

```
var constraints: [NSLayoutConstraint] = []

constraints += [
    orangeButton.leadingAnchor.constraint(equalTo: view.leadingAnchor, constant: 10),
    orangeButton.bottomAnchor.constraint(equalTo: view.bottomAnchor, constant: -10),
    orangeButton.trailingAnchor.constraint(equalTo: greenButton.leadingAnchor, constant: -10),
    orangeButton.topAnchor.constraint(equalTo: purpleView.bottomAnchor, constant: 10)
]

// All of our constraints will go here...

NSLayoutConstraint.activate(constraints)
```

I have created an `Array<NSLayoutConstraint>`, so I do not have to think of names for all of our constraints. The last line is where we tell the constraint system to activate all of our constraints. Any constraints we create from now on will go before this last line where I have our "All of our constraints will go here…" comment. Let's examine the constraints we have created here. We align our `orangeButton`'s `leadingAnchor` to the live view's `leadingAnchor` with a constant of 10 pixels.

> **Note** We will be using the new iOS 9 API in this chapter. In Chapter 1, we discussed the old API briefly.

The `bottomAnchor` is then constrained to the `bottomAnchor` of our live view. We then also supply a negative `constant` value here to offset the constraint by ten pixels, so our button will be ten pixels from the bottom of our live view. We then add our `trailingAnchor` to be constrained to the `leadingAnchor` of the `greenButton`. This will make sure our two buttons are always ten pixels away from each other. The last constraint we have defined is the `topAnchor` and we have constrained this to ten pixels away from our `purpleView`. Figure 7-13 shows the results of our constraints in our playground's live view.

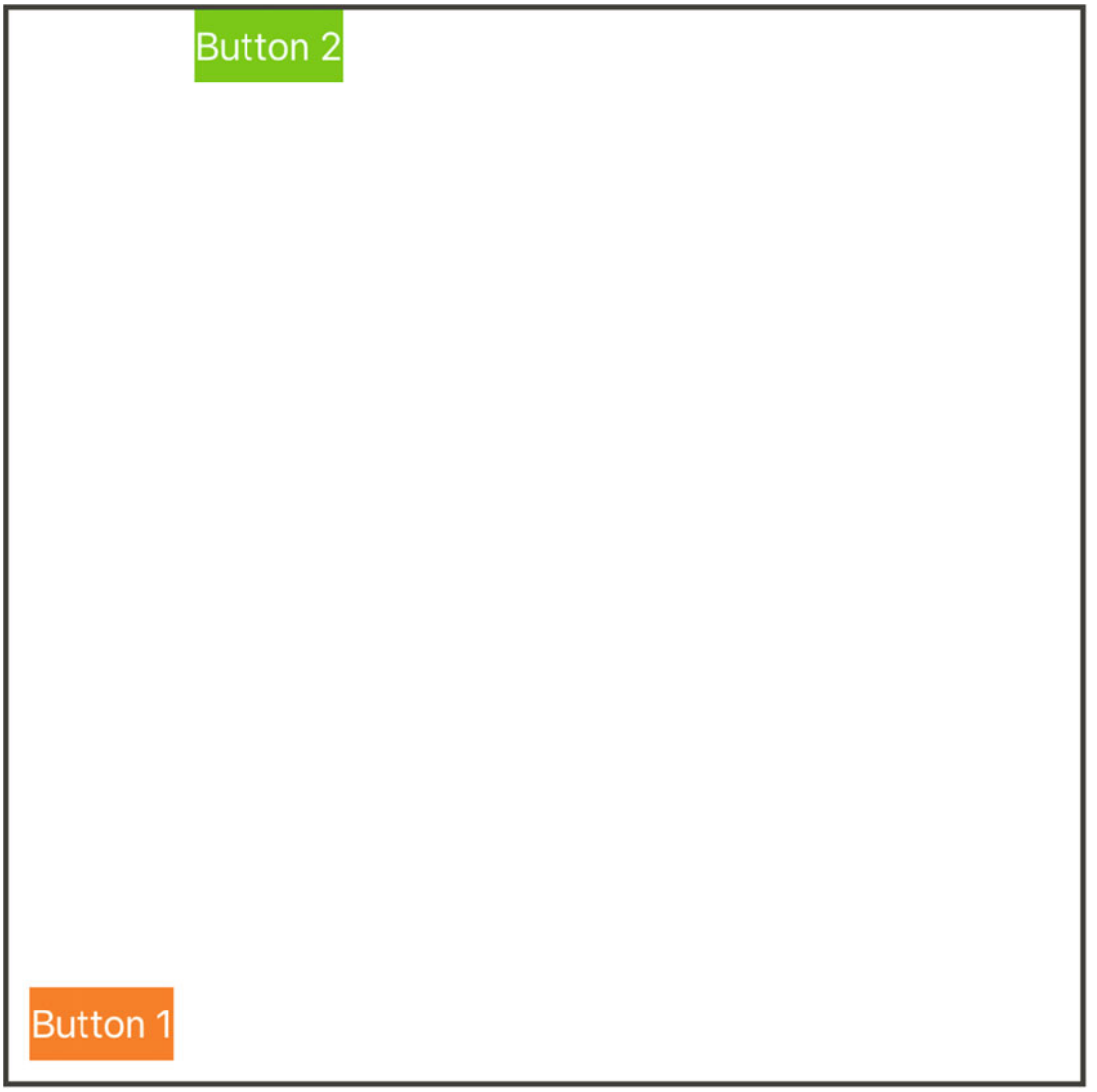


Figure 7-13. Our live view in our playground after applying constraints to our orangeButton

The `orangeButton` looks like it's in the right place, but our `greenButton` is just kind of floating. Let's add one more constraint to the previous definitions, so our `greenButton` will be closer to the final layout:

```
constraints += [
    orangeButton.leadingAnchor.constraint(equalTo: view.leadingAnchor, constant: 10),
    orangeButton.bottomAnchor.constraint(equalTo: view.bottomAnchor, constant: -10),
    orangeButton.trailingAnchor.constraint(equalTo: greenButton.leadingAnchor, constant: -10),
    orangeButton.topAnchor.constraint(equalTo: purpleView.bottomAnchor, constant: 10),
    orangeButton.firstBaselineAnchor.constraint(equalTo: greenButton.firstBaselineAnchor)
]
```

The new constraint is the last one in this array. We have aligned the `firstBaselineAnchor` of our `orangeButton` to the `firstBaselineAnchor` of the `greenButton`. Figure 7-14 shows the result of all of our constraints.

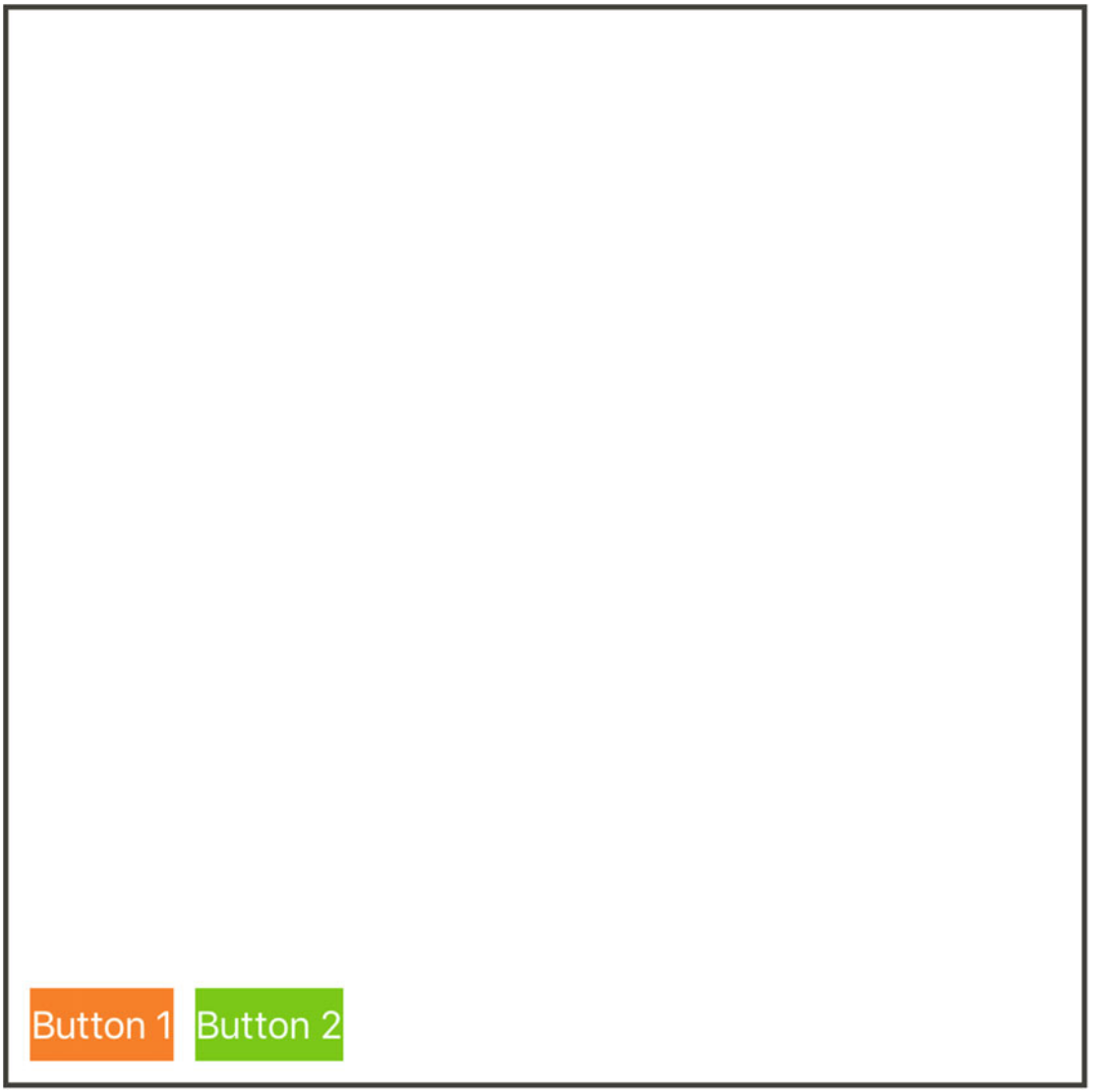


Figure 7-14. *All of our constraints including the firstBaselineAnchor constraint applied to the orangeButton*

The constraints for our two buttons are almost done. We need to add two more constraints, this time to our `greenButton`. The new constraints are as follows:

```
constraints += [
    greenButton.trailingAnchor.constraint(equalTo: view.trailingAnchor, constant: -10),
    greenButton.widthAnchor.constraint(equalTo: orangeButton.widthAnchor)
]
```

The constraints here tell the `greenButton`'s `trailingAnchor` to be constrained to our live view's `trailingAnchor` with a constant of ten pixels to offset the button just like our `orangeButton`. Our second constraint here then makes sure our two buttons have the same width. These constraints are all we need because the constraints for our `orangeButton` have defined the X and Y coordinates as well as the Height constraint for our `greenButton`. Adding the Trailing and Width constraints then defines the width of our buttons. Figure 7-15 shows our view with these new constraints.

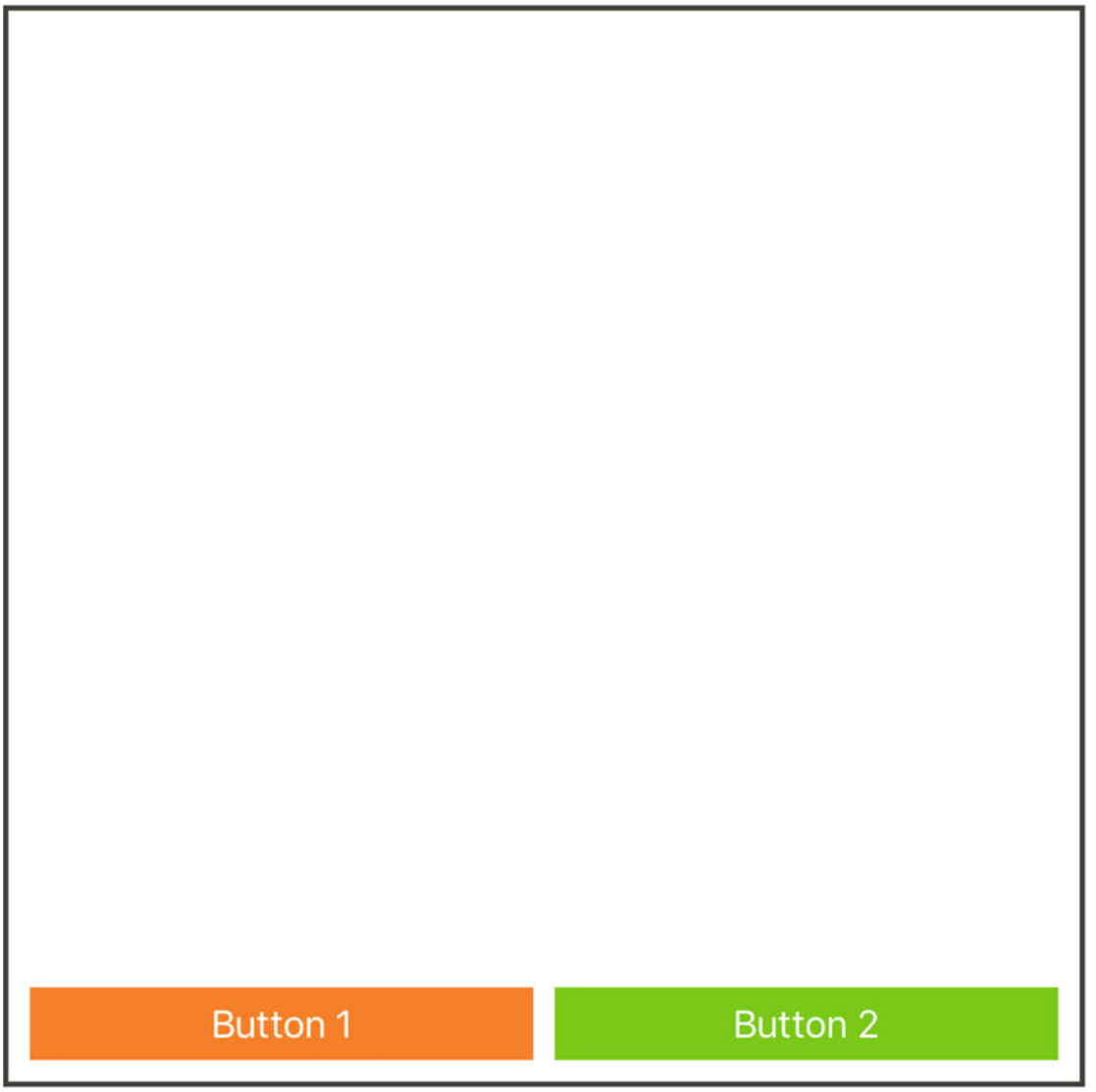


Figure 7-15. *The live view after our new constraints*

Last up is our `purpleView`. We will be able to get away with three constraints here. We have already defined the `bottomAnchor` for our `purpleView` in the constraints for our `orangeButton`. The following code is our last three constraints:

```
constraints += [
    purpleView.leadingAnchor.constraint(equalTo: view.leadingAnchor, constant: 10),
    purpleView.trailingAnchor.constraint(equalTo: view.trailingAnchor, constant: -10),
    purpleView.topAnchor.constraint(equalTo: view.topAnchor, constant: 10)
]
```

These last three constraints keep our purple view within the bounds of the view and the `bottomAnchor` is already set up to be constrained to the orange button. Figure 7-16 shows the final live view with all of our constraints applied.

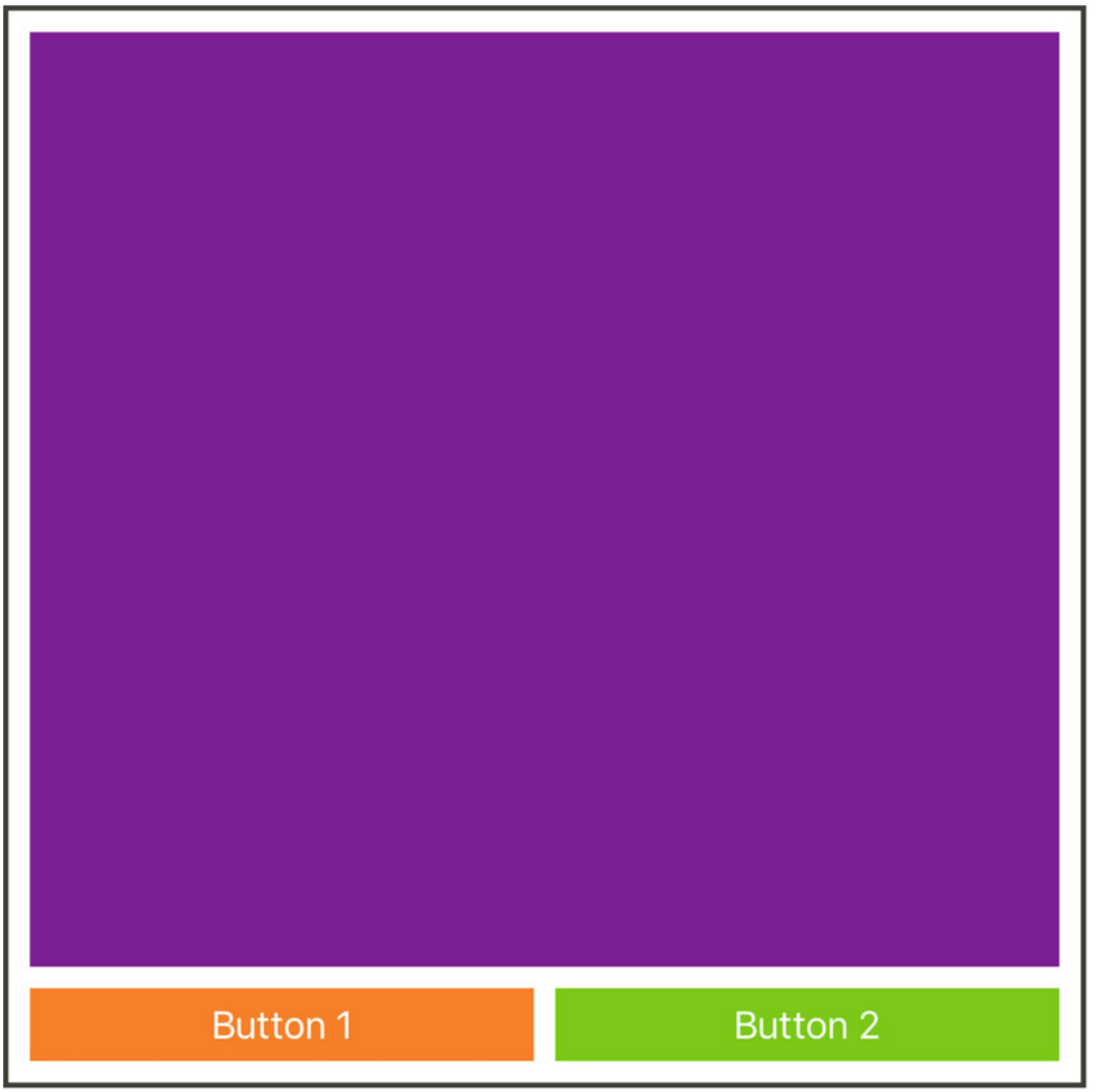


Figure 7-16. *The final live view with all constraints applied to our three UI elements*

We made it! Now, that we have all of our constraints set up, start changing the frame for the live view to see how the view adapts. It should automatically resize all subviews correctly. In the last part of this section, we are going to build the `canBlockView` protocol.

Blocking a View

Part of the reason I did not show you the implementation for this protocol when it first showed up was that it was not relevant to the chapter, but it also involved constraints. We can now implement this protocol. Here is the skeleton code for this section. I would highly recommend a new playground page:

```
import UIKit
import PlaygroundSupport

let view = UIView(frame: CGRect(x: 0, y: 0, width: 320, height: 640))
view.backgroundColor = .white
PlaygroundPage.current.liveView = view

protocol canBlockView {
    var view: UIView { get set }
    func blockUI()
}

extension canBlockView {
    func blockUI() {

    }
}

class ViewContainer: canBlockView {
    var view: UIView

    init(view: UIView) {
        self.view = view
    }
}

let container = ViewContainer(view: view)
container.blockUI()
```

> **Tip** Take this time to open the Assistant Editor and our Timeline to show the live view.

All right, we have our live view; we have the `canBlockView` protocol defined. In this example, I have added a `view` property to the protocol so we do not need to instantiate an actual `UIViewController` conforming to the protocol. We have an empty extension on our protocol and a `ViewContainer` class. This class conforms to our protocol so it must have a view and

we allow the view to be injected. We then instantiate the container with our live view and call `blockUI`. Currently, you should see a white screen. Figure 7-17 shows the final result of `blockUI` working successfully.

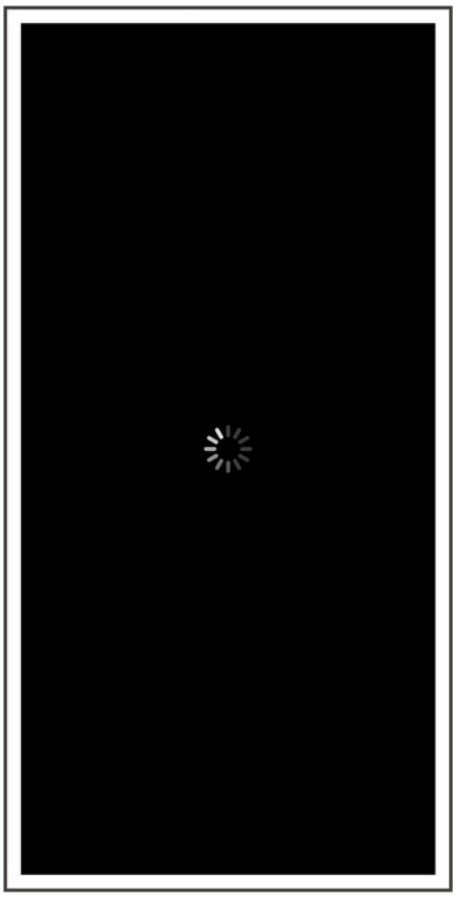


***Figure 7-17.** The playground's live view with blockUI running successfully*

In Figure 7-17, I have a black "blocking" view with a `UIActivityIndicatorView` in the center. I have added some padding to the edges of the "blocking" view so you can see the view behind it. Now, take some time and see if you can't code the constraints on this one. You will need a total of six constraints to have a view that blocks the other and a centered `UIActivityIndicatorView`. See what constraints you can apply to your blocking view and when you are ready to move on, here is the code for the `canBlockView` extension:

```
extension canBlockView {
    func blockUI() {
        let activityIndicator = UIActivityIndicatorView(activityIndicatorStyle: .whiteLarge)
        let blockingView = UIView(frame: .zero)

        blockingView.backgroundColor = .black
```

```
        activityIndicator.translatesAutoresizingMaskIntoConstraints = false
        blockingView.translatesAutoresizingMaskIntoConstraints = false

        blockingView.addSubview(activityIndicator)
        view.addSubview(blockingView)

        NSLayoutConstraint.activate(
            constraints(for: activityIndicator, on: blockingView) +
            constraints(for: blockingView)
        )

        activityIndicator.startAnimating()
    }

    private func constraints(for indicator: UIActivityIndicatorView, on blockingView:
    UIView) -> [NSLayoutConstraint] {
        return [
            indicator.centerXAnchor.constraint(equalTo: blockingView.centerXAnchor),
            indicator.centerYAnchor.constraint(equalTo: blockingView.centerYAnchor)
        ]
    }

    private func constraints(for blockingView: UIView) -> [NSLayoutConstraint] {
        return [
            blockingView.topAnchor.constraint(equalTo: view.topAnchor, constant: 10),
            blockingView.leadingAnchor.constraint(equalTo: view.leadingAnchor, constant:
            10),
            blockingView.bottomAnchor.constraint(equalTo: view.bottomAnchor, constant: -10),
            blockingView.trailingAnchor.constraint(equalTo: view.trailingAnchor, constant: -10)
        ]
    }
}
```

You might have coded yours a bit differently, but here is how I implemented `canBlockView`. I instantiated both of the UI elements. Then, I set the `translatesAutoresizingMaskIntoConstraints` property to `false` and added both of the elements to the live view. You have to add these elements to the view before you create the constraints. The constraints depend on the two views having some sort of hierarchical relationship. Then we just activate all of our constraints and make sure the activity indicator starts animating. The `UIActivityIndicatorView` will not show up if we do not call `startAnimating`. That's it for the protocol. Pretty simple, but powerful. We can constrain our protocol to work with `UIViewControllers` or some other protocol with a `view` property. Then we can just start blocking UI anywhere we have one.

In this section, we have successfully applied constraints in Interface Builder and created a view with constraints in code. We also implemented our old `canBlockView` protocol with constraints. I hope you have a greater understanding of Auto Layout and constraints now. If this has been more review than new for you, don't worry; we will be exploring advanced constraints in our next section.

Trait Variations

In this section, we will be working in our storyboard file again. We are going to build a view that will adapt to the orientation of the device, the size of the device, and other UI differences. Traits are not a new concept to Xcode 8 and iOS 10. They are just now stepping into the spotlight. The new device configuration pane from Figure 2-1 has the button *Vary for Traits* on the far right. This button puts Interface Builder into a state where any new constraints or configurations are based on the selected size class. This is invaluable when building our adaptive layouts.

This section is going to be a bit different than the last section. I am going to give you an overview of the layout we are going to build, but there are too many constraints and steps to reasonably walk through. It would also create a ton more figures. Instead, in this section I am going to show you how to changes traits and apply different constraints for different size classes. I then provide a table near the end of this section that lists all of the constraints and variations to build this view.

The view we are going to build here is going to contain a text view, two buttons, and a stack view containing three buttons. The idea of the view is that it's an entry form for some app. We want to maximize the amount of space the user has to enter text, so we do not want the other elements getting in the way or shrinking our entry field. We also need to make sure it's very easy for the user to tap both buttons and interact with the buttons in our stack view. Figure 7-18 shows the sample view we will build in this section in portrait and landscape orientations.

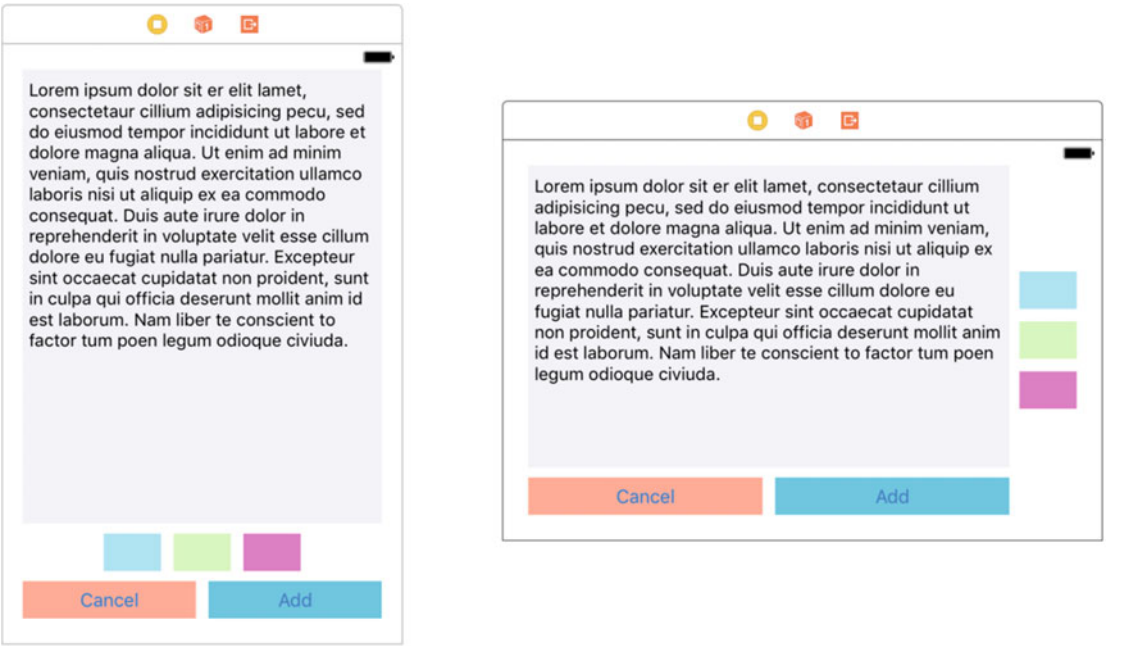


Figure 7-18. *The new view we are going to build in portrait and landscape orientations*

You can see in Figure 7-18 how we have maximized the space for our text view. We have also placed the buttons in good positions for the user to tap them easily. It might not seem like it, but there is a lot here. There are constraints and properties that only apply in certain size classes. This next part is going to cover how to modify these traits and then I am going to show you the table with all of the constraints and views so you can re-create this view.

Start by dragging a new view controller onto our storyboard, and then add a text view on the same view controller. Now, select the text view and select the *Attributes Inspector* in the right pane. If you look at the font color and font size attributes, you might notice a little plus (+) sign to the left of the attributes. Clicking this button will present a popover view to introduce a new trait variation. Figure 7-19 shows this popover.

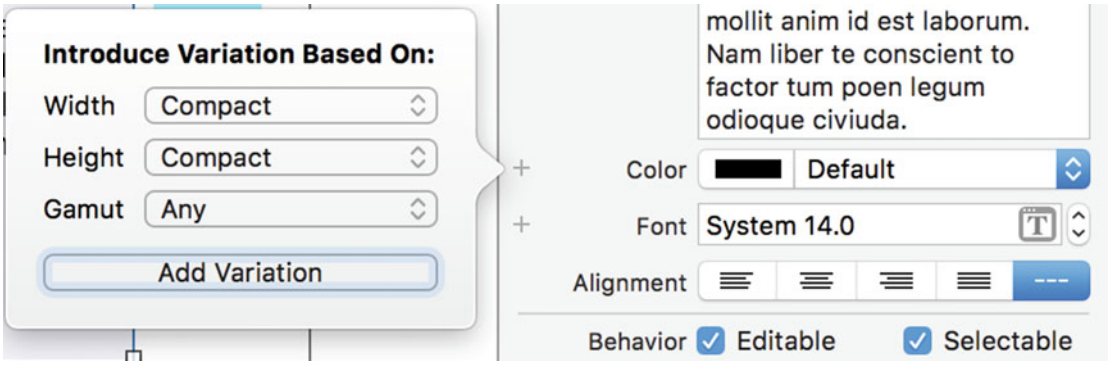


Figure 7-19. *The trait variation menu for the font color of a text view*

We have three options on our menu including *Width*, *Height*, and *Gamut*. The *Gamut* setting is for the colors in our app. We will not worry about this setting here. Within the *Width* and *Height* menus, we have the options *Any*, *Compact*, and *Regular*. The combination of the width and height create the size classes the new trait variation would apply to. If we were to select Any Width and Compact Height, then the new font color would only apply to views where the Height is classifiable as Compact. You will use this menu extensively to create the variations we need for Figure 7-18.

Way back in Figure 7-10 where we defined our Horizontal Space constraint, you can see there is a plus sign (+) next to the installed property. Checking/unchecking this property will install and uninstall the constraint and we can apply trait variations on this property, just like for our font color. If we added a specific size class to a constraint and unchecked the default *installed* property, this would effectively uninstall the constraint where we are not that specific size class. This is key to building our new view.

Here is an overview of the interface we are going to build. The view contains a `UITextView`, three `UIButtons` embedded in a `UIStackView`, and two other `UIButtons` with the text Cancel and Add. The three buttons embedded in the stack view are all Width: 46 and Height: 30. I have also added Width and Height constraints to these buttons before embedding them. Once embedded, I give a ten-pixel Spacing to the stack view. The stack view also starts out with a horizontal axis. Figure 7-20 is an abbreviated version of Figure 7-1. Refer to the format of the constraint in this figure, as this is the format I will use in the following table.

The following table is the list of all the constraints and trait variations needed to build this view for portrait, landscape, and different devices. Each element is a section with all of the corresponding constraints. The constraints under portrait or landscape are the constraints that should be activated for that orientation. All of the constraints under portrait and landscape still need to be applied to the element. The order of the equation is also important, so be sure to pay close attention.

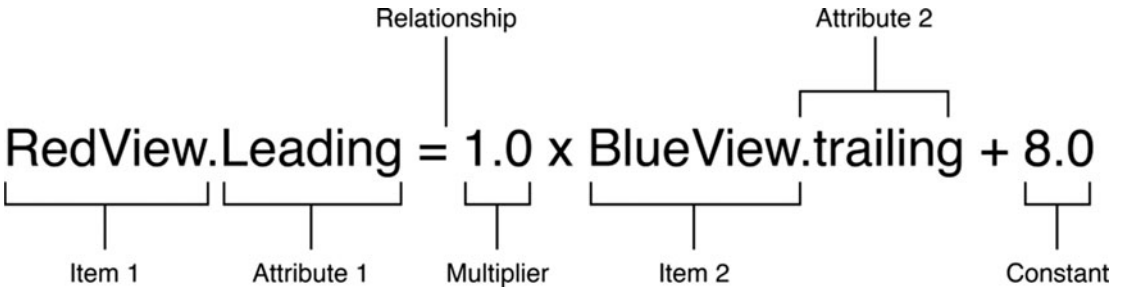


Figure 7-20. *An abbreviated version of Figure 7-1*

Applied Constraints

UITextView

Portrait	**Size Class**
`Superview.Trailing Margin = 1.0 × Text View.Trailing + 0.0`	wA hR
`Superview.Leading Margin = 1.0 × Text View.Leading + 0.0`	wA hA
`Text View.Top = 1.0 × Top Layout Guide.Bottom + 0.0`	wA hA
`Stack View.Top = 1.0 × Text View.Bottom + 8.0`	wA hR
Landscape	**Size Class**
Add.Trailing = 1.0 × Text View.Trailing + 0.0	wA hC
Superview.Leading Margin = 1.0 × Text View.Leading + 0.0	wA hA
Text View.Top = 1.0 × Top Layout Guide.Bottom + 0.0	wA hA
Stack View.Leading = 1.0 × Text View.Trailing + <Standard>	wA hC
Cancel.Top = 1.0 × Text View.Bottom + <Standard>	wA hC

UIStackView

Portrait		**Size Class**
Stack View.Center X = 1.0 × Superview.Center X + 0.0		wA hR
Cancel.Top = 1.0 × Stack View.Bottom + 8.0		wA hR
Stack View.Top = 1.0 × Text View.Bottom + 8.0		wA hR
Axis	Horizontal	wA hR
Spacing	10	N/A
Landscape		**Size Class**

Stack View.Center Y = 1.0 × Superview.Center Y + 0.0		wA hC
Superview.Trailing Margin = 1.0 × Stack View.Trailing + 0.0		wA hC
Stack View.Leading = 1.0 × Text View.Trailing + <Standard>		wA hC
Axis	Vertical	wA hC
Spacing	10	N/A
UIButton - Cancel		
Portrait		**Size Class**
Superview.Leading Margin = 1.0 × Cancel.Leading + 0.0		wA hA
Add.Leading = 1.0 × Cancel.Trailing + 10.0		wA hA
Bottom Layout Guide.Top = 1.0 × Cancel.Bottom + 20.0		wA hA
Cancel.Top = 1.0 × Stack View.Bottom + 8.0		wA hR
Cancel.Width = 1.0 × Add.Width + 0.0		wA hA
Landscape		**Size Class**
Superview.Leading Margin = 1.0 × Cancel.Leading + 0.0		wA hA
Add.Leading = 1.0 × Cancel.Trailing + 10.0		wA hA
Bottom Layout Guide.Top = 1.0 × Cancel.Bottom + 20.0		wA hA
Cancel.Top = 1.0 × Text View.Bottom + <Standard>		wA hC
Cancel.Width = 1.0 × Add.Width + 0.0		wA hA
UIButton - Add		
Portrait		**Size Class**
Superview.Trailing Margin = 1.0 × Add.Trailing + 0.0		wA hR
Add.Leading = 1.0 × Cancel.Trailing + 10.0		wA hA
Bottom Layout Guide.Top = 1.0 × Add.Bottom + 20.0		wA hA
Cancel.Width = 1.0 × Add.Width		wA hA
Landscape		**Size Class**
Add.Trailing = 1.0 × Text View.Trailing + 0.0		wA hC
Add.Leading = 1.0 × Cancel.Trailing + 10.0		wA hA
Bottom Layout Guide.Top = 1.0 × Add.Bottom + 20.0		wA hA
Cancel.Width = 1.0 × Add.Width		wA hA

> **Caution** The constraints will not display in the Size Inspector when they do not apply to the current trait collection (size class). I recommend switching between portrait and landscape and different devices while building this view.

The previous table contains all the constraints for all the UI elements on our view. Step through this slowly. There is a lot here. I have duplicated the constraints for portrait and landscape where the size class is listed as *wA hA*. When adding these variations, you can select the *Vary for Traits* button and select the trait you want to vary, *Width* and/or *Height*. Then add the constraints and variations. Xcode 8 will then automatically create the variations based on the selected size class.

This has been a relatively short section, but a lot has happened here. I hope you gave the trait variations a try first and also walked through the table and built a successful view that adapts to portrait and landscape orientations. It's been a long chapter and we are almost done. Next up, we are going to quickly go through storyboard tips and tricks.

Storyboard Tips and Tricks

This is going to be a quick section. We are going to discuss some cool features storyboards have available. The first cool features are custom views and gesture recognizers, all configured in our storyboard. The second set of cool features are the relatively new `@IBDesignable/@IBInspectable`. Let's jump right in.

Custom Views and Gestures

In the same storyboard we have been editing, let's add another view. Let's add this new view to our *Add Grocery List* view. Make sure you have selected a `UIView` element from the *Object Library* instead of the view controller. Then click and drag this new `UIView` directly next to the little yellow icon in the top bar of the view controller. Figure 7-21 shows the new

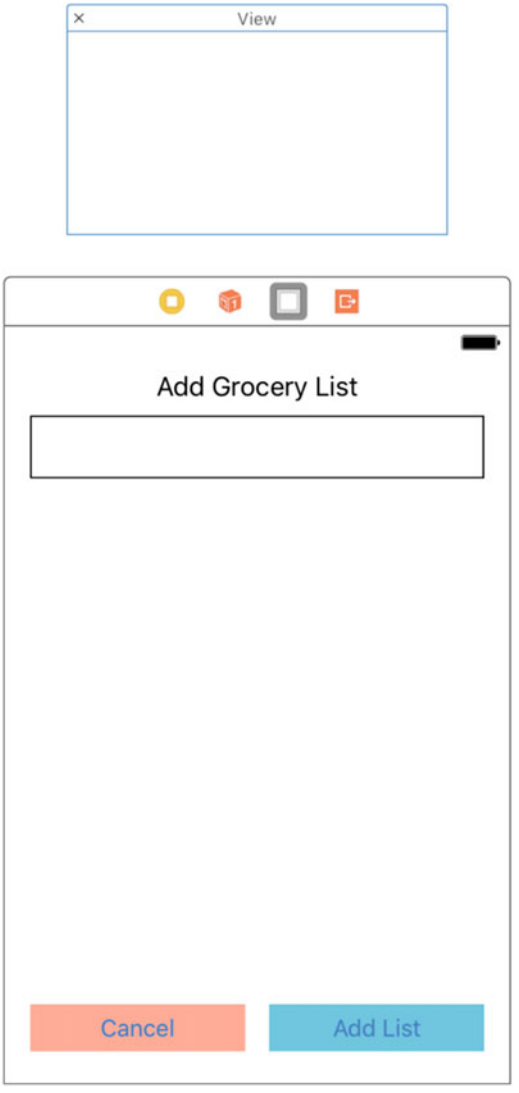


Figure 7-21. The new UIView added to our Add Grocery List view

view on our *Add Grocery List* view.

This feature allows us to create views in our storyboard using Auto Layout and constraints. These views are then not added to the base view on our view controller. However, we can create an `@IBOutlet` to the view and then add it later during the execution of our view controller. This can save a huge amount of time and coding, because we do not have to create this view in code and we can add as many extra `UIViews` to our view controller as we want. This feeds directly into the next cool feature of our storyboards, gestures.

We can add `UIGestureRecognizers` in the same way as the previous `UIViews`. If you search for "gesture" in the Object Library, you will receive a number of options including these:

- Tap Gesture Recognizer
- Pinch Gesture Recognizer
- Rotation Gesture Recognizer
- Swipe Gesture Recognizer
- Pan Gesture Recognizer
- Screen Edge Pan Gesture Recognizer
- Long Press Gesture Recognizer

Select any one of these gesture recognizers and drag it to any element on our view controller or custom view. I added a `UITapGestureRecognizer` to the base view on the view controller. We can then create an `@IBAction` in our view controller and link this to the tap gesture. This is really convenient when we have text fields and we want the user to be able to tap off the

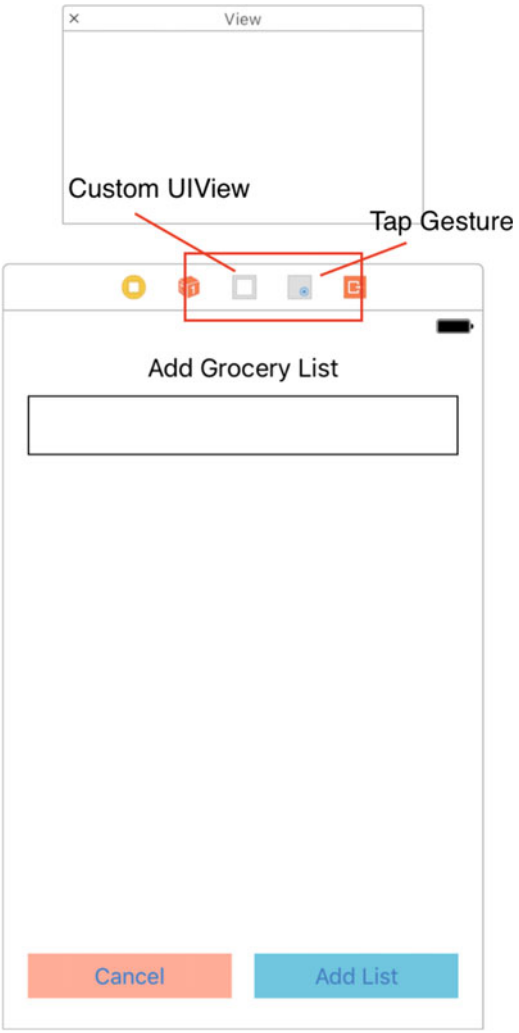


Figure 7-22. *Our Add Grocery List view controller with a custom view and a tap gesture*

field to close the keyboard. Figure 7-22 shows our view controller with a custom view and a tap gesture added.

That will do it for our custom views and gestures. They are really easy to use and can save

so much time and effort. Next, we will discuss the code side of things with `@IBDesignable/@IBInspectable`.

Designables and Inspectables

We are finally done with all the figures. Yay! There has been a lot in this chapter. We will be focusing on code for this section, so let's go to a new playground page and get started. First, create a new subclass of `UIButton` called `Button`:

```
import UIKit

class Button: UIButton {

}
```

The next step is adding the `@IBDesignable` attribute to the front of our class declaration. Now, `UIButton` should already be `@IBDesignable`, but it does not harm anything to add this attribute. We are then going to add one property called `cornerRadius` to our `Button` class. This property must then be marked with the `@IBInspectable` attribute. The following code shows these actions:

```
@IBDesignable class Button: UIButton {
    @IBInspectable var cornerRadius: CGFloat = 0.0
}
```

Adding these attributes to our class allows Interface Builder to interpret this class and add these options in the Attributes Inspector. We can then configure our views and our custom properties directly in Interface Builder. There is one last thing we need to cover here. This property is currently not hooked up to anything. The next code block is the standard way to hook up our new `cornerRadius` property:

```
@IBDesignable class Button: UIButton {
    @IBInspectable var cornerRadius: CGFloat = 0.0

    override func prepareForInterfaceBuilder() {
        super.prepareForInterfaceBuilder()

        layer.cornerRadius = cornerRadius
    }
}
```

The `prepareForInterfaceBuilder` method is called when compiling the storyboard. We can then link our `cornerRadius` property to the layer's `cornerRadius` property. However, there is an easier way we can set this up with the `prepareForInterfaceBuilder`.

The following code is how we can avoid using the `prepareForInterfaceBuilder` method. I have created a second class called `Button2` to illustrate this code:

```
@IBDesignable class Button2: UIButton {
    @IBInspectable var cornerRadius: CGFloat = 0.0 {
        didSet {
```

```
            layer.cornerRadius = cornerRadius
        }
    }
}
```

This uses Swift's property observers to immediately set the layer's `cornerRadius` property once we set our `cornerRadius` property. And that's it for this section. This has been a simplistic example, but I have used the `@IBDesignable`/`@IBInspectable` extensively to solve many problems and make my views configurable via the storyboard to save code.

Wrap Up

And that's a wrap. This chapter has had an absurd amount of figures. My hope is that you are more confident when building your interfaces with constraints now. I also hope you have a good understanding of trait variations and how to create different constraints for different layouts. We have also looked at some quick tips and tricks you can apply when using storyboards. We should also now be capable of configuring our views and adding tap gestures directly in Interface Builder.

Articles

1. Understanding Auto Layout
   - https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/index.html
2. Making Apps Adaptive, Part 1
   - https://developer.apple.com/videos/play/wwdc2016/222/
3. Making Apps Adaptive, Part 2
   - https://developer.apple.com/videos/play/wwdc2016/233/
4. Adaptive Interfaces Part 1: How UITraitCollection Changed Everything
   - https://possiblemobile.com/2016/07/adaptive-interfaces-uitraitcollection/
5. Auto Layout 101
   - www.weheartswift.com/auto-layout-101/
6. Swift: Today's Wow Moment. Adding Menus to Playgrounds
   - http://ericasadun.com/2015/03/25/swift-todays-wow-moment-adding-menus-to-playgrounds/

Chapter 8

Testing

This is the last chapter before we will start building our app. The goal of this chapter is to take you through testing in iOS. We are going to examine the idea of mocking and creating mocks. This will lead us into how to use Apple's XCTest framework and the idea of DRY and WET testing. Afterward, the section on the Swift Package Manager is going to expand on the Logger package from Chapter 3.

We will discuss test-driven development (TDD) in this chapter, including the benefits and the drawbacks. It is important to discuss TDD and other styles of testing because such development can be difficult in Swift/iOS and we must have reasonable expectations. We also want to test our code efficiently, without losing context; the last section in this chapter will examine how to find the balance here. This chapter is going to be a little light in the playground coding. We are going to add more to our playground in our first section, but the others sections are just more overviews and discussions.

What You'll Learn

This chapter is all about testing in Swift/iOS. In the first section, we are going to cover mocking in Swift. This is critical to testing in Swift because there is no way to automatically mock anything; it is all manual. We are going to build a simple mocking framework that we are going to use in the next few chapters.

We are then going to discuss Apple's unit testing framework, *XCTest*. XCTest is Apple's testing framework that is included with every Xcode project. We are going to discuss the API and what is available to you. After the introduction to XCTest, we are going to take a step back from coding and look at the concept of DRY tests and WET tests. These stand for *don't repeat yourself* and *well-expressed tests*.

After the more conceptual and stylistic discussion, we can move on to testing our Logger package from Chapter 3. This section is going to pull in techniques from Chapter 5 so we can actually test our package. This section will finish the discussion of the Swift Package Manager and our Logger package. Finally, this chapter is going to finish with *test-driven development (TDD)*. There are a lot of ideas, theories, and styles when it comes to TDD. We are going to discuss the practice and what it offers us as Swift/iOS developers. Let's get started with mocking!

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_8

Mocks

Mocking in Swift can be a real pain. Swift does not have any frameworks or any support provided by Apple to allow tests to access code in automatic mockable ways. What does this mean? In Java, there is a framework called *Mockito*. I do not fully understand how this framework works, but essentially it does a lot of work behind the scenes allowing it to intercept method invocations.

Swift, at a language level, does not allow this on a universal scale. There is something called *swizzle* where you can swap method implementations. This process has a lot of restrictions and we cannot use it reliably for testing purposes. Swizzling is also only possible where we can access the Objective-C runtime, such as `NSObject` and other C classes. Pure Swift objects and value types do not allow swizzling.

Therefore, there are no mocking frameworks that work like Java's Mockito. That does not mean we cannot mock objects though. We can write manual mocks by subclassing our objects in the test target. We then just need a way to record what happened when that mock was called. A framework called *MockFive* is a good framework to look at. MockFive is available on GitHub at https://github.com/DeliciousRaspberryPi/MockFive.

This framework facilitates testing and manual mocking by recording method invocations and the parameters that were passed into the mock. In this section, we are going to write our own lightweight version of this framework. We will then use this code in our app when we need to write mock objects. Open up your playground reference and let's get coding. We will start with a protocol:

```
protocol Mockable {
    var mocked: Mocked { get }

    func record(method call: String, with parameters: Any...)
    func value<T>(for call: String) -> T?
    func set(value: Any?, for call: String)
    func invocations(for call: String) -> Int
    func parameters(for call: String) -> [MockParameter]
}
```

Yay, protocols! First, we have a property that is of type `Mocked`. This will be the class that drives the data behind the mock. We then have a set of methods that will help facilitate our mock object. We can record the method call, set and get mock return values for a particular method, get the number of invocations of a particular method call, and finally get all of the parameters that were recorded. The `MockParameter` is an object that just helps us get the correct type out of the mock using generics. Let's create the `MockParameter` class and the stub for our `Mocked` class:

```
class MockParameter {
    var param: Any

    init(value: Any) {
        param = value
    }
```

```
    func value<T>() -> T? {
        return param as? T
    }
}

class Mocked {

}
```

You can see that the `MockParameter` class just holds onto the parameter value and has a generic method that will try to cast to our specific type. This is so we do not have to mess around with casting in our tests and we can be a bit more concise.

Next, let's fill in our `Mocked` class with an implementation. In fact, this object is going to have an almost identical API to our protocol. The `Mocked` object is also going to contain two arrays to record the invocations and hold onto our return values. I have setup two typealiases, `Invocation` and `ReturnValue`, for clarity. Let's see this class:

```
class Mocked {
    typealias Invocation = (signature: String, parameters: [Any])
    typealias ReturnValue = (signature: String, value: Any)

    var calls: [Invocation]
    var returnValues: [ReturnValue]

    init() {
        calls = []
        returnValues = []
    }

    func record(method call: String, with parameters: [Any]) {

    }

    func value<T>(for call: String) -> T? {
        return nil
    }

    func set(value: Any, for call: String) {

    }

    func invocations(for call: String) -> Int {
        return 0
    }

    func parameters(for call: String) -> [MockParameter] {
        return []
    }
}
```

I have added some default return values to get this class compiling. Other than that, we have a blank slate. You can probably see where I am going with this class, so how about you fill in the implementation and I'll wait… Awesome. The next blocks of code are my implementation:

```
class Mocked {
    typealias Invocation = (signature: String, parameters: [Any])
    typealias ReturnValue = (signature: String, value: Any)

    var calls: [Invocation]
    var returnValues: [ReturnValue]

    init() {
        calls = []
        returnValues = []
    }

    func record(method call: String, with parameters: [Any]) {
        calls.append( Invocation(signature: call, parameters: parameters) )
    }
```

Here, we have our `setup` and `record` method. The `record` method here takes two parameters, the method call as a string and the parameters for that method call. This is slightly different than the protocols implementation, though. The *parameters* parameter was variadic in our protocol. Here it is just an array. This is so we do not try to inject our variadic array as one parameter, which is what would happen if this was also variadic. Next, we have the implementation for the `valueForCall` method:

```
class Mocked {

    // ...

    func value<T>(for call: String) -> T? {
        return returnValues.first {
            $0.signature == call
        }?.value as? T
    }
```

This method is using a new Swift 3 Array API called `first`. This will select the first element it finds that satisfies our closure. We then just cast the returned value to our generic type. The last three methods of the `Mocked` class are `setValueForCall`, `invocationsForCall`, and `parametersForCall`. First let's look at `setValueForCall`:

```
class Mocked {

    // ...

    func set(value: Any, for call: String) {
        returnValues.append( ReturnValue(signature: call, value: value) )
    }

    func invocations(for call: String) -> Int {
```

```
        return calls.filter {
            $0.signature == call
        }.count
    }
```

These two methods just set up a new return value and then get the count for a particular method call. The `parametersForCall` method is the most complicated because we are returning an `Array` of our `MockParameter` object, not just the `Array<Any>`:

```
class Mocked {

    // ...

    func parameters(for call: String) -> [MockParameter] {
        let invocation = calls.first {
            $0.signature == call
        }
        return invocation?.parameters.map {
            MockParameter(value: $0)
        } ?? []
    }
}
```

Look through this carefully. It gets the `first` class that matches the signature, then `maps` the list of parameters to convert each one from an `Any` type to a `MockParameter`. Now, all that is left is to implement our protocol extension. Since the API is the same, you can probably guess the protocol methods are just a pass through to the `Mocked` property. Here is the code:

```
extension Mockable {
    func record(method call: String, with parameters: Any...) {
        mocked.record(method: call, with: parameters)
    }

    func value<T>(for call: String) -> T? {
        return mocked.value(for: call)
    }

    func set(value: Any?, for call: String) {
        mocked.set(value: value, for: call)
    }

    func invocations(for call: String) -> Int {
        return mocked.invocations(for: call)
    }

    func parameters(for call: String) -> [MockParameter] {
        return mocked.parameters(for: call)
    }
}
```

I don't know about you guys, but I find this really cool. Now, any mock object we create in tests can conform to this protocol and we can then use these methods to record method invocations, set and get mocked return values for tests, and get the parameters and number of invocations of a particular method.

This code here is very simple, but it will serve our purposes well. We are going to use this when testing our app in the second half of this book. If you need a mocking framework for a project, I would look into MockFive on GitHub. It offers a lot more features. Well, that's the end of our playground reference. The rest of this chapter is going to be more of an introduction/overview of XCTest and Swift Package Manager testing. Let's now discuss Apple's testing framework XCTest.

XCTest

This section is all about unit tests; there are going to be no new playground additions in this section, so sit back and relax. When you create a new Xcode project, you have the choice to include a target for unit tests. Figure 8-1 shows these options on the *Choose Options for Your New Project* screen.

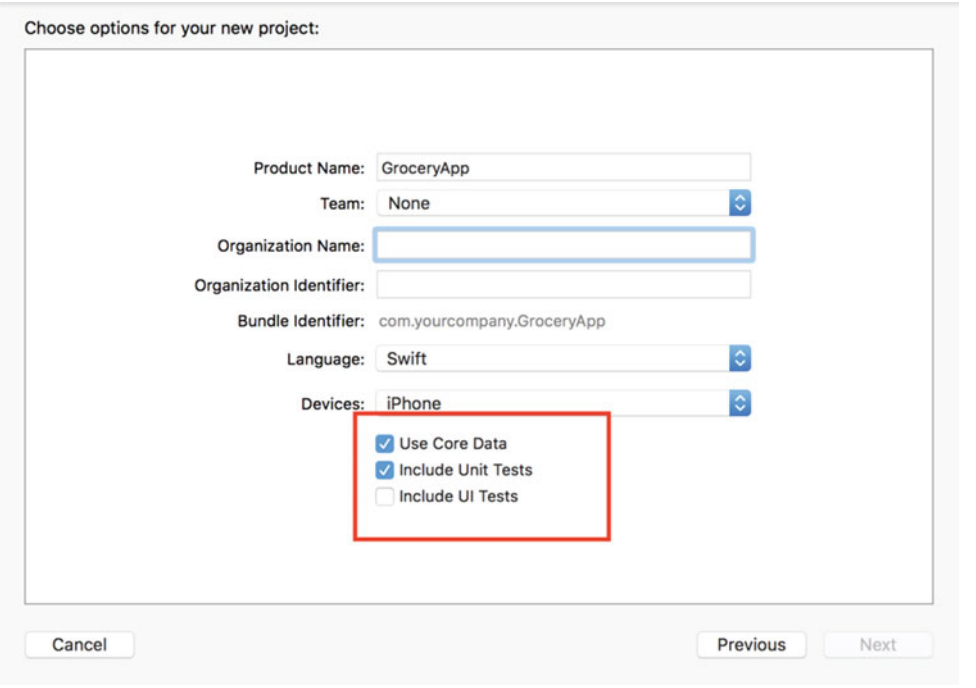


Figure 8-1. *The Choose Projects for Your Ne Project screen where you have the choice to include Unit Tests/UI Tests*

If you select the *Include Unit Tests* option, the XCTest framework is then automatically provided with a test target for the project. The following code block is what Apple provides after creating a new test file:

```
import XCTest
@testable import Chapter8

class Chapter8Tests: XCTestCase {

    override func setUp() {
        super.setUp()
        // Put setup code here. This method is called before the invocation of each test
        method in the class.
    }

    override func tearDown() {
        // Put teardown code here. This method is called after the invocation of each test
        method in the class.
        super.tearDown()
    }

    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the correct
        results.
    }

    func testPerformanceExample() {
        // This is an example of a performance test case.
        self.measure {
            // Put the code you want to measure the time of here.
        }
    }
}
```

The first thing to notice is the `@testable` attribute near the top of the file. In Chapter 3, we discussed the idea of Swift modules. Those modules can be imported into other Swift modules. We have to import our module to allow the tests to access our code. The added `@testable` attribute then gives our tests access to the internal members within our module. Internal members are any class, struct, enum, or variable that is not marked with `public` or `private` keywords. However, we still cannot access the `private` members even with the `@testable` attribute.

Our `Chapter8Tests` class then inherits from `XCTestCase`, which is the base class for all unit test classes. This class gives us a bunch of methods and functionality to help write tests. The first three methods are all pretty standard, `setUp`, `tearDown`, and an example test method. The last method is actually pretty cool. We can measure the performance of our code. This can come in really handy when working on nonfunctional requirements, such as performance and speed.

The XCTest framework uses *XCTest Assertions* to provide test results. These assertions range from equality tests to exception tests. A list of these assertions can be found at the bottom of the article at the following link: https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/04-writing_tests.html. There is not much to talk about in XCTest. There is a lot to it, but most you can learn by just writing tests. I do want to talk about *expectations* though.

Expectations in XCTest are very important and allow you to test asynchronous code without any weird dispatch async. If you write a test that actually makes a network call, whether that is good practice or not, you cannot write a normal unit test because your networking code is probably asynchronous (I hope it is!). This problem is solved by *expectations*, so let's look at how to use them. You can define an expectation with the `expectationWithDescription` method:

```
let expect = self.expectation(withDescription: "My Expectation")
```

Now, you need to set up a `waitForExpectations`. This method expects a timeout to be passed in an optional closure. This closure is where you would place your `XCTAsserts` for the test. If the `waitForExpectations` times out, it is automatically treated as a test failure:

```
self.waitForExpectations(withTimeout: 5.0) { error in
            XCTAssertEqual(1, 1)
}
```

Now, this is not enough to test our code. The closure injected here on `waitForExpectations` will not run unless our expectation object has been fulfilled. The expectation object has the `fulfill` method on it and you can use this method when an asynchronous task has completed. So, if we look at a network call, it might look something like this:

```
let network = MyNetwork()


network.makeRequest {
    expect.fulfill()
}
```

We have a networking class that performs an asynchronous network request, which is the method under test, and that method takes a closure itself. This is a fairly common pattern for this type of code. We can then `fulfill` our `expectation` in the completion block that we pass in. This is what signals `waitForExpectations`. We then must put all of our `XCTAssert` calls in the closure corresponding with the `waitForExpectations`. Let's look at the full test:

```
func testMyClassAsynchronous() {
        let expect = self.expectation(withDescription: "My Expectation")
        let subject = MyNetwork()

        subject.makeRequest {
            expect.fulfill()
        }

        self.waitForExpectations(withTimeout: 5.0) { error in
            XCTAssertEqual(1, 1)
        }
}
```

You might be wondering why we cannot assert on our values within our asynchronous method closure. We cannot do this because our tests end when they hit the last line of the method scope. After the end of the scope, no test results are recorded. Therefore, any asserts in our asynchronous closure would either not run or not be recorded. This can lead to faulty tests as they might pass, but our assertions never actually ran.

Now that we have gone through a quick introduction to testing in iOS, let's get into a more stylistic discussion. The next section is going to cover the concepts of DRY testing and WET testing. Let's jump back over to our playground and finish it with one last reference.

DRY vs. WET Testing

What are these two concepts? Well, *DRY* stands for *don't repeat yourself* and *WET* stands for *well-expressed tests*. These two concepts are the driving force behind a lot of different styles of testing. This is where we start to get into the whole idea of balance and efficiency in testing. I wanted to discuss these two concepts in this book because I believe by using Swift, we have a unique opportunity to create well-balanced tests. More on that later; let's start with the problems with WET and DRY testing.

WET Testing

Well-expressed tests can be a good thing. It is the same idea as when you leave a codebase for a couple of months and when you come back, you want to be able to understand the code. When we look at tests for our codebase, we want to be able to understand the concept behind the test and how it works. If tests are too DRY, we might not be able to fully understand what the test is trying to accomplish.

Let's explore some code from Chapter 5 where we presented a `UIAlertController`. The code I am referring to is from the playground page titled *Ch05—Protocols in UIKit.* This is going to take some effort to set up, so bear with me. The following block is the code from this page to refresh your memory:

```
protocol canPresentViewControllers {
    func present(_ viewControllerToPresent: UIViewController, animated flag: Bool,
completion: (() -> Void)?)
}

//: #### Alert Displayer using struct and protocol

struct AlertDisplayer {
    var canPresentControllers: canPresentViewControllers

    init(canPresentControllers: canPresentViewControllers) {
        self.canPresentControllers = canPresentControllers
    }

    func displayAlert(withTitle title: String?, andMessage message: String?) {
        // present UIAlertController
    }
}

extension UIViewController: canPresentViewControllers {}
```

This section is going to explore this code more and see how we would write tests for the usage of this code. To properly test this code, we are going to need to write a bit more for this alert displayer. Let's start our new playground page and add the following code:

```
import UIKit

//: # DRY vs WET Testing with AlertDisplayer

//: #### 1. Protocols from Ch05

protocol canPresentViewControllers {
    func present(_ viewControllerToPresent: UIViewController, animated flag: Bool,
    completion: (() -> Void)?)
}

//: #### 2. New Protocol for mocking purposes

protocol AlertDisplayer {
    var canPresentControllers: canPresentViewControllers { get set }
    func displayAlert(withTitle title: String?, andMessage message: String?)
}
```

All right, so the first protocol is nothing new, but in step 2, we have a new protocol that now represents the `AlertDisplayer`. In Chapter 5, we discussed traits, abilities, and interfaces. The protocol from step 1 is our trait, and we have now introduced an interface for the `AlertDisplayer`. The next step is to create a new struct that conforms to this `AlertDisplayer` protocol:

```
//: #### 3. New Struct to conform to protocol from step 2


struct ErrorAlertDisplayer: AlertDisplayer {
    var canPresentControllers: canPresentViewControllers

    init(canPresentControllers: canPresentViewControllers) {
        self.canPresentControllers = canPresentControllers
    }

    func displayAlert(withTitle title: String?, andMessage message: String?) {
        let alert = UIAlertController(title: title, message: message, preferredStyle:
.alert)
        alert.addAction( UIAlertAction(title: "OK", style: .cancel, handler: nil) )
        canPresentControllers.present(alert, animated: UIView.areAnimationsEnabled,
        completion: nil)
    }
}
```

The `ErrorAlertDisplayer` is our new struct and it looks pretty similar to the previous `AlertDisplayer` struct. The big difference here is we have now opened ourselves up for testing. Before, we just had a struct and structs cannot be subclassed. Testing the usage of

the previous struct would have been difficult because we would not have been able to mock it. We now can with this new protocol. Before we get to the tests, let's create one more protocol:

```
//: #### 4. New Protocol for composition


protocol canDisplayErrors: class {
    var alertDisplayer: AlertDisplayer { get set }
}
```

This protocol illustrates *composition* over inheritance. This means you want to compose your object of other objects to split out responsibilities instead of having one class that handles everything. This makes a system very easy to test, because you can test the usage of that object instead of the task the object accomplishes. The protocol from step 4 can be used on view controllers to give them an `AlertDisplayer` property.

This is where things get a little tricky. We are still in our playground, but tests cannot run in a playground. We want to keep this reference code in our playground, but feel free to boot up a new project and try this out for real. For the playground, I am just going to write functions as if they were in an `XCTestCase` subclass. Let's first start by creating a `subject` for our tests:

```
//: #### 5. View Controller used for testing

class MyViewController: UIViewController, canPresentViewControllers, canDisplayErrors {

    /// Lazily initialized AlertDisplayer so self can be injected
    lazy var alertDisplayer: AlertDisplayer = {
        return ErrorAlertDisplayer(canPresentControllers: self)
    }()

    override func viewDidLoad() {
        super.viewDidLoad()

        alertDisplayer.displayAlert(withTitle: "Title-1", andMessage: "Message-1")
    }
}
```

> *A **lazy stored property** is a property whose initial value is not calculated until the first time it is used. You indicate a lazy stored property by writing the lazy modifier before its declaration.*
>
> —Apple Inc.

So this code might look a little weird with the `lazy` keyword. I used a `lazy` variable so I can inject self into the `ErrorAlertDisplayer`. This could have been accomplished by overriding `init` on the view controller and initializing `alertDisplayer` there. I then `override` the `viewDidLoad` method and call `displayAlert` there.

So what would the test for this look like? Let's start with some setup. First we need a `FakeDisplayer` class:

```
//: #### 6: Fake Displayer used for testing purposes

class FakeDisplayer: AlertDisplayer {
    var title: String? = nil
    var message: String? = nil

    var canPresentControllers: canPresentViewControllers

    init(presentVC: canPresentViewControllers) {
        canPresentControllers = presentVC
    }
    func displayAlert(withTitle title: String?, andMessage message: String?) {
        self.title = title
        self.message = message
    }
}
```

There is nothing special here, so let's move on to the actual test function:

```
//: #### 7. Test function for MyViewController


func testShouldCallDisplayAlertWET() {
    let subject = MyViewController()
    let fake = FakeDisplayer(presentVC: subject)
    subject.alertDisplayer = fake

    // Action
    subject.viewDidLoad()

    print(fake.title == "Title-1")
    print(fake.message == "Message-1")
}

testShouldCallDisplayAlert()
```

Finally, we have our test. This is pretty cool and it tests our view controller subject perfectly. In a real test, the prints at the end would be `XCTAssertEquals`. Now that we have seen how we might test this sort of functionality, let's get back to the point of this section: well-expressed tests. It is clear from the previous code that we are testing the `MyViewController` class, specifically the `viewDidLoad` method.

Now, this by itself is not what this style of testing is about. Let's say we implement more on our view controller subject and we end up using the alert displayer in three or four more places. The WET style of testing would say we have to make sure each test expresses itself well. This might mean repeating the above code in all tests with different actions and assertions. You can see how this might get a bit repetitive. Then, if other view controllers adopt this protocol and need to present alerts, this gets even worse. Now, let's see how a DRY test might be written.

DRY Testing

This next part is going to require a bit of imagination. We are going to simulate an `XCTestCase` subclass for testing the `MyViewController` subject. To make our test more DRY, we are going to need a `subject` and `fakeDisplayer` that live outside of the specific test. We can then use these two variables in multiple tests. Here is the code for this style of testing:

```
//: ## DRY

//: #### 8. Set up all tests with the same subject and fake displayer

var subject: MyViewController!
var fakeDisplayer: FakeDisplayer!

func setUp() {
    subject = MyViewController()
    fakeDisplayer = FakeDisplayer(presentVC: subject)
    subject.alertDisplayer = fakeDisplayer
}

func testShouldCallDisplayAlertDRY() {
    // Action
    subject.viewDidLoad()

    print(fakeDisplayer.title == "Title-1")
    print(fakeDisplayer.message == "Message-1")
}

setUp()
testShouldCallDisplayAlertDRY()
```

The `setUp` function here mimics the `XCTestCase setUp` and then we have our test. You can see the test just calls the action and then asserts on the correct values. Now, if we have multiple cases where we have to test this functionality, we are already covered. This is a lightly version of DRY. Some might argue this is still not DRY enough, but the point here is that we should keep repetitive code out of our tests. The issue here is that we can lose context very easily. The tests are no longer self-contained and depend on setup and tear down. Depending on how many tests we have, things can get lost.

So, what is this all about? Why are we discussing this subject? Here's where I hope everything comes together. This entire book has been building toward the app we are going to build. The purpose of the first eight chapters is meant to prepare you for the techniques and concepts we are going to use in the app. I also said we are going to write tests for our app. Chapters 1 through 7 have hopefully prepared you for the production code, but this chapter is meant to prepare you for testing.

Now, if you look back on the code from this section, you can see how we have used protocol-oriented programming techniques and concepts explored in Chapter 4 with architecture and single responsibility to write very testable code. Let's now use these same ideas to write code meant for testing to achieve balanced tests instead of strictly DRY or WET tests.

Balanced Testing

This section is going to explore how we can strike a balance between the ideas of DRY and WET testing strategies. The purpose of this section is to present concepts that can accomplish the goals of both the DRY and WET strategies. We want our tests to not be dependent on external pieces, but we want very efficient and concise tests that are easy to write. Using the concepts of this book, specifically protocol-oriented programming, I believe we can achieve this in Swift in a clear and concise manner.

Let's start with a new playground page. We are going to need to copy steps 1 through 6 from the *Ch08—DRY vs. WET Testing* page into our new playground page. Once we have the necessary code copied in, we can start with a new protocol:

```
//: # Balanced Testing

//: #### 1. New Protocol meant for subclasses of XCTestCase

protocol AlertDisplayerTestBehavior {
    func assert<T: canDisplayErrors & canPresentViewControllers>(subject: T,
                displaysAlertWithTitle title: String?,
                andMessage message: String?,
                onAction action: (T) -> Void
    )
}
```

This new protocol defines a trait for tests called `AlertDisplayerTestBehavior`. The whole idea is that it describes the behavior of an `AlertDisplayer`. This protocol's method has a lot to it, so let's break it down. First, it is a generic method where the generic type is a combination of two protocols: `canDisplayErrors` and `canPresentViewControllers`.

> **Note** This & symbol describes the new form of protocol composition in Swift 3. Previously this code would have been written as `protocol<canDisplayErrors, canPresentViewControllers>`.

Then, the parameters are the `subject`, `title`, `message`, and `action`. The combination of all of these pieces is what we are trying to test for the behavior in `MyViewController`. Now, since we are creating a trait here, let's write an extension. This extension is going to handle the nitty-gritty of the test:

```
extension AlertDisplayerTestBehavior {
    func assert<T: canDisplayErrors & canPresentViewControllers>(subject: T,
                displaysAlertWithTitle title: String?,
                andMessage message: String?,
                onAction action: (T) -> Void
        ) {
```

```
        let fakeDisplayer = FakeDisplayer(presentVC: subject)
        subject.alertDisplayer = fakeDisplayer

        action(subject)

        print(fakeDisplayer.title == title)
        print(fakeDisplayer.message == message)
    }
}
```

This code looks almost identical to the code from step 7 on the *Ch08—DRY vs. WET Testing* page. We create our fake displayer and set it up on the `subject` of the test. We then call the `action` passed in with the now setup `subject`. Then we can write the assertions (or in this case, the prints) for the test. Now, let's see how we would use this in a test. We will need an actual class, so we can use our protocol:

```
//: #### 2. Simulate Test with behavior protocol

class Test: AlertDisplayerTestBehavior {

    func testShouldCallDisplayAlert() {
        let subject = MyViewController()

        assert(subject: subject, displaysAlertWithTitle: "Title-1", andMessage: "Message-1")
{
            $0.viewDidLoad()
        }
    }
}

let testInstance = Test()
testInstance.testShouldCallDisplayAlert()
```

And that's our test. This protocol has allowed us to write the test so it can express itself very well. We know what the subject is, what the action is, and the values we expect. However, the test is completely ignorant of the fake displayer or the setup of the `subject`. It does not need to know how to do this. Then, if we need to test multiple actions slightly differently, we just reuse the protocol method and all of our tests are DRY, yet well expressed.

Using this style works really well for repeated tasks. There are still plenty of cases where you would need to test one-off actions that are not repeated. Those tests do not need a behavior protocol defined with all of the setup. It would be overkill and inefficient. The takeaway for this section is that we need to examine our tests and treat them more like production code than just something we have to write with lots of repetition. If we can write our code to be clear and concise, we need to write our tests in that same manner.

This is the end of our playground reference. After this chapter, we will rely on the concepts from our playground to build our app. The next section is going to expand on Chapter 3's Logger package and this section as well. We need to refactor our package to make it more testable and we need to write balanced tests. Dust off the Logger package and let's get started.

Swift Package Manager Testing

In this section, we are going to examine our Swift package from Chapter 3 and write unit tests for it. Our Logger package from Chapter 3 is very simple and not very testable. We have just used the global `print` function in our Logger struct. To refresh your memory, here is the code we currently have in the `Logger.swift` file:

```
public enum LogFilter {
    case Info
    case Severe
}

public struct Logger {

    public var severity: LogFilter

    public init() {
        severity = .Info
    }

    public func log(item: String?, withSeverity severity: LogFilter) {
        if self.severity == severity {
            print(item)
        }
    }
}
```

Also, since writing this code, we have not looked at the tests that were generated by `swift package init`. This code actually breaks the original tests. Before, we can start writing tests, we need to abstract some of the functionality out of this `Logger` struct because there is no good way to test the `print` statement. I am also going to switch the `Logger` to a class instead of a struct. Within the `Logger.swift` file, let's create two new protocols and a new struct:

```
/// Interface to describe printing an Any type item
public protocol Print {
    func customPrint(_ item: Any)
}

/// Trait for object composition
public protocol canPrint {
    var printer: Print { get set }
}

/// Struct that conforms to the Print protocol to encapsulate global print function
public struct Printer: Print {
    public func customPrint(_ item: Any) {
        print(item)
    }
}
```

Now that we have our protocols and new struct in place, what does the `Logger` class now look like? Well, it will conform to the new `canPrint` protocol for one. Then it will have access to a type that can encapsulate the global print function, thereby making it testable:

```
public class Logger: canPrint {


    public var severity: LogFilter
    public var printer: Print

    public init(severity: LogFilter = .Info, printer: Print = Printer()) {
        self.severity = severity
        self.printer = printer
        self.printer.customPrint("-- Starting Logger with severity: \(severity)")
    }

    public func log(item: String?, withSeverity severity: LogFilter) {
        if self.severity == severity, let i = item {
            printer.customPrint("-- \(i)")
        }
    }
}
```

The new `Logger` class is still very simple. We have not changed any of the logic, we've just added a new dependency. On the note of dependencies, we are using dependency injection here to give the `Logger` class an object that conforms to the `Print` protocol as well as the `severity` level instead of just defaulting to `Info`.

We are then defaulting these parameters to be the `Printer` struct if the user of this class does not want to provide one. Similarly, we still give the `severity` a default of `Info` if one is not specified. Then, it just uses this new object and the custom print method instead of the global print function. Now that we have all of the pieces, we can start to focus on tests. The following block is what has been provided to us. We don't want to remove all of this yet:

```
import XCTest
@testable import Logger

class LoggerTests: XCTestCase {
    func testExample() {
        // This is an example of a functional test case.
        // Use XCTAssert and related functions to verify your tests produce the correct
results.
        XCTAssertEqual(Logger().text, "Hello, World!")
    }

    static var allTests : [(String, (LoggerTests) -> () throws -> Void)] {
        return [
            ("testExample", testExample),
        ]
    }
}
```

Currently, this does not compile because we removed the initial `text` property from Logger. The important part we want to keep is the `allTests` property at the bottom of this class. If we go to the Terminal app and look at the `Tests` directory, we will see a file that is not in Xcode—`LinuxMain.swift`:

```
Tests/
├── LinuxMain.swift
└── LoggerTests
    └── LoggerTests.swift
```

```
import XCTest
@testable import LoggerTests

XCTMain([
     testCase(LoggerTests.allTests),
])
```

This work is what Xcode does for us in normal iOS projects. The `allTests` property and this file are creating a manifest of all the tests that need to be run and the corresponding method references to run. Working in Xcode, we do not need to worry about this, but we want to make sure we add the appropriate references, so this can all still work from the command line and on Linux machines. The first test we are going to write is going to verify that our defaulted parameters are correct:

```
class LoggerTests: XCTestCase {


    func testShouldDefaultTheSeverityAndPrinterProperties() {
        let subject = Logger()

        XCTAssertEqual(subject.severity, LogFilter.Info)
        XCTAssertTrue(subject.printer is Printer)
    }

    static var allTests : [(String, (LoggerTests) -> () throws -> Void)] {
        return [
            (
                "testShouldDefaultTheSeverityAndPrinterProperties",
                testShouldDefaultTheSeverityAndPrinterProperties
            ),
        ]
    }
}
```

You can see that we have added our new test to the `allTests` property. Before we go any further, let's bring in the concepts from the last section on balanced tests. We are setup perfectly with our protocols. Now, it might seem like overkill right now, but if we want to add any more logging methods, this will come in handy. In the `LoggerTests.swift` file, let's create a fake printer and a new test behavior protocol. This time, we can use real `XCTAsserts` in the extension:

```
class FakePrinter: Print {
    var printedItems: [Any] = []

    func customPrint(_ item: Any) {
        printedItems.append(item)
    }
}

protocol CustomPrintTestBehavior {
    func assert<T: canPrint, U: Equatable>(subject: inout T,
                callsCustomPrintWithItem item: U,
                onAction action: (T) -> Void
    )
}
```

The `FakePrinter` class is as simple as it gets. We just save off each item in an array. Then `CustomPrintTestBehavior` has all of the items we need to test our functionality. The interesting part about this behavior is that we have two generic types here `T` and `U`. `T` is constrained to the `canPrint` protocol and `U` is constrained to the `Equatable` protocol. `Equatable` is a protocol that has been provided by Apple for comparing two objects. The parameter for the custom print method is an `Any` type and this is not `Equatable`. To use `XCTAssertEquals`, we must use an `Equatable` type.

The second important part to notice is that the subject type as been label as an `inout` parameter. In the previous section, the protocol was a class protocol. This meant it was a reference type and the internals could change with the `assert` method. However, in this example, our `canPrint` protocol is not a class protocol. Therefore, we need to use the `inout` keyword to tell the compiler we have the original type and not a copy, which would be constant. Let's now see how the extension looks:

```
extension CustomPrintTestBehavior {
    func assert<T: canPrint, U: Equatable>(subject: inout T,
                callsCustomPrintWithItem item: U,
                onAction action: (T) -> Void
    ) {
        let fakePrinter = FakePrinter()
        subject.printer = fakePrinter

        action(subject)

        XCTAssertEqual(item, fakePrinter.printedItems.last as? U)
    }
}
```

Just like the alert displayer test behavior, we create an instance of the `FakePrinter` and set the `printer` property on `subject`. Then it calls `action`, injecting the `subject`, and finally it asserts that the `item` is the last item on the `fakePrinter`. We should now be able to reuse this method anywhere we want to test that an object that conforms to `canPrint` uses the `FakePrinter`. Now, we can finally write our test using this behavior:

```
class LoggerTests: XCTestCase, CustomPrintTestBehavior {

    func testShouldDefaultTheSeverityAndPrinterProperties() {
        let subject = Logger()
```

```
        XCTAssertEqual(subject.severity, LogFilter.Info)
        XCTAssertTrue(subject.printer is Printer)
    }

    func testShouldLogTheItemBasedOnMatchingSeverity() {
        var subject = Logger()
        subject.severity = .Severe

        assert(subject: &subject, callsCustomPrintWithItem: "Hello") {
            $0.log(item: "Hello", withSeverity: .Severe)
        }
    }

    static var allTests : [(String, (LoggerTests) -> () throws -> Void)] {...}
}
```

The test class now conforms to the test behavior protocol and in the second test method, we use it to test the `log` method on `Logger`. Now, if you run the tests… they should fail. The `Logger` class prefixes every log with two dashes. I did this intentionally, so you can see that the test does not fail within the actual `XCTestCase` class. It fails in the method on our extension. This works here, but if the extension was in a different file and it was used more than once, it would be very difficult to discern which test failed.

Let's revisit our protocol and extension to fix this issue. We need to make sure the test fails in the actual test method. To accomplish this, we need two more parameters on our protocol method. Here is the updated signature:

```
protocol CustomPrintTestBehavior {
    func assert<T: canPrint, U: Equatable>(file: StaticString, line: UInt,
                subject: inout T,
                callsCustomPrintWithItem item: U,
                onAction action: (T) -> Void
    )
}
```

These two parameters are Swift debug identifiers. They represent the actual file and line that this method will be called on. Next, we need to add these two parameters to the extension with default values:

```
extension CustomPrintTestBehavior {
    func assert<T: canPrint, U: Equatable>(file: StaticString = #file, line: UInt = #line,
                subject: inout T,
                callsCustomPrintWithItem item: U,
                onAction action: (T) -> Void
    ) {
        let fakePrinter = FakePrinter()
        subject.printer = fakePrinter

        action(subject)

        XCTAssertEqual(item, fakePrinter.printedItems.last as? U, file: file, line: line)
    }
}
```

The `file` and `line` parameters have the values `#file` and `#line`, respectively. These are the actual debug identifiers and you'll notice these two parameters are now being used on the `XCTAssertEqual` call. This allows the test error to display on the line where this is called, instead of the `XCTAssertEqual` call. Now, without changing any code in the `LoggerTests` class, re-run the tests and it should fail on the correct line. Figure 8-2 shows how this should look now.

The last step is to make this test pass. To do so, we just need to change our assert to expect `-- Hello` instead of just `Hello`. The following code block has the entire `Logger.swift` file:

Figure 8-2. *The failing tests on the correct line instead of in our test behavior protocol*

```
import XCTest
@testable import Logger

class FakePrinter: Print {
    var printedItems: [Any] = []

    func customPrint(_ item: Any) {
        printedItems.append(item)
    }
}

protocol CustomPrintTestBehavior {
    func assert<T: canPrint, U: Equatable>(file: StaticString, line: UInt,
                subject: inout T,
                callsCustomPrintWithItem item: U,
                onAction action: (T) -> Void
    )
}

extension CustomPrintTestBehavior {
    func assert<T: canPrint, U: Equatable>(file: StaticString = #file, line: UInt = #line,
```

```
                subject: inout T,
                callsCustomPrintWithItem item: U,
                onAction action: (T) -> Void
    ) {
        let fakePrinter = FakePrinter()
        subject.printer = fakePrinter

        action(subject)

        XCTAssertEqual(item, fakePrinter.printedItems.last as? U, file: file, line: line)
    }
}

class LoggerTests: XCTestCase, CustomPrintTestBehavior {

    func testShouldDefaultTheSeverityAndPrinterProperties() {
        let subject = Logger()

        XCTAssertEqual(subject.severity, LogFilter.Info)
        XCTAssertTrue(subject.printer is Printer)
    }

    func testShouldLogTheItemBasedOnMatchingSeverity() {
        var subject = Logger()
        subject.severity = .Severe

        assert(subject: &subject, callsCustomPrintWithItem: "-- Hello") {
            $0.log(item: "Hello", withSeverity: .Severe)
        }
    }

    static var allTests : [(String, (LoggerTests) -> () throws -> Void)] {
        return [
            (
                "testShouldDefaultTheSeverityAndPrinterProperties",
                testShouldDefaultTheSeverityAndPrinterProperties
            ),
            (
                "testShouldLogTheItemBasedOnMatchingSeverity",
                testShouldLogTheItemBasedOnMatchingSeverity
            )
        ]
    }
}
```

And that is it. We have successfully tested our Logger package. If you were to turn on Xcode's code coverage, you would see that we have successfully covered our `Logger` class. We just wrote a lot of code. Hopefully, it reinforced the idea that was presented in the balanced testing section, but I also hope you have learned a little more about the Swift Package Manager and how to write tests using it. I think the last piece left to do is to run the tests via the command line. When you use the following command:

```
$ swift test
```

you will get the following output:

```
Compile Swift Module 'LoggerTests' (1 sources)
Linking ./.build/debug/LoggerPackageTests.xctest/Contents/MacOS/LoggerPackageTests
Test Suite 'All tests' started at 2016-09-25 15:53:36.589
Test Suite 'LoggerPackageTests.xctest' started at 2016-09-25 15:53:36.590
Test Suite 'LoggerTests' started at 2016-09-25 15:53:36.590
Test Case '-[LoggerTests.LoggerTests testShouldDefaultTheSeverityAndPrinterProperties]'
started.
-- Starting Logger with severity: Info
Test Case '-[LoggerTests.LoggerTests testShouldDefaultTheSeverityAndPrinterProperties]'
passed (0.001 seconds).
Test Case '-[LoggerTests.LoggerTests testShouldLogTheItemBasedOnMatchingSeverity]' started.
-- Starting Logger with severity: Info
Test Case '-[LoggerTests.LoggerTests testShouldLogTheItemBasedOnMatchingSeverity]' passed
(0.000 seconds).
Test Suite 'LoggerTests' passed at 2016-09-25 15:53:36.591.
        Executed 2 tests, with 0 failures (0 unexpected) in 0.002 (0.002) seconds
Test Suite 'LoggerPackageTests.xctest' passed at 2016-09-25 15:53:36.592.
        Executed 2 tests, with 0 failures (0 unexpected) in 0.002 (0.002) seconds
Test Suite 'All tests' passed at 2016-09-25 15:53:36.592.
        Executed 2 tests, with 0 failures (0 unexpected) in 0.002 (0.003) seconds
```

This is the end of our discussion on the Swift Package Manager. My hope is this tool is the start of Swift code being developed and run anywhere. The next and last section will discuss TDD in Swift/iOS and this whole idea of balance and efficiency.

Test-Driven Development

Let's bring this chapter in for a landing. I have shown you multiple techniques for testing. We have used concepts from throughout this book to create what I have called a balanced testing strategy through the use of protocols. I stand by everything I have shown you, but here is the tricky part. Every time we created a new protocol, we were abstracting out some concept.

Whether it was the composition of the object like the `canPrint` and `canDisplayErrors` protocols, it was complexity that was not necessary to complete the production code. That is not to say it might not come in handy later. However, as of right now, you can argue it is a bit of overkill. That's the complexity that comes with testing. So, how does all of this relate to TDD?

Using *test-driven development*, you are supposed to build up tests that create a specification of your code. This means the tests are built first, and each describes something the object does, eventually leading to the object you want. TDD can be a good thing and it can be a good place to start, if you do not know how you production code is going to end up.

However, the downside to this process is efficiency. In Chapter 1, we discussed the new API design guidelines. The driving force was clarity and conciseness. We want our code to be as clear as it can be so when someone else starts to modify it, they will know exactly what is going on. The more unnecessary abstractions we create, the more complexity we add. As we add more complexity, we in turn make our code unclear. We then lose efficiency. One big domino effect. This is where balance comes in again. Regardless of the testing strategy, the key is to make sure we are not over-simplifying or over-engineering. This is a very tough problem, which is why there are so many different schools of thought on the subject matter.

So, how does this affect us? Well, in Swift, due to the limitations we discussed in the section on mocking, we have to be very careful. In the second half of the book, while building our app, we have to make sure our code is as clear as it can be. We want to add tests, but we cannot lose sight of the bigger picture. When we build our app, there will be code that is tested. There will also be code where we do not write any tests And we are going to flip between writing tests first and writing tests after the code. We will not be using strict TDD practices, but a mixture of different testing strategies.

Wrap Up

That's a wrap for Chapter 8 and the first half of this book. Well, maybe a little more than half. We have covered a lot of topics in this chapter and somehow, it's all related to testing. We first looked at mocking in Swift. The open-source framework MockFive on GitHub influenced the mocking framework we built. This code is going to greatly simplify our mocks in the tests for our app.

Afterward, we went through an introduction to XCTest. We have only scratched the surface of Apple's testing framework here, but hopefully this has been enough to get you started. This took us directly into our discussion about DRY vs. WET testing. Despite the differences, the central goal of these two styles is properly testing our code and I believe this is accomplished through the use of both strategies.

We then finished our discussion of the official Swift Package Manager that was started in Chapter 3. We pulled in concepts from Chapters 4, 5, and 6 to write our tests for the Logger package. Putting everything together, we were able to write clear, yet expressive tests, properly validating our Logger code. And finally, we briefly discussed test-driven development in Swift. We talked about the ideas of abstraction and complexity and how they can make code testable, but also unclear. This is where we have to bring in all of our concepts to make sure we are not creating unnecessary complexity and abstractions without finding a balance to our tests and production code.

I hope you have found this chapter helpful and that you use the discussions started here to figure out your strategy for writing and testing Swift code. This is the end of the first half of book, our playground reference, and the more conceptual chapters. Starting in Chapter 9, we are going to build our Grocery List app!

Articles

1. Swift Package Manager
   - https://swift.org/package-manager/
2. Swift Evolution Proposal—SE-0019
   - https://github.com/apple/swift-evolution/blob/master/proposals/0019-package-manager-testing.md
3. Example Package Playing Card
   - https://github.com/apple/example-package-playingcard
4. About Testing with Xcode
   - https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/01-introduction.html
5. MockFive
   - https://github.com/DeliciousRaspberryPi/MockFive
6. Swift: The Only Modern Language Without Mocking Frameworks
   - http://blog.pragmaticengineer.com/swift-the-only-modern-language-with-no-mocking-framework/
7. To Mock or Not to Mock
   - http://clean-swift.com/to-mock-or-not-to-mock/

Part II

Building the Grocery App

Chapter 9

Grocery List App Interface Builder

You made it! Welcome to the second part of the book. Chapters 9–12 are going to walk you through building a Grocery List app. It might not sound very exciting, but it will be. We are going to use the knowledge we gained throughout the first section about API design, design patterns, protocol-oriented programming, and testing to build a clean and concise app. This chapter, along with Chapters 10–12, will cover the design for the app or MVP (minimal viable product).

This is where the playground reference will be very important. We have explored concepts and written code that we can use in our app. We are going to use protocols heavily and rely on our protocol-oriented programming skills to build some of the core functionality for the app. I believe protocol-oriented programming is the future of Swift development, so let's get with the times.

Design patterns from Chapter 4 will also be stars in this app. We talked about the normal iOS patterns like MVC and singleton, but we are going to try the MVVM pattern for this app. This will present some unique opportunities to keep core data away from our view controllers. Removing the idea of singletons will also force us to think about how data flows through our app.

And finally, as explained in Chapter 8, we are going to write unit tests for our code. This is where protocols and mocking will be used. TDD (test-driven development) was also discussed in Chapter 8 and we are going to use a form of TDD while building our app. However, I want us to try to keep our testing balanced, so if there is an opportunity to refactor and clean up tests, we are going to take it.

Before we get to any code though, there is Chapter 9. This chapter is going to focus on the interface for our grocery app. We are going to build the entire interface in this chapter; however, there will be pieces that will need to be finished later. We are not going to create any view controllers in code so we can hook up our `@IBOutlets` and `@IBActions` in future chapters.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_9

What You'll Learn

Even though this chapter will focus on building the interface for our app, I am also going to explain concepts in Interface Builder like segues, presentation styles, and other interface-related subject matter. If you have built iOS apps using storyboards before, this chapter might be somewhat of a review for you. If this is the case, follow along, let's get this interface built and then we can move onto the code in the next chapter.

Otherwise, if you are more familiar with Nib files or are just new to iOS in general, this should be a good chapter. In fact, we are going to start off by talking about how to transition from Nib files to storyboards. I know Nib files are getting a bit old, but they are still relevant, even in today's iOS development. First, let's start with setting up our project.

Project Setup

Before we forge ahead, let's discuss the purpose of our app. This app is going to hold onto our grocery lists. When you go to the grocery store, you might have a list or several lists, and we need to know what items to actually buy. Now, this app is not going to be flashy or have many features. That is kind of the point. I want to present an MVP implementation of this app and let you expand on it.

This app is a simple CRUD app, minus the UD. There are other features that are missing as well; for instance, it does not allow the user to check an item off the list. So now you might be wondering what the app is going to do. Well, not much, but the architecture of our app is going to be clean. We are going to write unit tests. We are going to build the base so this app can expand and scale with new features. That is our goal. We need to lay out a solid foundation.

Now, first things first, we need a new Xcode project. (I am using Xcode 8 with Swift 3.0.) Let's open up Xcode and start a new project. Figure 9-1 shows the new project screen in Xcode. Be sure to select the *Single View Application* template. I usually also start with the single view, and leave the other templates alone.

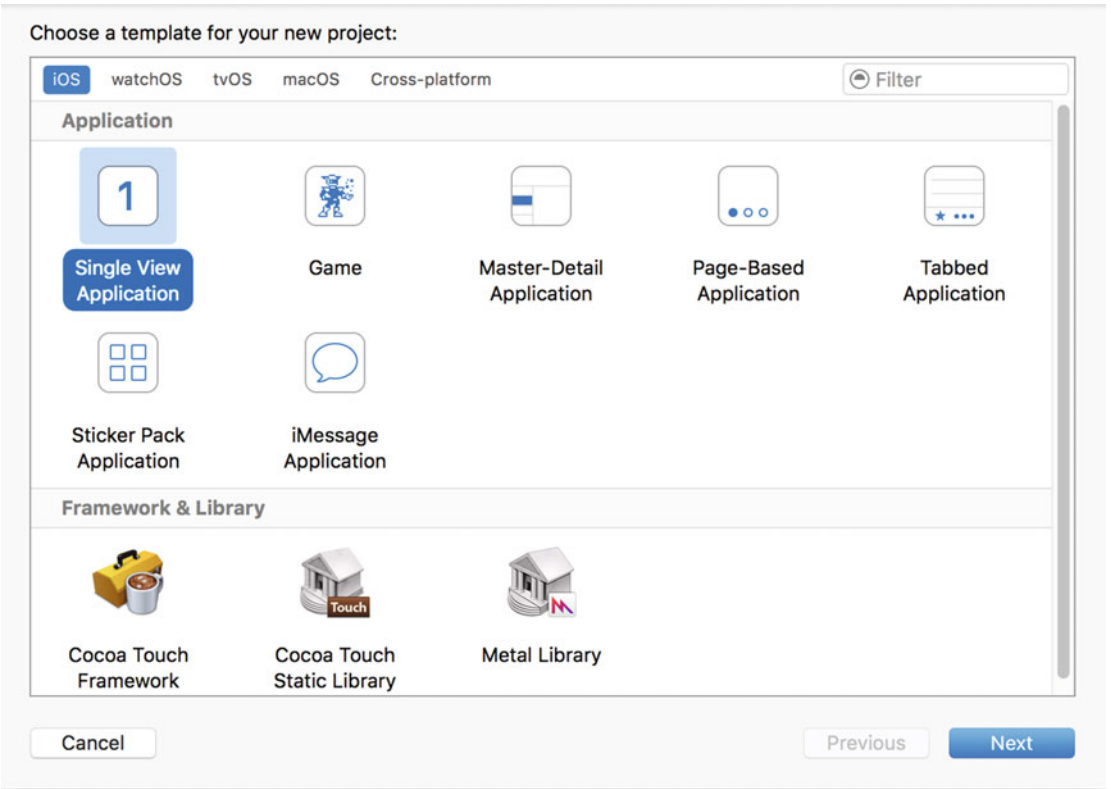


Figure 9-1. *The new project template screen with the Single View Application template selected*

Then you will be taken to the project configuration screen in Figure 9-2. You do not need to name your app GroceryApp, but it would make things easier. If you choose to name it something else, just make sure you use that name when importing the module into tests.

We also want to make sure we have the *Use Core Data* and *Include Unit Tests* options turned on, as can be seen in Figure 9-2. By default, Apple gives us code for accessing core data when we turn this option on. We do not want to have to mess with this code ourselves.

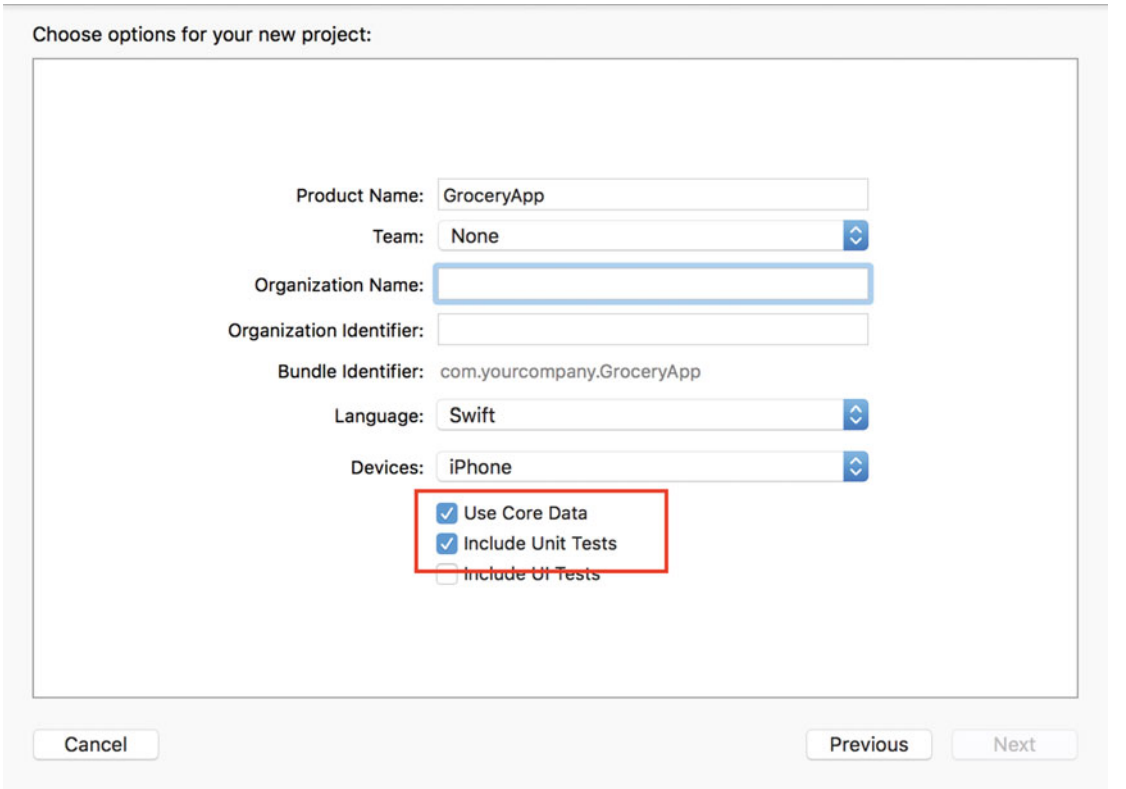


Figure 9-2. *The project configuration screen with Use Core Data and Include Unit Tests turned on*

I have left my team and organization identifier out in Figure 9-2, but you probably want to configure your project so you can install it on a device. Now, we should have a blank project all set up and ready to go.

Before we start coding or building our interface, I wanted to talk about the formatting for this and future chapters. In the previous chapters, where we added code to our playground reference, I included the playground page name(s) for the section. The playground page name was also included at the top of the formatted code block.

Since we are in an actual Xcode project now, I am not going to include any file name at the beginning of any section. I am still going to include the file name at the top of the code block, but our files are going to have much more code than our playground. Swift's comment `MARK:` syntax will be used to differentiate sections of code. The code within the `MARK:` section that is not relevant to the current task will be omitted. The following code block describes this new style.

```
class MyViewController: UIViewController {
    // MARK: - Properties

    //...

    // MARK: - Lifecycle
    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

In the preceding example, all of the properties have been omitted because the `Lifecycle` section is the focus. And as always, any code that is just for reference will not have a file name at the top of the code block. All right, we have our project set up. Let's talk a little about Nib files and storyboards.

Nib Files and Storyboards

As, I stated earlier, this section might be more of a review for you if you are used to working in storyboard files. You do not have to follow along with this section. This is just an overview.

In the olden days of iOS development, *Nib* files were used to build interfaces and storyboard files did not exist. You can still see this today in one of the initializers for `UIViewController`. There is an init with a Nib name on `UIViewController` to instantiate a view controller from a particular Nib file. Every time you wanted to transition to another view controller, you had to manually instantiate the view controller and then present it. Using storyboards, this is no longer necessary because of the *segues*.

Segues

One of the largest limitations of using Nib files is segues. They do not exist in Nib files because these files describe singular views and view controllers instead of the multiple view controllers like storyboards. *Segues* describe how a view controller transitions to the next view controller and through what action. Figure 9-3 shows a segue in a storyboard between two view controllers. The first view controller has a button on it called *Next*. When this button is tapped, it will transition to the next view controller.

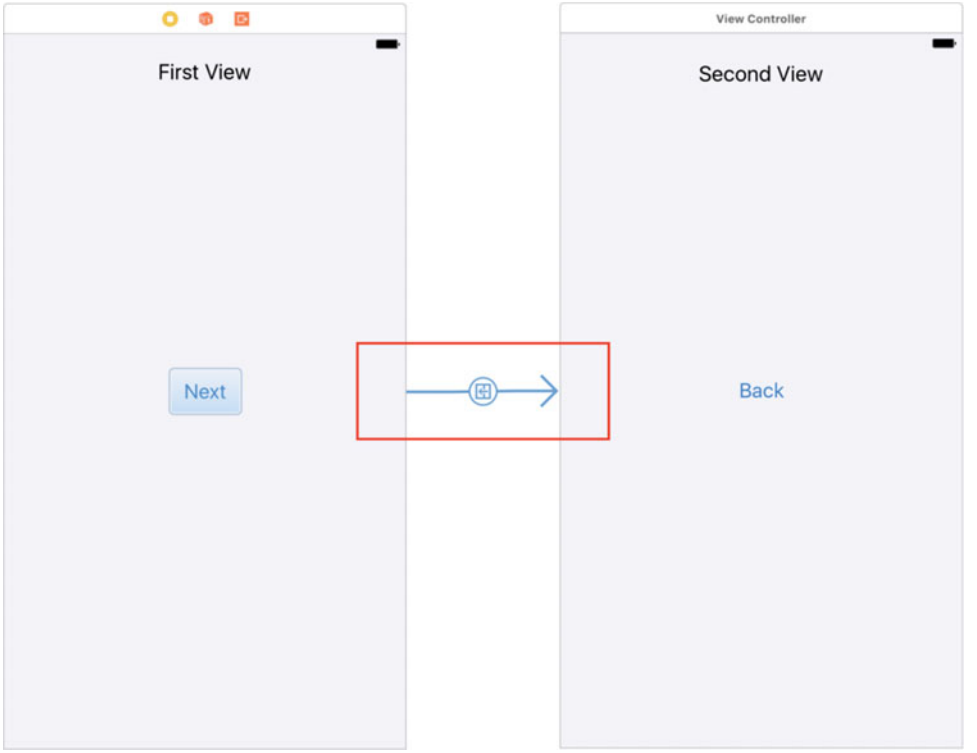


Figure 9-3. *Two view controllers that are linked via a segue from the Next button*

To create this segue, you want to Ctrl-drag from the *Next* button to the other view controller. This will bring up a menu that allows you to select the type of segue you want to use. This can be seen in Figure 9-4.

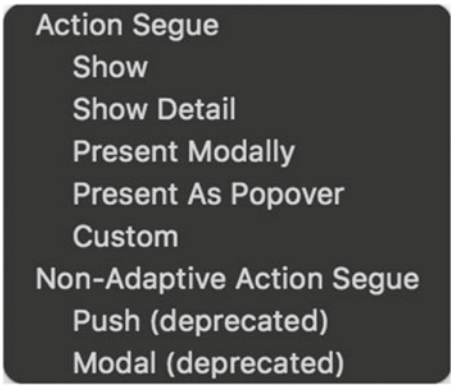


Figure 9-4. *The menu used to select a segue type*

This menu allows you to choose the basic transition between two view controllers. I do not want to discuss the details of these options here. I would rather talk about them within the context of our app, so we can save this discussion for later. The advantage of using segues is we no longer have to worry about writing code to transition between two of our view controllers. This is really powerful and can be helpful when you are rapid prototyping. Without writing any code, you can lay out an entire app.

When to Use Nib Files

Even though we have storyboards and they can alleviate a lot of redundant coding, we still want to use Nib files Say we have a complex view to build, but we want it to be highly reusable. We do not want to put this on our storyboard because it needs to be duplicated anywhere we need it. This is a perfect candidate for a Nib file.

We can then manually use this Nib file in our code when we need to. Nib files have the flexibility to describe entire view controllers or just plain old `UIViews`. This might seem like a small use case, but it is important when we have a large app and we want to abstract out view layers without going as far as storyboards.

Sometimes, it might not even be possible to get the full power of storyboards for a highly reused view. If we want a view whose purpose is to interrupt the current flow and it can happen on any view, we do not want to have to hook up segues. That would create a storyboard with segues that look tangled together. Using a Nib file in this instance can save us time and make things easier on us.

Storyboard Limitations

The caveat to using storyboards is when we start to put everything in one storyboard file. In Chapter 7, we discussed the new Refactor to Storyboard feature. Splitting out view controllers to their own storyboards can be a huge benefit, especially when you are working with a team of people who are all responsible for editing the app's interface. Now, with this new feature, we do not have to bridge the gap between storyboards with code. Previously, if we had defined a separate storyboard, it would require code to load the initial view controller. Now, segues can cross storyboard boundaries, which can alleviate this pain, but it is still possible to step on others' toes with multiple people working on the same storyboard.

With the new refactor feature, we can group features within each of our storyboards. If there is a login process, that could be one storyboard, while another might be the profile section for a user. The next limitation might not be seen as such, depending on your viewpoint. Segues offer a lot of power since we do not need to handle any transitions or flow via code. However, taking on this responsibility, we lose insight into where the current view controller is transitioning.

There is a method on `UIViewControllers` called `prepareForSegue`. This method allows us to reference the next view controller through the `segue` object. So, instead of creating the new view controller by hand, we have a catch all method for any new view controller. This might not seem like a limitation, but it changes how we pass data through the view controllers of our app. We will see this in the next chapter when we build our app.

I hope this quick section gave you some insight into the two worlds of building UI in iOS. If you have never used one of these features, either Nib files or storyboards, I highly recommend you check both out. Using one or the other might not be right for your project. The more complex apps get, the more these two concepts have to live in harmony. All of this being said, we are going to use storyboards exclusively in our app. Our app is very simple and we can reasonable fit it on one storyboard. If you chose to expand on this app in the future, you should re-evaluate then. All right, it is finally time to start building the Grocery App!

Grocery Lists and Items

This section is going to cover setting up two screens on our storyboard. These two screens will be responsible for displaying the table of grocery lists and the table of grocery items contained within specific lists. Figure 9-5 shows the result of the two screens we are going to build here. The first screen is for our grocery lists and the second screen is for the items.

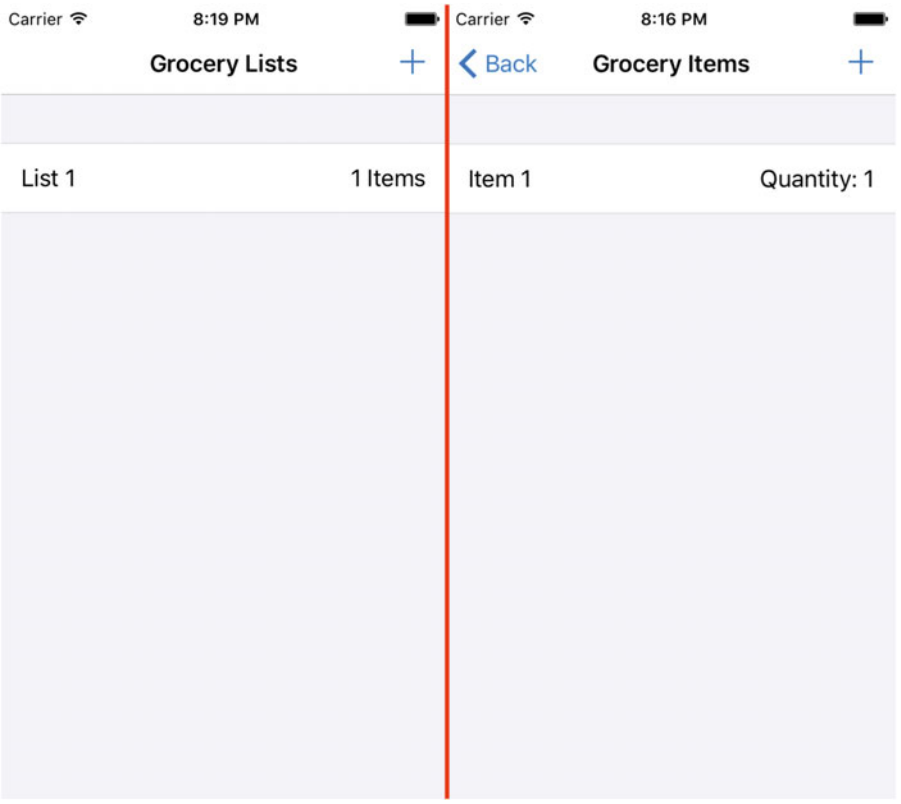


Figure 9-5. *The final result of the two screens we are going to build in this section*

Figure 9-5 shows an actual grocery list and grocery item in our table, but this chapter will just be over the interface. Adding lists and items will come later, but as you can see from Figure 9-5, we are going to have a navigation controller, a table view, and some buttons and titles. Let's go to our storyboard file and get started.

Grocery Lists

When you open your storyboard, you are going to see the blank initial view controller. Figure 9-6 shows our storyboard with the new device configuration panel open. In the figures for this chapter, I will be using the iPhone 4s portrait device configuration for better figures. You can design your interface in any configuration you would like.

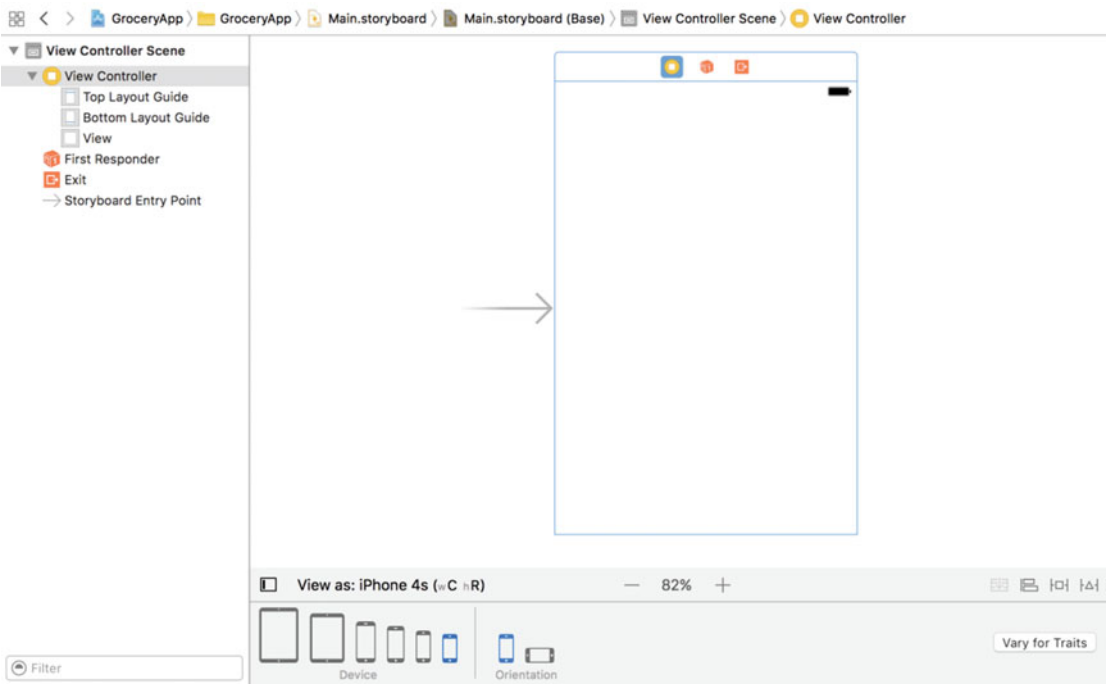


Figure 9-6. *Our storyboard file with the blank initial view controller and the new device configuration panel*

We know we are going to need a navigation controller from Figure 9-5. So, let's add this first. Figure 9-7 shows the menu option for embedding our view controller in a navigation controller. Make sure you have the initial view controller selected and then go to *Editor* ➤ *Embed In* ➤ *Navigation Controller*.

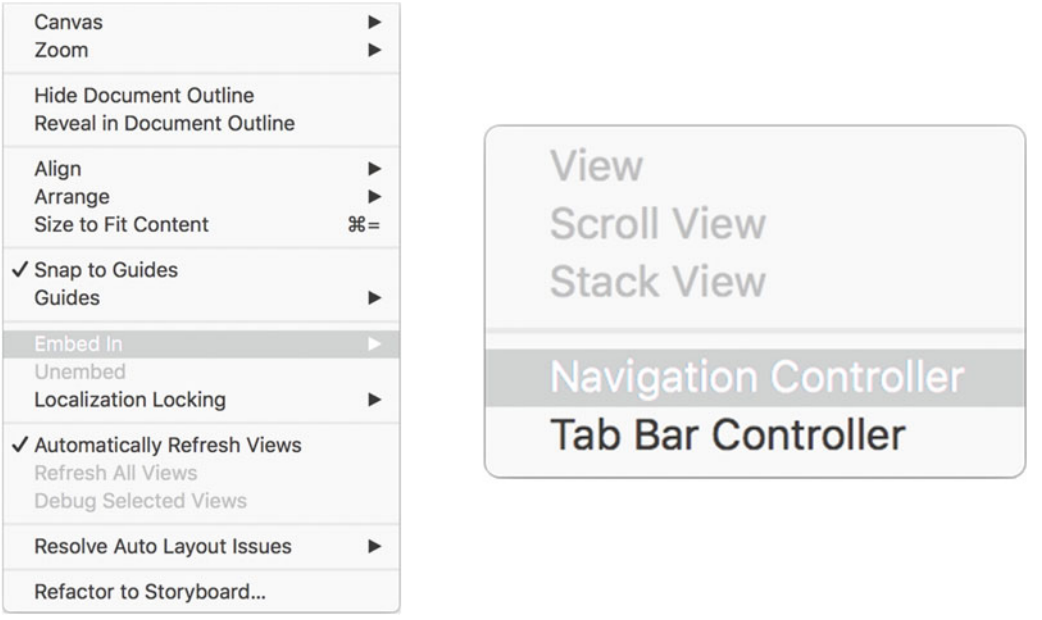


Figure 9-7. *The Editor menu and the Embed In ➤ Navigation Controller option. This will automatically link the selected view controller with a navigation controller*

Embedding the initial view controller in a navigation controller is going to affect our entire app. `UINavigationControllers` do exactly what their names suggest: control navigation. A navigation controller is what gives apps the bar at the top of the screen. That is the `UINavigationBar`. The way this works is through the use of containers. This is a relatively new feature and it allows view controllers to automatically embed another view controller.

If this seems like a weird concept, think of it like this: the navigation controller is itself a `UIViewController`. It contains any view controller that is given to it and has a view of its own that contains a `UINavigationBar`. It then controls the transitions from the root view controller to all others keeping a stack of the view controllers it visits. This stack then allows it to go back all the way to the root view controller.

What this means for our app is that any view controller we present will have a couple of choices, but let's keep going before we talk about these options. Once the initial view controller is embedded, let's add a table view. If you navigate to the Utility Pane ➤ Object Library, we can add our table view. Figure 9-8 shows the *Table View* in the Object Library. We want to be sure to avoid the table view controller.

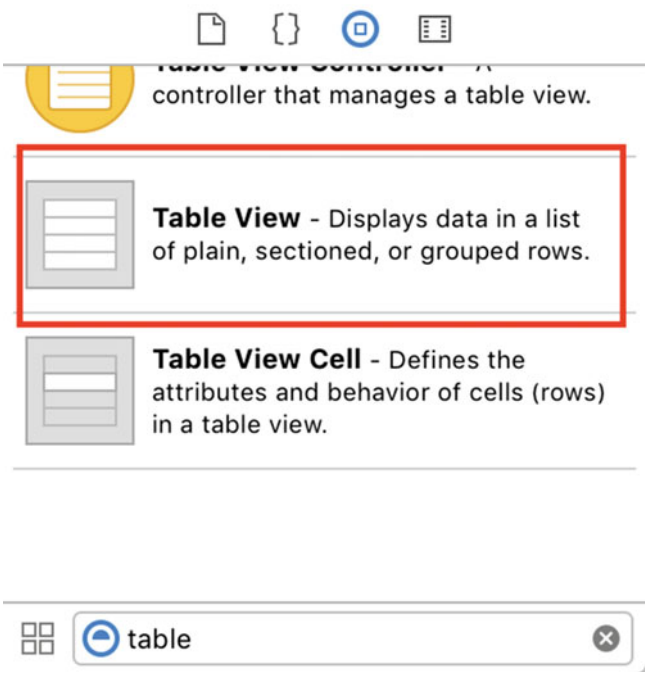


Figure 9-8. A table view in the Object Library in Xcode. Make sure you do not add the table view controller

Before we move on, let's examine the navigation bar and the interaction with the table view. Depending on your setup, you might notice the table view you just added looks a bit funny. By default, the navigation controller's navigation bar is set to translucent. This means the table view is expecting to be underneath the navigation bar and is compensating for the height difference. This might be what you want and it is what gives the table view the effect of content scrolling underneath the navigation bar. For our app, I am going to turn this option off. Now the top of the table view can be aligned to the bottom of the navigation bar.

Once we have our table view aligned correctly, let's add some constraints or, in fact, let's not. Remember way back in Chapter 2 when we discussed the new features in Xcode? We can use the old *Autoresizing Masks*. Our table view should just follow the size of the view, so this is a perfect candidate for using autoresizing instead of constraints. Figure 9-9 shows the settings to allow autoresizing to handle the behavior of our table view.

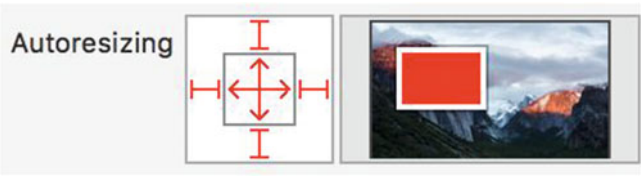


Figure 9-9. The autoresizing settings for our table view. These settings replace any constraints we would have added to the table view and are far simpler

We want to turn on the springs and stick to edges options. This will force our table view to expand and contract with the view and stick to the right, left, bottom, and top. Once you configure these settings, change the device configuration panel to see how the table view reacts. It should stick to the sides of the view and resize appropriately. This is pretty cool since we have zero constraints on our table view. Let's take a break for a second and look at what we have so far. Figure 9-10 shows our current progress.

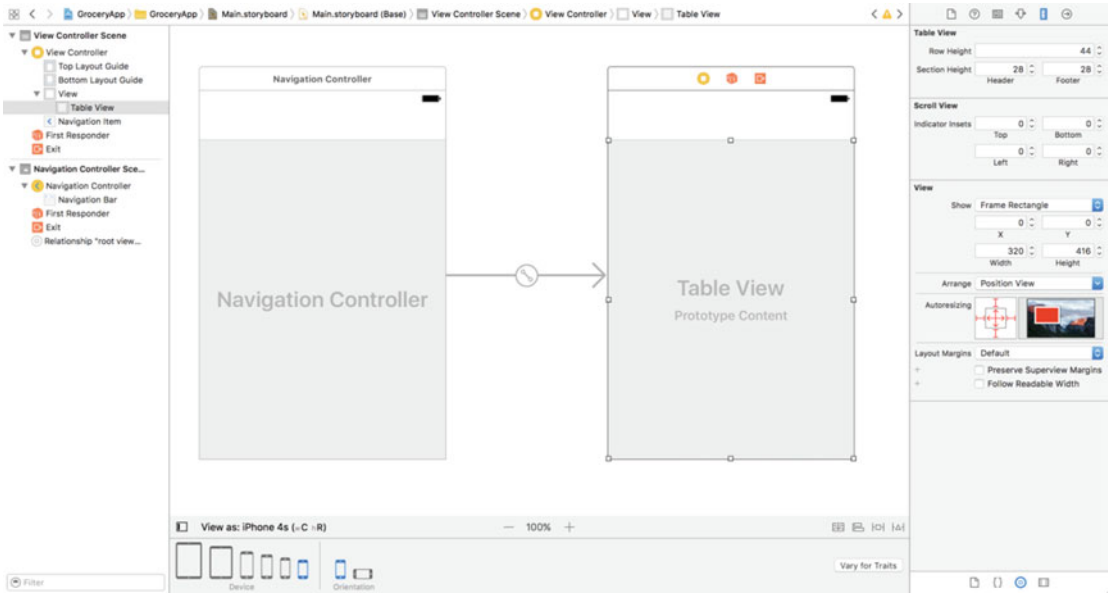


Figure 9-10. *The interface so far. 1) We have embedded our view controller in a navigation controller. 2) We have added a table view to the view controller. 3) We have set up the autoresizing mask for the table view*

All right, we have one last piece to add before we can move on to the second view controller. We need to add the title and the + button that were on our navigation bar in Figure 9-5. If you reopen the Object Library and search for navigation item, drag the new item to our view controller. It does not matter where you drop the navigation item. It should appear on top of the navigation bar with a default title.

The navigation item goes hand in hand with the navigation bar. It has a title and two areas on either side for buttons and other content. This is where you will usually see actions that can take place on the current view. In our case, this action is going to be adding more lists and items. This is the navigation item for the grocery list view controller, so let's configure this item.

If you open up the Attributes Inspector there should be three options: *Title*, *Prompt*, and *Back Button*. We need to configure the title to be *Grocery Lists* and the back button to be *Back*. The *Back Button* setting is actually the text for the back button that shows on the view controller directly after this one. In a navigation controller, the back button appears on the left hand side of the following view controller. The title setting is meant to be the main title, but the purpose of the prompt feature is to give the user additional information. We do not need this, so let's just ignore it. Figure 9-11 describes the settings for our navigation item on this view controller.

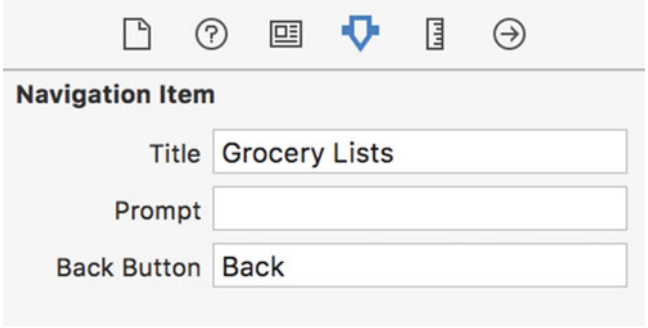


Figure 9-11. The Attributes Inspector for our new navigation item with the appropriate configurations

The last step is to add our + (plus) button. In the Object Library, let's find a `UIBarButtonItem` and drag this to the right side of our navigation item. It might seem a little weird, but navigation items cannot contain `UIButtons`. Instead, we need to use the aforementioned `UIBarButtonItem`. We then want to configure the *System Item* to be *Add*. This will give us iOS's built-in plus symbol. This saves us from using an image or just typing in +. The configuration for our bar button item can be seen in Figure 9-12.

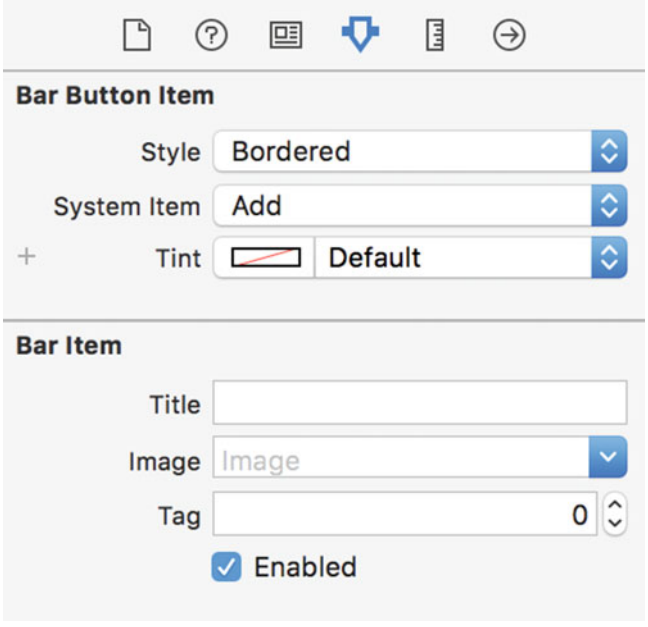


Figure 9-12. The configuration in the Attributes Inspector for the UIBarButtonItem. We want the System Item attribute to be Add

Now, we should have our first view controller properly configured. Let's recap: first, we embedded our initial view controller in a navigation controller. We then added a table view to the view and made sure to turn off the translucency of the navigation bar. Without adding constraints, we made sure our table view resized and stuck to the edges all through

autoresizing. Shout out to Chapter Two! Then finally, we added and configured a navigation item. This navigation item has the title Grocery Lists and a + button on the right-hand side. Figure 9-13 shows our storyboard with all of these features.

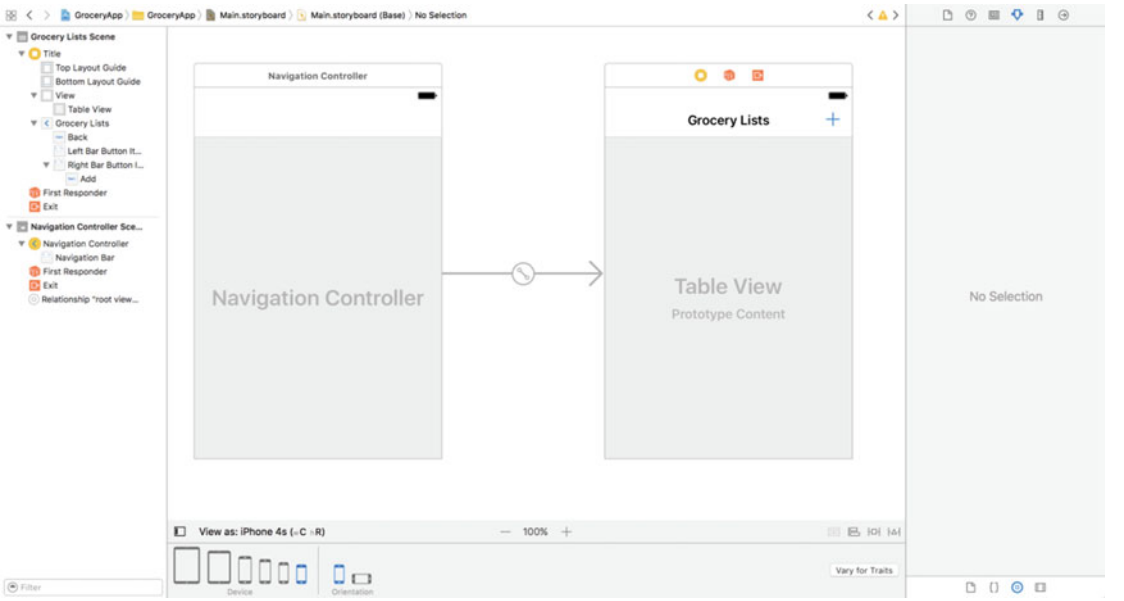


Figure 9-13. *Our storyboard with the first view controller of our app. The view controller is embedded in a navigation controller; it has a table view, and a navigation item*

Whew! It doesn't seem like that much when you are building these things, but there were a lot of steps there. We have our first view controller configured, so this next part should be easier. We are going to create our next view controller. This view controller is meant to hold the items in the selected grocery list. You can see on the right side of Figure 9-5, it is almost identical to the previous view controller.

Grocery Items

Let's start in the Object Library. Let's drag in a new view controller to our storyboard directly to the right of our previous one. Just like our previous view controller, let's add a table view. Now, we do not have a navigation bar yet, so for now, let's make sure our table view spans the entire view of the new view controller. Figure 9-14 shows the new view controller with a table view added.

Figure 9-14. *The second view controller with a table view. As of right now, this view controller is not linked with any other view controller*

Now, if you have already added the autoresizing behavior to the table view, that was the next step, so don't steal my thunder.☺ The correct autoresizing settings can be seen in Figure 9-9. Once you verify your table view can resize and adapt correctly, we need to link everything together. Let's hop back over to the original table view, because there are a couple of things left to do. First, if you were looking closely on Figure 9-3 where I showed our two view controllers side by side, you will notice both table views are configured with a grouped style instead of a plain one. Let's tackle this first.

On the Grocery List view controller, select the table view, go to the Attribute Inspector and change the *Style* to *Grouped* instead of *Plain*. Figure 9-15 shows the settings for our table view. The top two settings in Figure 9-15 are for the table view cells. The *Dynamic Prototypes* setting is meant to control whether the content for the table is dynamic or static.

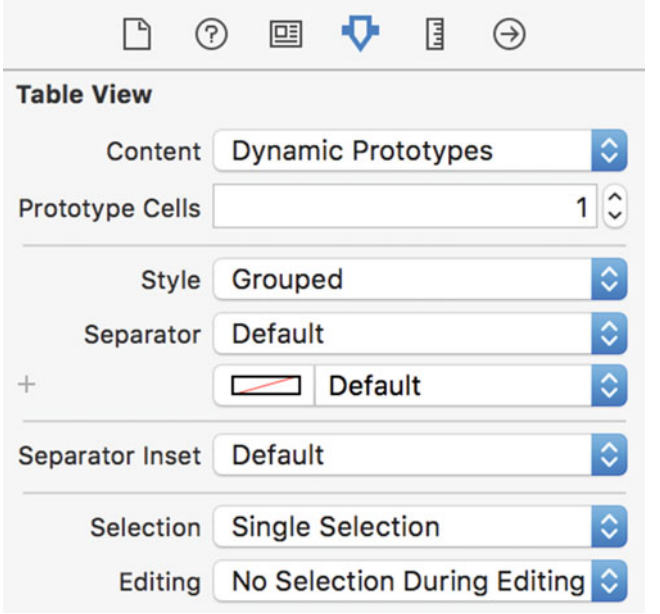


Figure 9-15. *The Attributes Inspector for the first view controller's table view configured with a Grouped style*

The other option is *Static Cells*. With *Static Cells*, you configure a table view to have as many cells as you want, and then you can give each cell labels and other content. With dynamic prototypes, the cells will be reloaded with the delegate/datasource methods on your view controller, but this is not the case for static cells. Static cells will not reload because they are well, static. Then can be an easy way to create a settings page like in many apps. You can get the nice look of a table view without the headache of dynamically generating your content.

> **Caution** The only caveat to this approach is *UITableViewController* subclasses are the only types of view controllers that can use static cells. We have only added a table view to our view controller in our storyboard.

Now, this next part is going to require a table view cell on our table view. If you saw it in the Object Library, find it again and let's add one to the table view we just configured. Figure 9-16 shows our table view with a group style and a table view cell added.

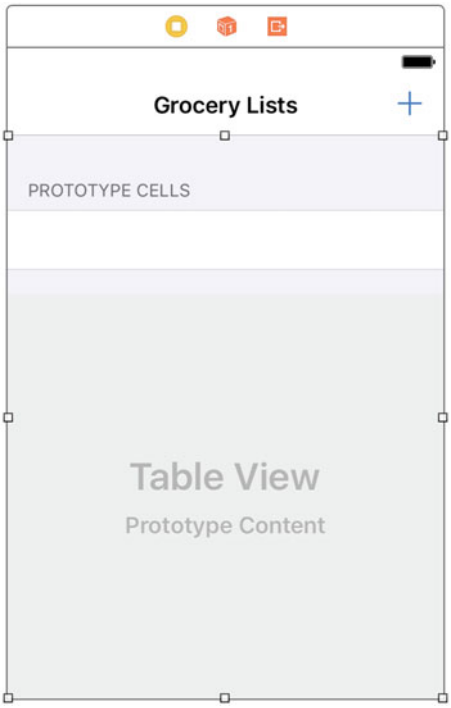


Figure 9-16. *The configured table view with a table view cell*

For this app, we are not going to create any custom table view cells. We are going to use the stock *Right Detail* style cell. The *Style* setting is in the Attributes Inspector directly at the top. There are two settings we need to configure for this cell. The first setting is the reuse identifier. The *reuse identifier* is how we can identify table cells in code through the use of the table view method: `dequeueReusableCellWithIdentifier`. We want to set the reuse identifier (called *Identifier* in Figure 9.17) for our cell to be *Cell*.

The second setting we want to configure is the cell's *Accessory*. We want our *Accessory* to be a *Disclosure Indicator*. This will put a little gray arrow on the right-hand side of the table cell. This also indicates to users that there is more content available if you tap on this cell. Figure 9-17 shows the settings for our table view cell.

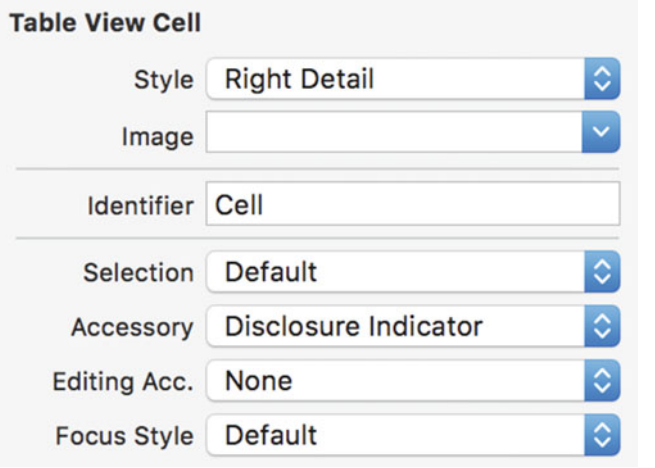


Figure 9-17. *The settings for the table view cell on the Grocery Lists view controller. It is configured to be a Right Detail cell with a reuse identifier of Cell*

The first view controller in our storyboard, the Grocery Lists view controller, should be completely configured now. The only remaining work is on the second view controller. Now, before we go configuring the second view controller, let's create a segue between the Grocery Lists view controller and the second view controller. To create this segue, we want to Ctrl-drag from the first table view cell to our second view controller and select the *Show* option on the popup menu. Finally, this brings us to the discussion I put off near the beginning of this chapter. In Figure 9-4, which I've placed again here so you do not have to flip all the way back to it, we have the segue type menu.

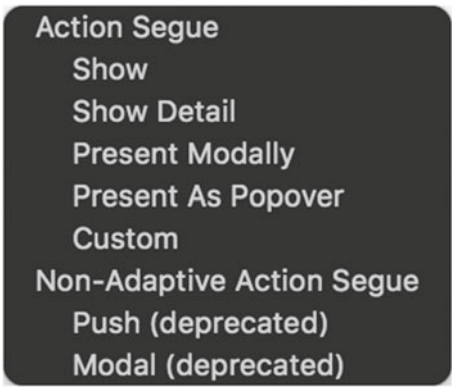


Figure 9-4. *The menu used to select a segue type*

Let's take a break from our app for a second and talk about the options in this menu. Selecting the *Show* option when creating the segue will use the current context to present the linked view controller. Since our Grocery List view controller is in a navigation controller, it will use a navigation push animation to present the second view controller.

In other contexts, the show option might present a view controller in a *Modal* presentation style. A *modal* is where the view controller presents on top of the presenting view controller. This is useful where a view does not necessarily fit into a flow, but is more of an interruption. You will also notice in Figure 9-4 there is a *Present Modally* option. This option forces the presentation to be a modal style even if there is a navigation controller in play.

The remaining options are *Show Detail*, *Present as Popover*, and *Custom*. We are not going to use these options in our app, but a full description of their purpose can be found in Apple's documentation at `https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/UsingSegues.html`.

So, for the previous task, I said to use the *Show* option. This means based on our navigation controller context, when the user taps on the table cell, the second view controller will be presented via the navigation controller. Figure 9-18 shows our progress with the segue highlighted. You will notice the table cell is also highlighted. This means we have correctly configured our segue on the tap action for the table cell.

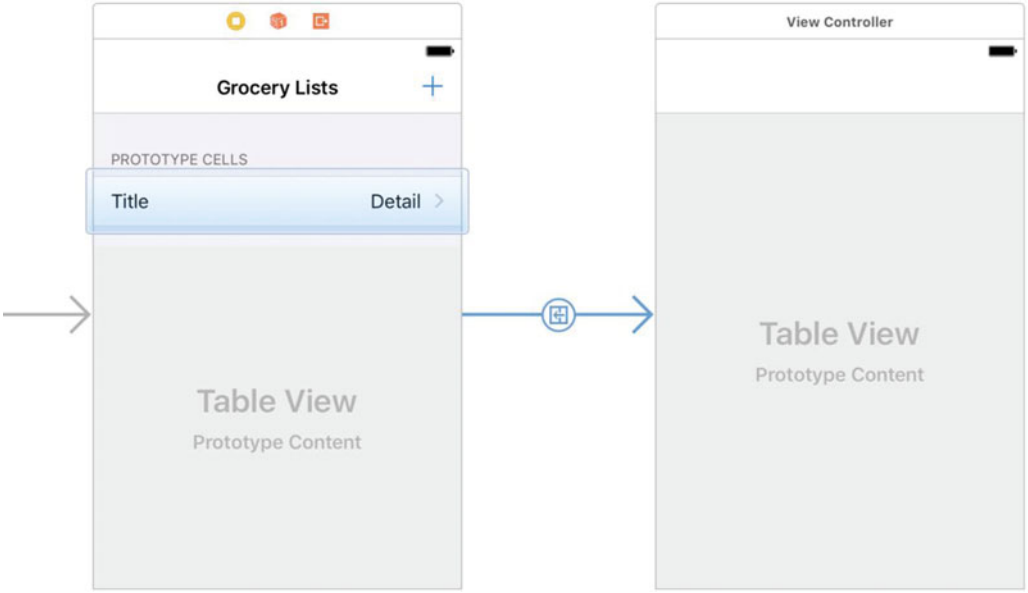


Figure 9-18. *The fully configured Grocery Lists view controller with a segue between the table view cell and the second view controller*

Awesome! The only thing left to do is to finish up our second view controller. I am not going to walk you through this one. You already know how to configure this view controller properly. All it needs is a table view cell with the correct style/reuse identifier, and a navigation item with a title (*Grocery Items*) and + button. Figure 9-19 shows both fully configured view controllers. If you have any trouble configuring this view controller, go back through this section because the steps are identical.

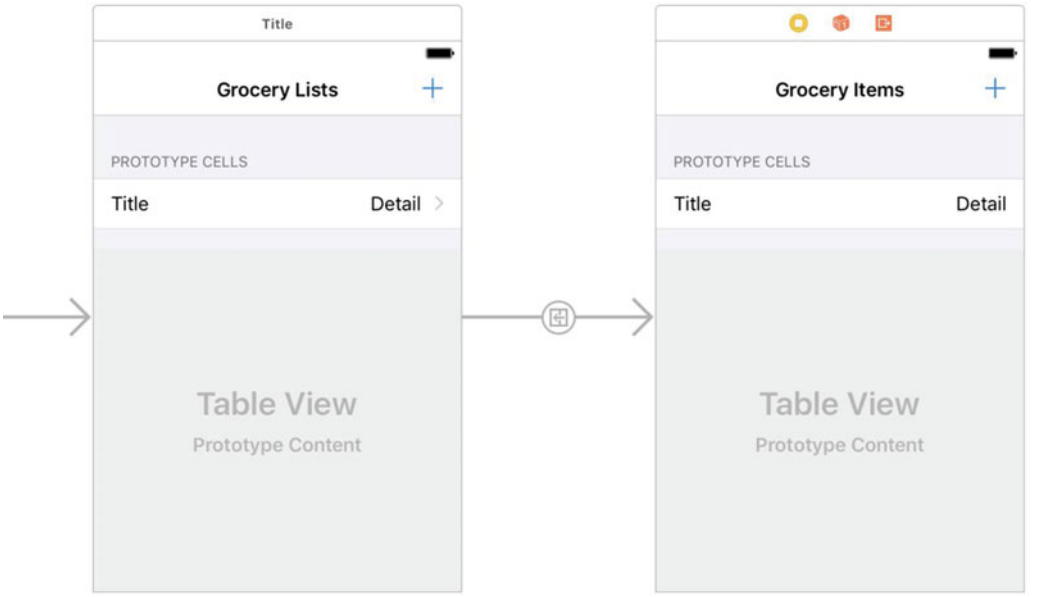


Figure 9-19. *The Grocery List view controller and the Grocery Items view controller fully configured*

All right, we have just finished this section and there was a lot here. We have finished two out of the four view controllers that make up our app. You saw how to embed view controllers in a navigation controller, how to add table views with table view cells, and how to add and configure a navigation item. Up next, we are going to build the last two view controllers in our app. The good news is that we have already built one of the screens in a previous chapter.

Adding Lists and Items

Way back in Chapter 7, we built the view from Figure 7-2. I have placed Figure 7-2 again here for your convenience. This view has a label and text field, constrained to the top of the view. There are also two buttons constrained to the bottom. The two buttons are also constrained to be the same width and height.

Figure 7-2. *The interface for adding grocery lists*

Since we built this view previously, we are not going to walk through every step to building this view. However, we do need to link this view in our storyboard, so this is what we are going to cover in this section. The first step is to add a new view controller to our storyboard. Let's add this view controller directly below the Grocery List view controller. Figure 9-20 shows our storyboard with the new view controller.

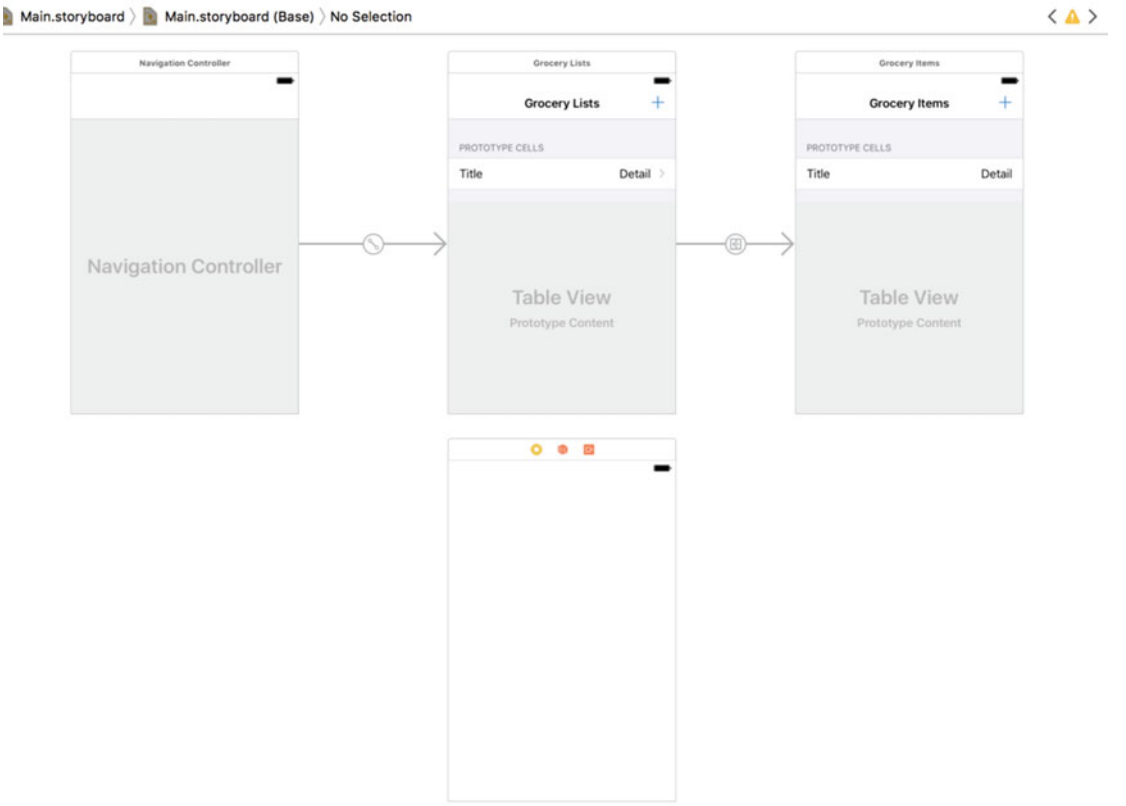


Figure 9-20. *Our new view controller for adding grocery lists directly below the Grocery List view controller*

Now, we need a segue to this new view controller and we want it to show by tapping the + button in the top right of the Grocery Lists view controller. Just like our table cell, let's Ctrl-drag from the + button to our new view controller. However, this time, instead of selecting the *Show* option, let's choose *Present Modally*. Selecting this option will allow our view controller to present over top of the Grocery List view controller instead of using the navigation controller to push onto the navigation stack.

In Figure 9-21, you can see the link between the Grocery List view controller and our new view controller via our navigation item's + button. There is not much storyboard work left. We have already built the interface for our new view controller, so I am not going to cover the specifics here. We only have one view left. Before we get ahead of ourselves, let's finish this out strong.

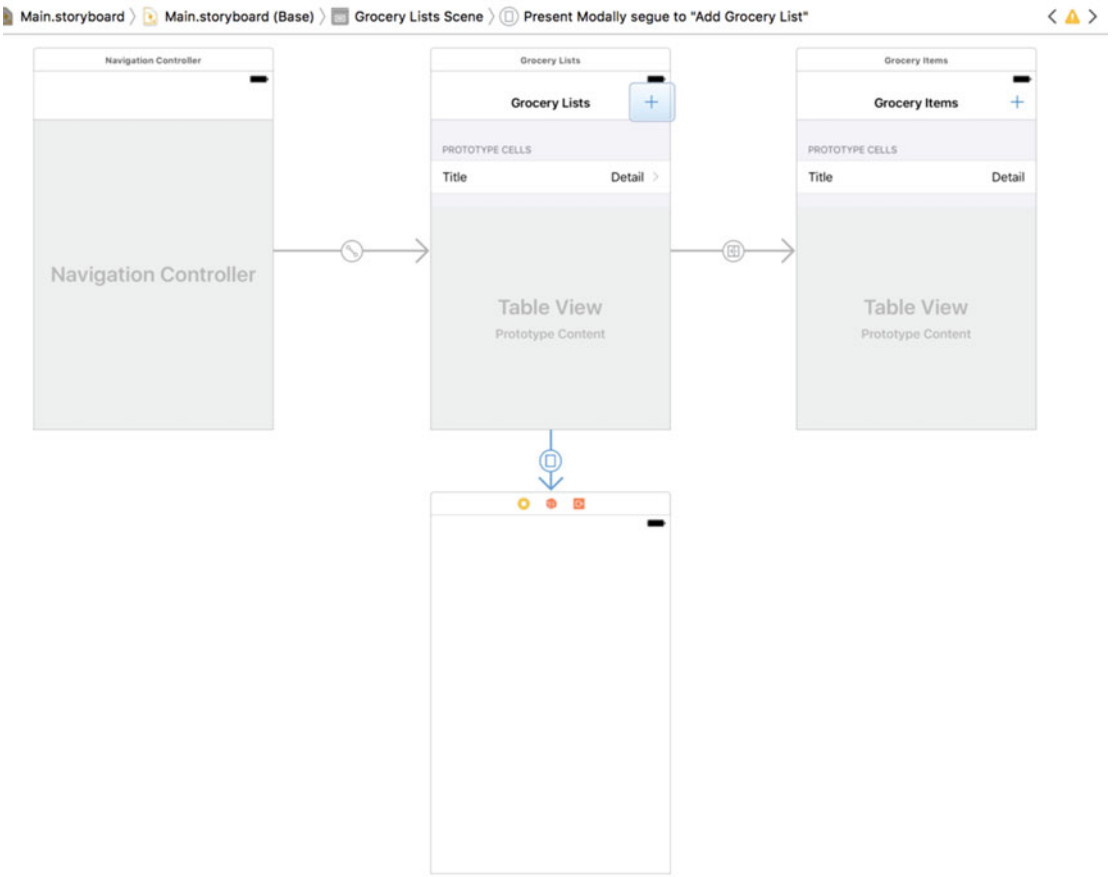


Figure 9-21. *Our new view controller linked via a segue to the Grocery List view controller*

Next, I want you to try and build the new view controller. If you need to refer to Chapter 7, that's all right. Once you have it ready, try changing the device configuration and orientation to make sure the interface works for all screens. Figure 9-22 shows my storyboard after finishing the Add Grocery List view controller.

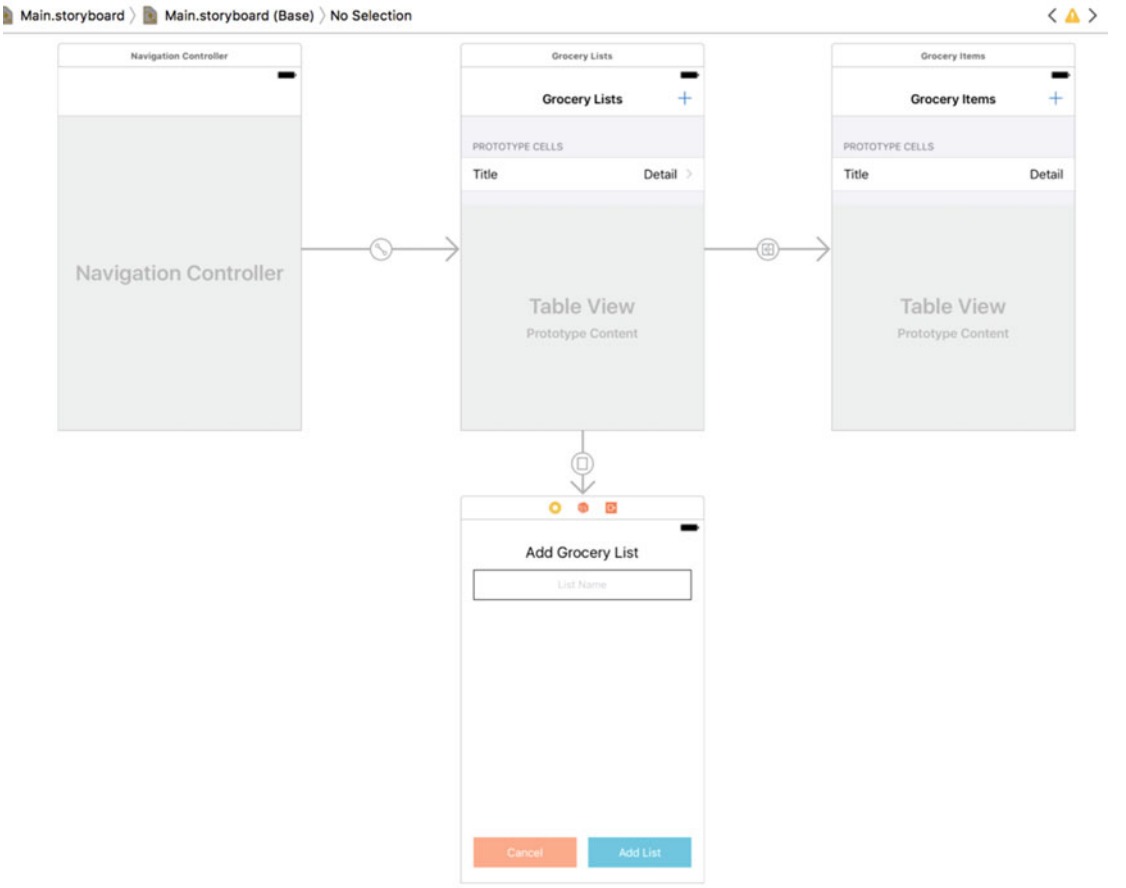


Figure 9-22. My storyboard after finishing the Add Grocery List view controller

We are almost there. There is just one view controller left. This next view controller is going to be almost identical to the previous view controller. The only difference is there will be an extra text field. Figure 9-23 shows the last view controller we are going to build in this chapter.

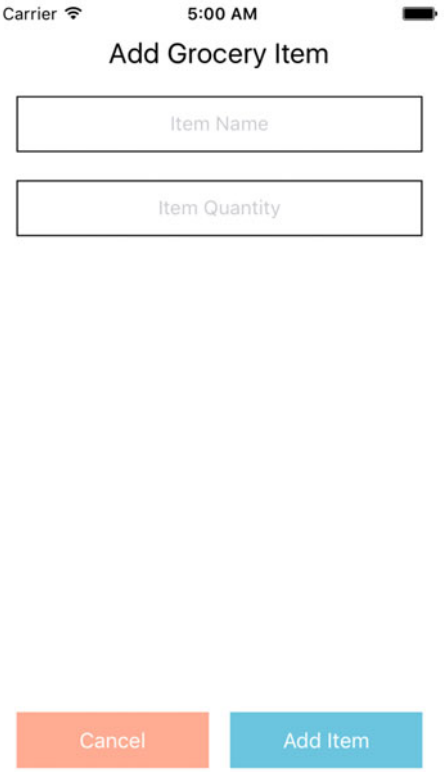


Figure 9-23. *The last view controller we are going to build*

For this view controller, I want you to copy the previous view controller. It might be good practice to rebuild it. Then place the copy underneath the Grocery Items view controller. Then, just like in Figure 9-21, let's add a segue between the Grocery Items navigation item + button and our last view controller. Once this is done, the flow of our app is complete. We only have one last thing to do. Let's add the new text field. You can either copy the text field that is already on the view, or you can drag in a new one from the Object Library. Whichever method you choose, the settings are going to be almost identical to the existing text field. The main difference for this text field is going to be the keyboard type. This text field is so the user can input the quantity of the grocery item. We do not want the user to be able to type words here, so we want to make this a *Number Pad Keyboard Type*, which can be seen in Figure 9-24.

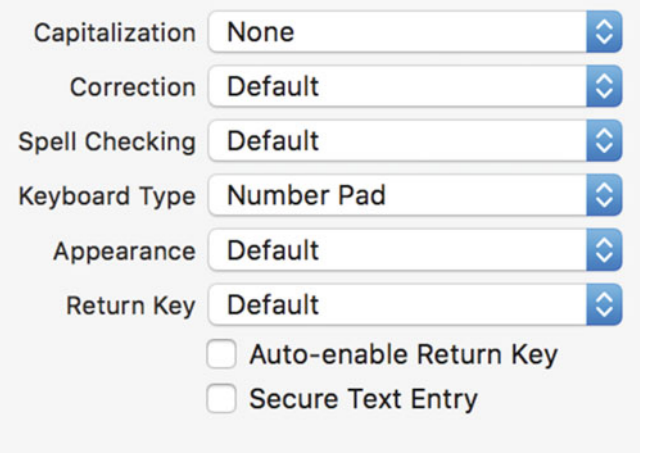


Figure 9-24. *The keyboard settings for the new text field. The keyboard type setting should be a number pad for this text field*

And finally, we need constraints. The constraints for this text field are going to be leading and trailing constraints, with a top constraint to the first text field and a height constraint of 40. Figure 9-25 shows the Size Inspector with all of these constraints.

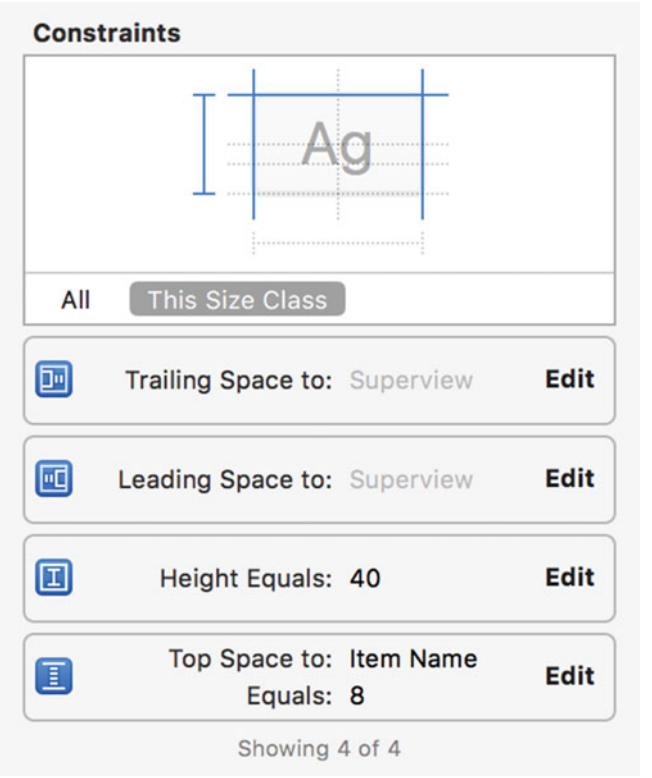


Figure 9-25. *The four constraints in the Size Inspector on the new text field*

And we're done! We just built the entire interface for our app. There was a lot to this chapter and I hope you had fun building the view. If you are sick of Interface Builder, don't worry, because we only have minimal work left in Interface Builder for this app. Most of the work in Chapters 10–12 will be coding. One last figure; Figure 9-26 shows the final storyboard for our app. Make sure your storyboard looks like Figure 9-26 before moving on to the next chapter, or make sure you understand the differences, if you deviated from the design.

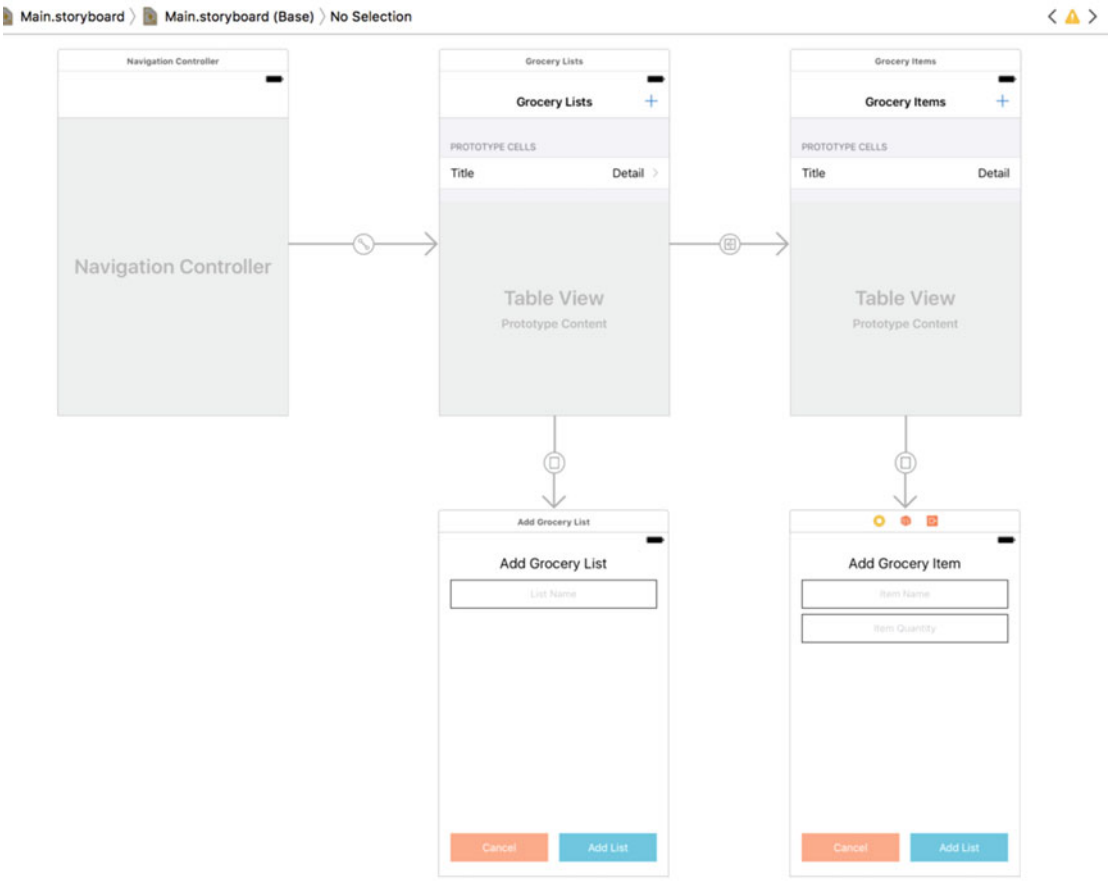


Figure 9-26. *The final interface for the grocery app after following this chapter*

Wrap Up

That's a wrap for Chapter 9. In this chapter, we saw how to build the interface for our app, but we also took a larger look at storyboards in general. We started this chapter with a discussion of the differences between Nib files and storyboards. Even though Nib files might be considered the old way of developing interfaces in iOS, they are still very useful and we should not discount them. We also have to understand the limitations in storyboards. Collaboration on storyboards can be difficult.

The main focus of this chapter was building the interface for our app. Usually, we would not be able to knock it out all at once. Many times, I have just built sections of my app and then started coding the view controllers and other pieces. However, our app was simple enough that we could build this all in one go. On more complicated projects, it can be useful to flesh out code before building the entire interface because the code can provide ideas of how to structure your interface. I have also run into situations where my view influenced the code, so just do what's right for your project and have fun with it.

The next chapter is what you have been waiting for. We are going to start coding! It's time to pull in all the concepts from this book and our playground reference to build our app. Let's get to it.

Chapter 10

Grocery App: MVVM

We have finished the interface for our app! We have all of the views in place; we just need the code behind them now. In Chapter 4, we discussed design patterns; specifically MVVM, which is the design pattern we use to build the Grocery App. We could use pure MVC with this app, but I wanted to show you how we can achieve a very clean and concise app using this pattern.

There are other advantages to using MVVM as well, which we will explore in later chapters. This chapter will also showcase our use of *dependency injection*. We will use dependency injection extensively to make sure we are not coupling ourselves and to stay extensible. This chapter is going to be light in the protocols, but we will explore more protocol-oriented programming techniques in Chapter 12. This is going to be a fun chapter, so let's get started.

What You'll Learn

This chapter is going to be all about the MVVM design pattern with a little MVC thrown in. Since we are working in iOS, we cannot escape the MVC pattern. We are first going to look at how we can start implementing this design pattern without real data behind our app. Setting up Core Data can be a chapter by itself, so I wanted to avoid that somewhat confusing process until we have the foundation of the app in place.

The MVC part is referring to our view controllers. We need to implement the view controllers that are going to control the two table views from the interface in Chapter 9. These two view controllers are going to challenge how data flows through our app. This is a very important concept and as with everything, there are multiple ways we can accomplish this task. We are going to discuss a few of our options and then make it so. This chapter is all about design patterns, so by the end, you should feel comfortable selecting and implementing design patterns for your future projects.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_10

Grocery Lists

Let's go over the background for our app before we just jump right in. You have already seen the interface for the Grocery App, but what exactly is each piece going to do? The idea for the app is that users will be able to create and manage grocery lists for when they go shopping. Each list will consist of a group of grocery items. Each item will have a name and a quantity associated with it. That is the entire app and we are going to add all its functionality in this chapter and in Chapters 11 and 12.

For now, we are going to focus on the functionality for our grocery lists. The grocery lists are the first thing users see when they run our app. We are going to start this section with the placeholder data. This fake data will allow us to build up the functionality free from Core Data. Once we complete the fake data, we will integrate with the MVVM pattern, and finally, we will finish this chapter by implementing our Grocery Lists view controller.

Let's start by closing our storyboards and opening the Project Navigator in Xcode. Unless you changed the name of the file/class, you will most likely see a file called `ViewController.swift`. Figure 10-1 shows the files in our project after setup with the Single View template.

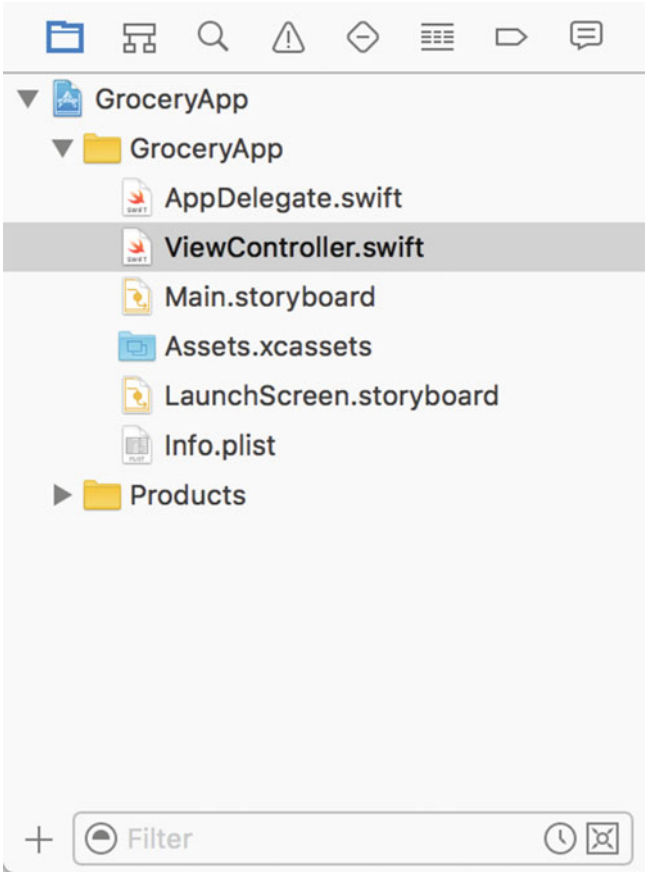


Figure 10-1. The Project Navigator in Xcode with the files created from the Single View template

We are starting fresh. Before we get to the view controller or even to our view model, we need to discuss our placeholder data. This is going to be what we address first, and it will kick start the MVVM pattern.

Placeholder Data

We are going to start this off by creating a new file and group, called `Placeholder.swift` and Data, respectively. Figure 10-2 shows the Project Navigator after we've adding the new file and group. If this is going a little slow for you, don't worry, we are going to pick things up soon.

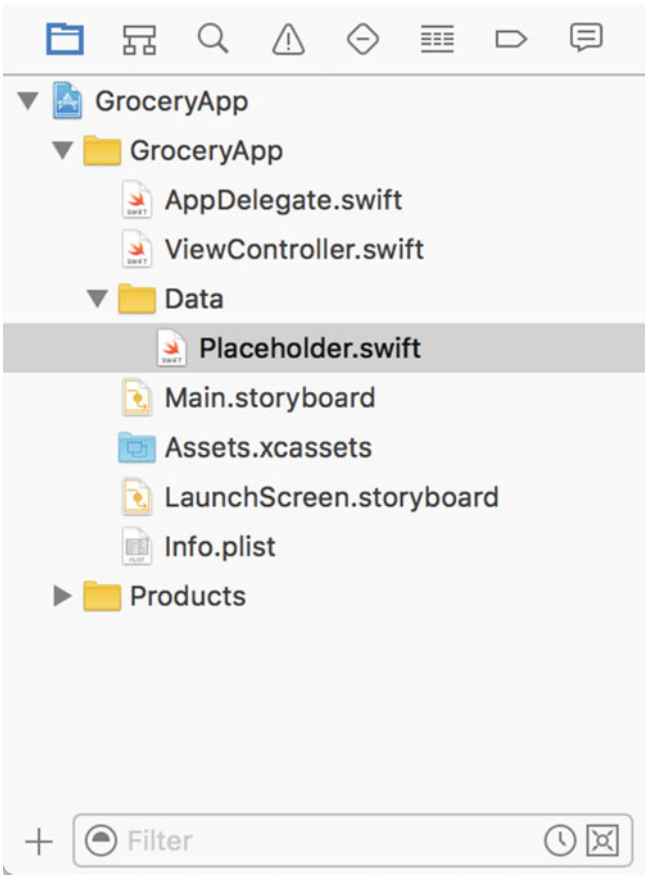


Figure 10-2. *The Project Navigator with the new Placeholder.swift file and the new Data group*

Now, you might be asking yourself, wouldn't it be easier in the long run to just integrate Core Data now? Aren't we going to need to refactor our app if we just use fake data? Well, no.☺ The following code block should be placed at the top of the `Placeholder.swift` file:

```
import Foundation

// Typealiases that mimic future objects
```

```
typealias GroceryList = (name: String?, items: NSSet?)
typealias GroceryItem = (name: String?, quantity: Int16)
```

These two typealiases are going to be our secret sauce. These two aliases just describe tuples that have named values, which allows us to access these tuple elements without having to use the usual `.0`, `.1`, `.2`, and so on. We are also using an `NSSet` instead of an array because Core Data represents lists as `NSSets`, not arrays. The `Int16` type is also just to appease Core Data types. Using these tuples, we are going to pull the tablecloth out from under the dishes, figuratively. We are going to reference these types throughout our app like they are our real model classes. Then, when it comes time to integrate Core Data, we will not have to refactor much code. This is why we have used Core Data types for the properties. Pretty cool, huh? Before I get too ahead of myself, let's finishing faking our data:

```
class DataContainer {
    var data: [GroceryList] = [
        GroceryList(name: "List 1", items: NSSet(array: [
            GroceryItem(name: "Milk", quantity: 2),
            GroceryItem(name: "Eggs", quantity: 12),
            GroceryItem(name: "Cereal", quantity: 1)
            ])),
        GroceryList(name: "List 2", items: NSSet(array: [
            GroceryItem(name: "Cookies", quantity: 20),
            GroceryItem(name: "Bread", quantity: 1),
            GroceryItem(name: "Cheese", quantity: 2)
            ]))
    ]
}
```

This is just a simple `DataContainer` class that has an array of our `GroceryList` tuples. I find it really cool that we can reference tuples like this. In the preceding code block, it looks like we are instantiating new objects, but really, they are just tuples. Feel free to put in any data you want. Okay, we have our fake data, but now what? Here's where protocols come into play. We need a protocol that can be swapped just like our type aliases. The following code block describes this protocol:

```
protocol PlaceholderDataContainer {
    var container: DataContainer { get set }
}
```

This protocol is pretty simple. It only requires a `DataContainer` property. Now, let's conform to this protocol in our `AppDelegate` class:

```
import UIKit
import CoreData

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, PlaceholderDataContainer {
```

```
    var window: UIWindow?
    var container: DataContainer = DataContainer()

    // ...
}
```

Now, our `AppDelegate` class conforms to our protocol and has a container of fake data. This is the link that will make our entire app work later on. That's it for our fake data. If it seems confusing at this point, don't worry, it's about to clear up in the next section. We are going to start implementing the MVVM pattern. Before we move on, here is the entire `Placeholder.swift` file for reference:

```
import Foundation

// Typealiases that mimic future objects
typealias GroceryList = (name: String?, items: NSSet?)
typealias GroceryItem = (name: String?, quantity: Int16)

class DataContainer {
    var data: [GroceryList] = [
        GroceryList(name: "List 1", items: NSSet(array: [
            GroceryItem(name: "Milk", quantity: 2),
            GroceryItem(name: "Eggs", quantity: 12),
            GroceryItem(name: "Cereal", quantity: 1)
            ])),
        GroceryList(name: "List 2", items: NSSet(array: [
            GroceryItem(name: "Cookies", quantity: 20),
            GroceryItem(name: "Bread", quantity: 1),
            GroceryItem(name: "Cheese", quantity: 2)
            ]))
    ]
}

protocol PlaceholderDataContainer {
    var container: DataContainer { get set }
}
```

View Model

We just implemented our fake data; so let's start using it. The purpose of the MVVM pattern in iOS is to keep your model data away from the view controller. This means our view model is going to hold onto the data, and in our case, the fake data. Let's start this by creating another group and file. The group should be called Objects and the file `ViewModel.swift`. You can see our progress in the Project Navigator in Figure 10-3.

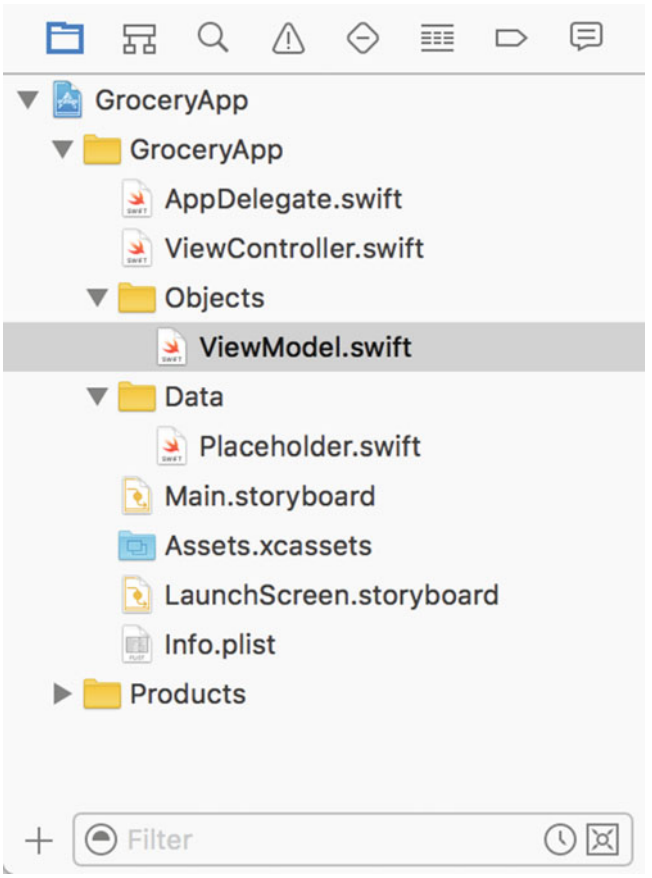


Figure 10-3. *The Project Navigator after creating our Objects group and ViewModel.swift file*

Now, in our `ViewModel.swift` file, let's create a `ViewModel` class and a property that references our `PlaceholderDataContainer` protocol. We also want a computed property to access the protocol's `DataContainer` property. This following code block implements this functionality:

```
import Foundation

class ViewModel {

    // MARK: - Properties

    var placeholderContainer: PlaceholderDataContainer?
    var dataContainer: DataContainer? {
        return placeholderContainer?.container
    }
}
```

So, we obviously need to get a reference to the PlaceholderDataContainer and currently the only thing that conforms to this protocol is our AppDelegate. Let's add an initializer to this that uses the AppDelegate, but we want to make sure we use dependency injection:

```
import Foundation
import UIKit

class ViewModel {

    // MARK: - Properties

    var placeholderContainer: PlaceholderDataContainer?
    var dataContainer: DataContainer? {
        return placeholderContainer?.container
    }

    // MARK: - Initializer

    init(placeholderContainer: PlaceholderDataContainer? = UIApplication.shared.delegate as?
AppDelegate) {
        self.placeholderContainer = placeholderContainer
    }
}
```

So what happened here? We created a new init method on our view model and we allowed a PlaceholderDataContainer to be injected. We have also had this property default to using the AppDelegate when nothing is injected. Technically, this code allows the PlaceholderDataContainer to come from anywhere in the app. For our purposes, this will only ever be the AppDelegate, but it does not have to be.

That is it for the base view model class. There is not much to it, but that's okay. Before we move on to the view controller layer, let's build one more view model. Luckily the next view model will be more fun. The next view model we are going to create will be the GroceryListsViewModel. As always, let's create a new group called Grocery Lists and a new file called GroceryListsViewModel.swift under the new group.

This file is going to contain the GroceryListsViewModel class that inherits from our base ViewModel class. This class needs an array with type GroceryList. This property is going to be a computed property that will use the data within our DataContainer class. This property is optional and our array should not be, so let's use the Nil Coalescing operator. This operator saves us the trouble of an if statement by using the property if it is not nil or the supplied non-optional property in the other case. Let's see this new functionality:

```
import Foundation

class GroceryListsViewModel: ViewModel {

    // MARK: - Properties

    var groceryLists: [GroceryList] {
        return dataContainer?.data ?? []
    }

}
```

We have one last thing to add here. We need a method that can give us the data for a specific index. We already know we are going to use tables, so let's allows this API to use the `IndexPath` object. We also want to consider the Swift 3 API design guidelines. This method will not have any side effects, so we can use a noun here:

```
class GroceryListsViewModel: ViewModel {

    // MARK: - Properties

    var groceryLists: [GroceryList] {
        return dataContainer?.data ?? []
    }

    // MARK: - Grocery List Data

    func groceryList(at indexPath: IndexPath) -> (name: String?, itemCount: Int) {
        let list: GroceryList? = groceryLists[indexPath.row]
        return (list?.name, list?.items?.count ?? 0)
    }
}
```

This method just uses the computed array property to access the appropriate grocery list and returns the name and item count. We want to return a tuple with this data so our future view controllers can stay ignorant of the data. They do not need to deal with parsing the data off the object, just displaying it. Before we call it a day on this class, I have one last thing to do. In Chapter 1, I talked about accounting for *optionals* in our code. Now, we have done a pretty good job so far. We have used optionals where appropriate and we have no implicit unwrapping taking place. We did, however, access an index of the grocery lists array.

It is not obvious from the previous code block, but there is a crash where we access the grocery lists at the given `indexPath`. If you look at the documentation for the subscript method, it says the following:

> *The position of the element to access. position must be a valid index of the collection that is not equal to the endIndex property.*
>
> —Apple Subscript API Reference

This is right from Apple's documentation and you can see it only expects valid indices. If we were to give this method an `IndexPath` object that is out of bounds, our app crashes. Now, the odds of this are slim. But there is a little bit of work we can do to mitigate even this small risk. We are going to implement an extension on the `Array` type. So, let's create our new group and new file called Extensions and `Array+Extensions.swift`, respectively. Let's take a look at our Project Navigator to see our progress in Figure 10-4.

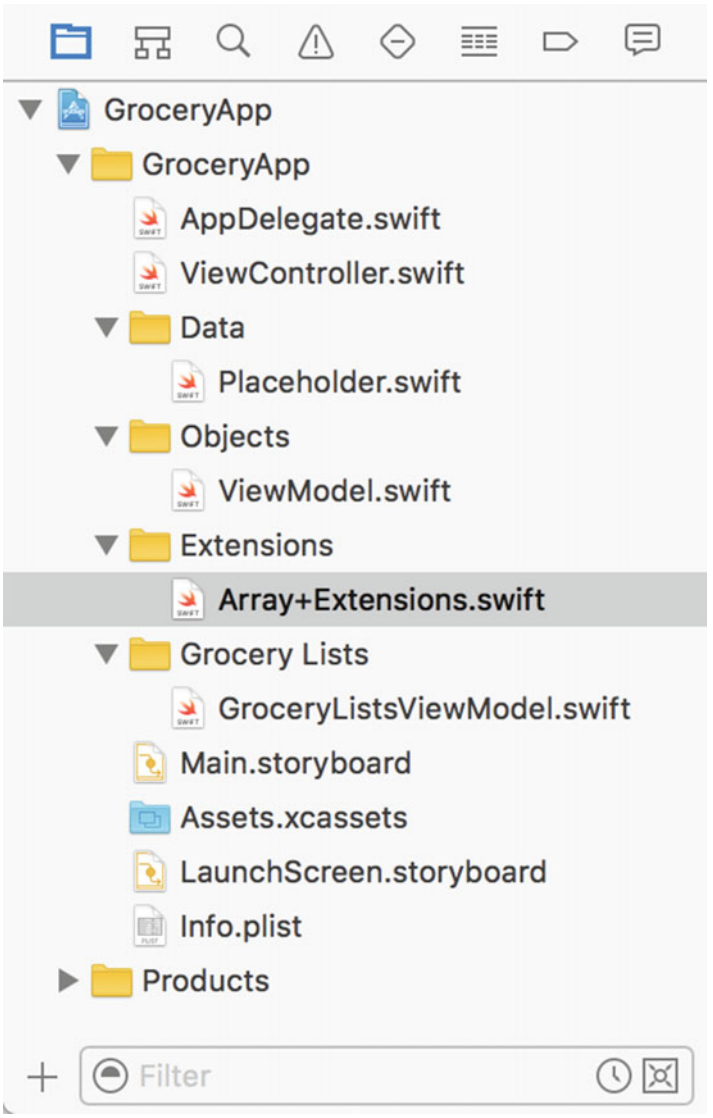


Figure 10-4. *Our progress in the Project Navigator after creating the Extensions group and Array+Extensions.swift file*

This extension is going to be simple, so before we implement our code, let's start writing some tests. You knew it was coming, just not when. We have discussed testing and test-driven development (TDD). This is a really good place to start with TDD to get some experience. If you included a unit test target when you created your project, you should be all ready to go. If you didn't, let's see how to do that here.

Navigate to the project settings where you will see Figure 10-5. We have all of the different targets for our app and their settings here.

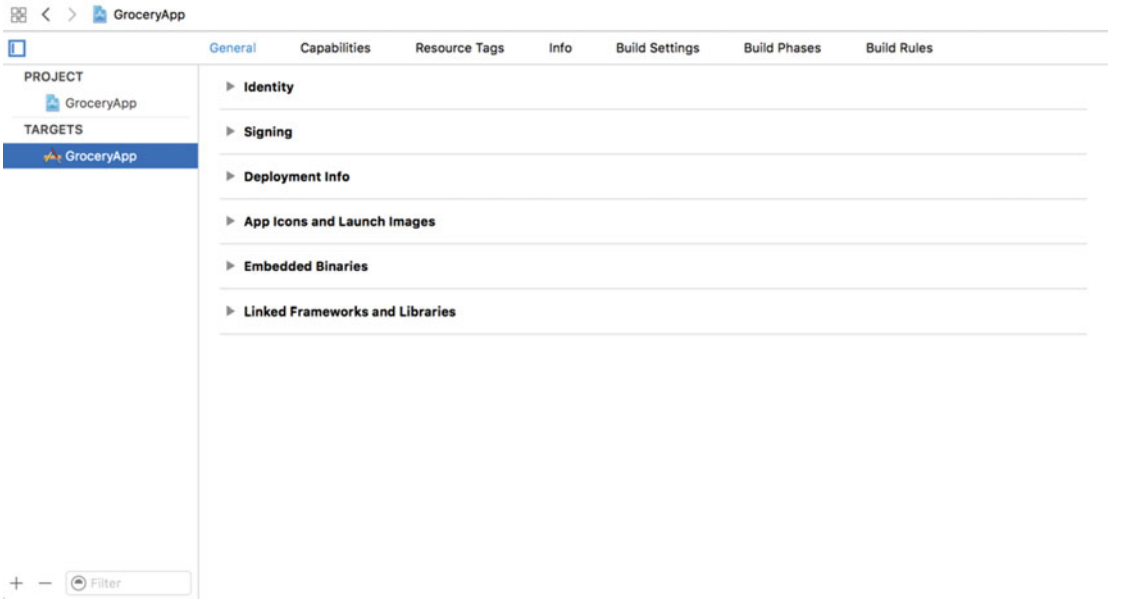


Figure 10-5. *The settings for our app in Xcode including the app's targets*

Looking at the bottom-left corner of Figure 10-5, you will see a little + button. Clicking this button will allow you to create a new target. Figure 10-6 shows the New Target menu. There are a lot of options here, such as the new iMessage extension, but if you search for tests, you will find the two options listed in Figure 10-6.

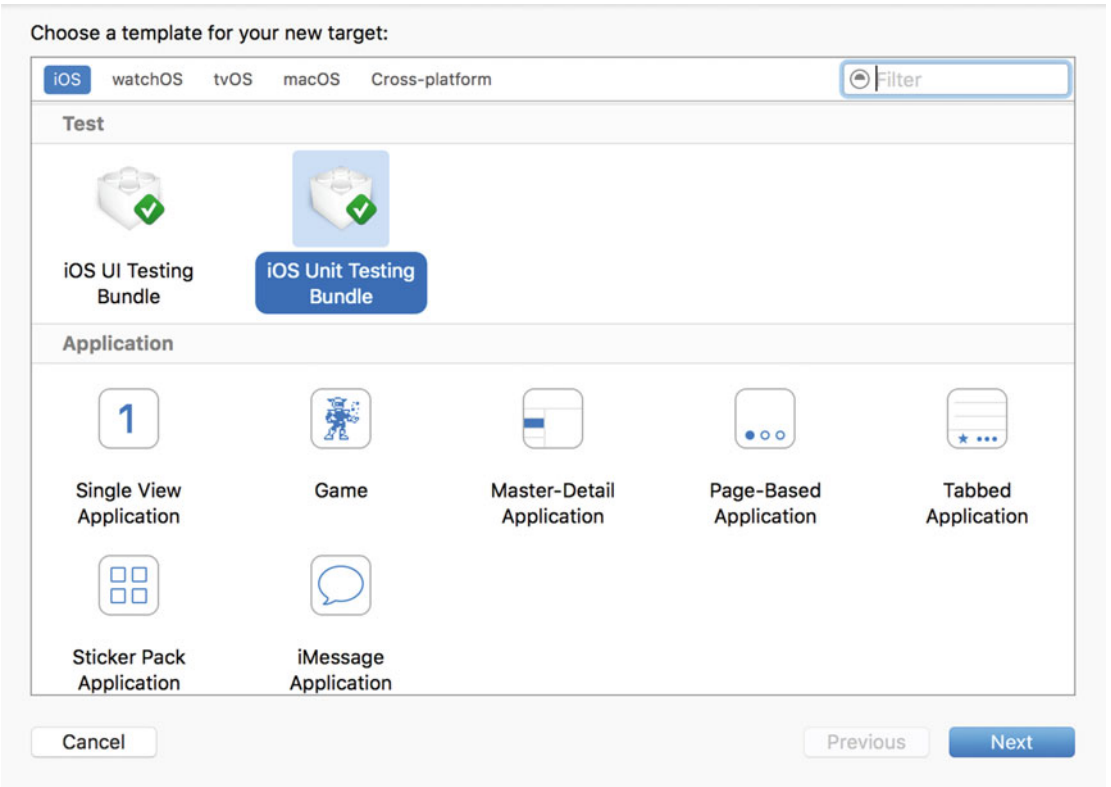


Figure 10-6. *The New Target menu after clicking the + button in the bottom-left corner of Figure 10-5*

Let's select the *iOS Unit Testing Bundle* and click *Next*. Follow the next menu's instructions making sure you have the *Project* setting and the *Target to Be Tested* option set to *GroceryApp*. You can see the settings I chose in Figure 10-7.

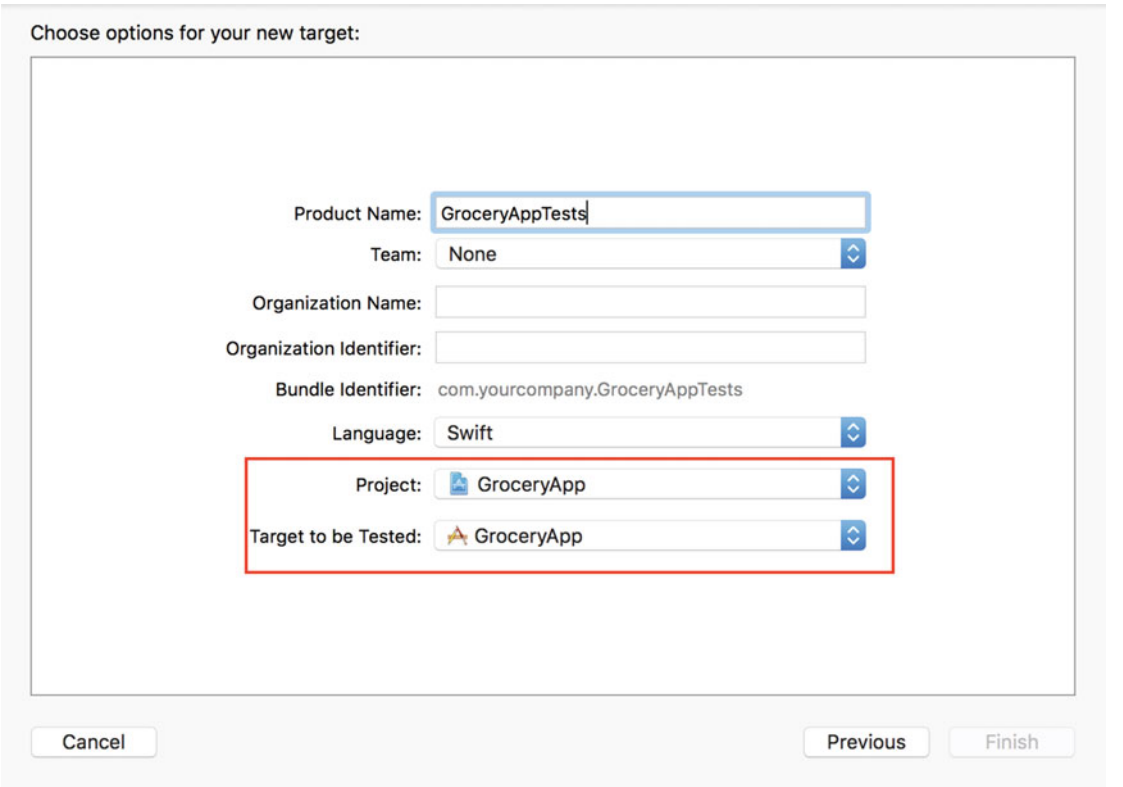


Figure 10-7. *The settings screen for creating new targets*

Once everything looks right, click *Finish* and we will have our new testing target. So, now if we look at the Project Navigator, we have a brand new group called GroceryAppTests with a corresponding file `info.plist`. Let's start by deleting the `GroceryAppTests.swift` file and creating a new file called `Array+ExtensionsTests.swift`. Figure 10-8 shows the file dialog. We want to make sure we have selected the GroceryAppTests for Target Membership of this file. This means the file will be linked/compile with the settings of the GroceryAppTests target instead of the GroceryApp target.

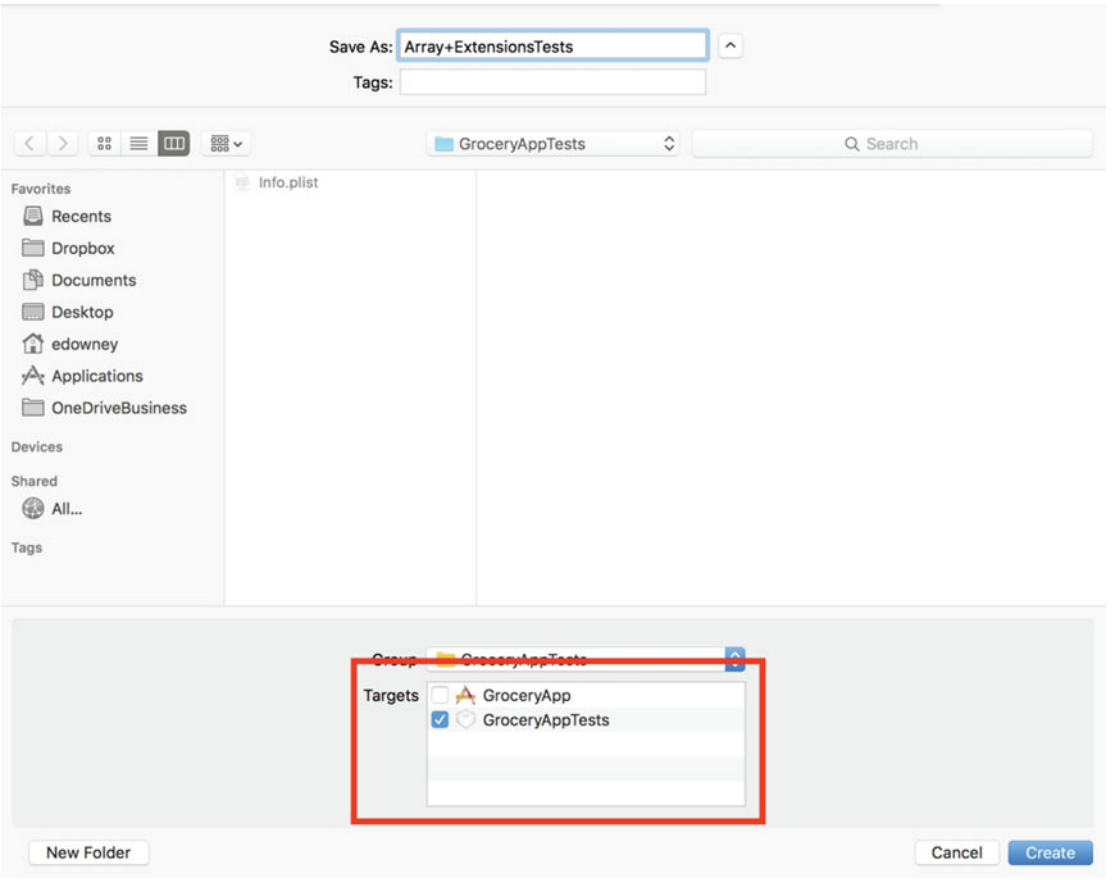


Figure 10-8. *The file dialog screen with the target membership set to the GroceryAppTests*

All right, we have our testing target and we have our first test file. Let's start writing our array extensions tests. The first thing we need to do is make sure our test file looks like the following:

```
import XCTest
@testable import GroceryApp

class Array_ExtensionsTests: XCTestCase {

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

}
```

As discussed in Chapter 8, we need to import the `GroceryApp` module, and the `@testable` attribute allows the test to access the internal members. So, let's go over the requirements for our array extension. First, we do not want our app to crash if we try to access an element

beyond the endIndex. We also do not want to crash if we try to access anything below zero, and we need it work as normal if the index we use is valid. To me, this looks like three tests. Let's start with an easy one, the valid index test:

```
class Array_ExtensionsTests: XCTestCase {

    var subject: [String]!

    override func setUp() {
        super.setUp()

        subject = ["A", "B", "C"]
    }

    func testShouldGetTheCorrectValue() {
        let index = 1

        let result = subject.value(at: index)

        XCTAssertEqual("B", result)
    }

}
```

First, we have a subject at the top of our test class. We then instantiate our subject in the setUp method. The test then just asks for a value at the specific index. You can see we are adhering to the new API design guidelines as best we can. This test will not compile, so let's flip over to our Array+Extensions.swift file. Let's just create the value(at:) method so it is basically just the subscript method:

```
import Foundation

extension Array {
    func value(at index: Int) -> Element? {
        return self[index]
    }
}
```

This is basically the same thing we have in our GroceryListsViewModel. It is important to notice we are using the Element type. Array is a generic struct where one of the types is Element. This allows us to have an array for any type. Now, our tests should compile and run. They should also pass. You could say the tests drove the implementation. We'll take a break for TDD in a second, but let's finish this extension first. Now, let's go back to our Array+ExtensionsTests.swift file and write some more tests. We are going to start with the test for the endIndex:

```
class Array_ExtensionsTests: XCTestCase {

    var subject: [String]!
```

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

        subject = ["A", "B", "C"]
    }

    func testShouldGetTheCorrectValue() {
        let index = 1

        let result = subject.value(at: index)

        XCTAssertEqual("B", result)
    }

    func testShouldGetNilForAnIndexBeyondEndIndex() {
        let index = 10

        let result = subject.value(at: index)

        XCTAssertNil(result)
    }

}
```

The second test we have is expecting `nil` for an index of 10. If you build and run this test, it's going to crash. Now, before you get sea sick, let's finish out the tests here before we go implementing. The last test we need to write is to make sure we receive `nil` when we ask for a negative index:

```
class Array_ExtensionsTests: XCTestCase {

    var subject: [String]!

    override func setUp() {
        super.setUp()

        subject = ["A", "B", "C"]
    }

    func testShouldGetTheCorrectValue() {
        let index = 1

        let result = subject.value(at: index)

        XCTAssertEqual("B", result)
    }

    func testShouldGetNilForAnIndexBeyondEndIndex() {
        let index = 10

        let result = subject.value(at: index)
```

```
        XCTAssertNil(result)
    }

    func testShouldGetNilForAnIndexLessThanZero() {
        let index = -5

        let result = subject.value(at: index)

        XCTAssertNil(result)
    }

}
```

The previous code block is the entire `Array+ExtensionsTests` class. The last test here tries to access an index of –5 and this will crash. Now, let's go back to the `Array+Extensions.swift` file and make these tests pass:

```
extension Array {
    func value(at index: Int) -> Element? {
        guard index >= 0 && index < endIndex else { return nil }
        return self[index]
    }
}
```

Now, run your tests and see what happens. These should all pass. Let's take a look at this. We are using a `guard` statement to make sure the index we are injecting is in the correct range. We just return `nil` otherwise. This was a very simple example, but this is TDD. We also did not follow the practice perfectly. The practice would have had us implement the logic after our second test to make it pass. We then would have written the third test and then the last piece of our implementation. It makes me tired just explaining it. This is where we can find the balance. We knew the requirements for this extension. There was nothing stopping us from writing all three tests first. I wanted to give you guys a taste of TDD for the app. We are going to be using a mixture of these techniques for the rest of this app.

Our last task is to refactor the `GroceryListsViewModel`. We need to use this awesome new extension we built. Here is the `groceryList` method for using our new extension:

```
func groceryList(at indexPath: IndexPath) -> (name: String?, itemCount: Int) {
    let list: GroceryList? = groceryLists.value(at: indexPath.row)
    return (list?.name, list?.items?.count ?? 0)
}
```

Almost nothing has changed, but if you look really closely, you can see we are now using the new array extension to safely access our `groceryLists`. Before we move onto the next section, I just wanted to summarize what we have done here.

This section was all about implementing our view model. We created our base view model that allows a protocol type to be injected that can access our placeholder data. We then implemented a grocery list's view model that holds onto the `GroceryLists` type and gives us a safe way to retrieve a specific grocery list. To make sure we can safely access elements within an array, we created an array extension. While implementing this extension, we made sure we were covered with tests. Now, in Figure 10-9, you can see our progress in the Project Navigator; afterward we can finally get to our view controller.

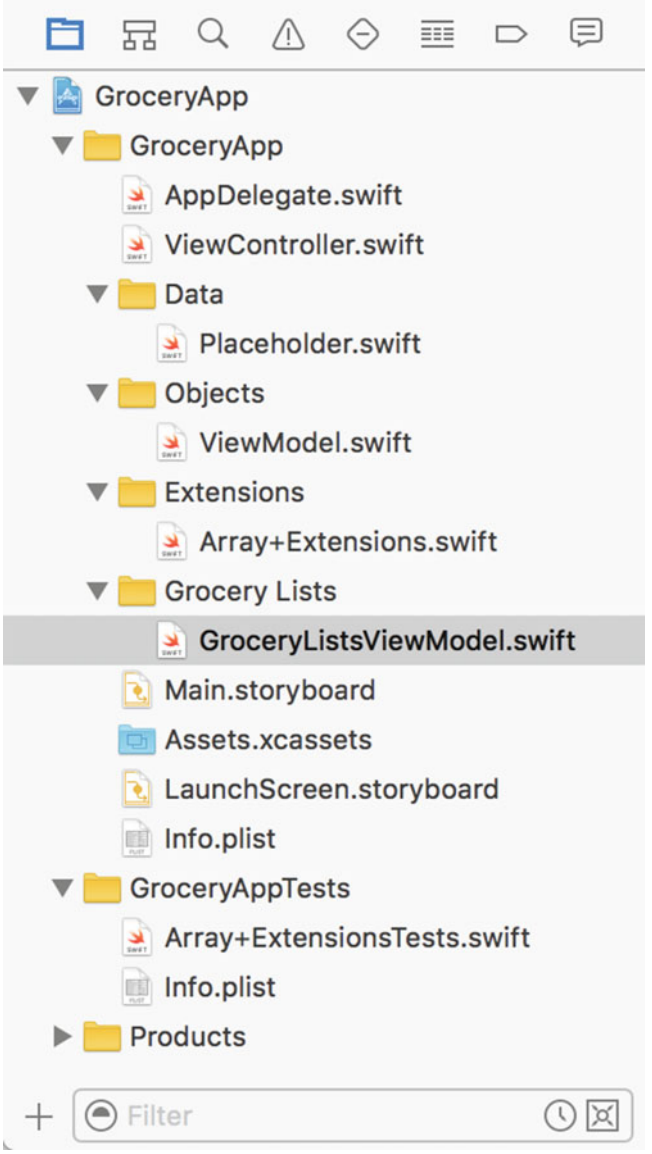


Figure 10-9. *The progress of the Grocery App in the Project Navigator*

Grocery Lists View Controller

We just finished our view model and the Grocery Lists view model. Let's put these two pieces to work and get our fake data displaying on the screen. First, we are going to need a `GroceryListsViewController.swift` file. Let's put this file under the Grocery Lists group. You can either create a brand-new file or reuse the template's `ViewController.swift` file. Either way, we want to create a `GroceryListsViewController` like the following:

```
import UIKit

class GroceryListsViewController: UIViewController, UITableViewDataSource,
```

```
UITableviewDelegate {

}
```

Let's take a look at this class before we add more implementation. We want to make sure our class inherits from the base `UIViewController` class and conforms to the `UITableViewDataSource` and `UITableViewDelegate` protocols. This will give us the methods to connect with the table view on our storyboard. Before we link our table view up, let's add a table view `@IBOutlet` property:

```
class GroceryListsViewController: UIViewController, UITableViewDataSource,
UITableviewDelegate {

    // MARK: - Properties

    @IBOutlet var groceryListTableView: UITableView?

}
```

Now, we can head into our storyboard and link up the `dataSource`, `delegate`, and our new `@IBOutlet` properties. Figure 10-10 shows the interface for the view controller we are going to focus on here.

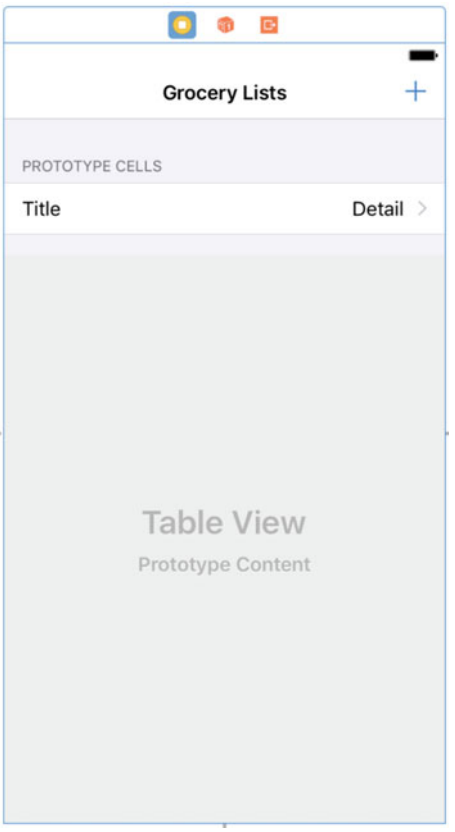


Figure 10-10. *The Grocery Lists View Controller in our storyboard*

To set up the table view `dataSource` and delegate, select the table view and control-drag to the yellow icon at the top of the view controller. This will open up a menu where you select `dataSource`. Repeat this step for the `delegate` as well. Now, before you can hook up the `groceryListsTableView` property, we need to switch the identity of this view controller. In the Utility Pane on the right side, open up the *Identity Inspector* and change the *Class* identity to `GroceryListsViewController` instead of `ViewController`. Figure 10-11 shows the Identity Inspector after changing this property.

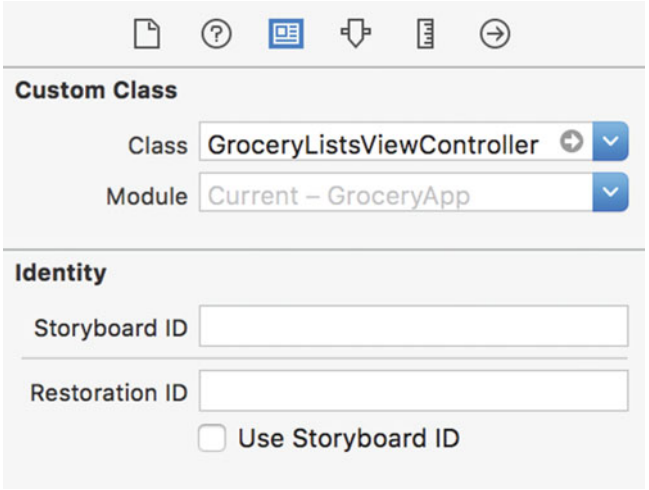


Figure 10-11. *The Identity Inspector for the Grocery Lists view controller*

Now, we can control-drag from the yellow icon representing our view controller to the table view. This lets us set up the `groceryListsTableView` outlet. Our storyboard should now be fully configured for this view controller, so now we can get back to coding.

If you were to try and compile the app now, it would not work. The view controller does not conform to the table view protocols. Let's start here. There are three methods that we want to implement. The following code block has these three methods with hardcoded data:

```
class GroceryListsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate {

    // MARK: - Properties

    @IBOutlet var groceryListTableView: UITableView?

    // MARK: - Table View

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return 0
    }

    func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt
indexPath: IndexPath) {

    }
```

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

The three methods we want to implement are `cellForRow`, `willDisplayCell`, and `numberOfRowsInSection`. These three methods will control the cell that is displayed in our table view, and how many are displayed. First, we are going to implement the `cellForRow` method. This is where we will use the cell reuse identifier we talked about in Chapter 9:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    return tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
}
```

In the preceding code we used the `dequeueReusableCellWithIdentifier` method to instantiate the table cell from our storyboard at the given `indexPath`. Let's next implement the `willDisplayCell` method. But wait, we still need a view model for this view controller! We could do this by adding a view model property and giving our view controller the correct view model, but let's create a protocol for this behavior called `ViewModelContainer`. This should be a new file in a new group called `ViewModelContainer.swift` and Protocols, respectively.

This protocol comes from Chapter 6 where we discussed protocol associated types. This protocol is going to need an associated type and a `viewModel` property. The following code block is the implementation for the `ViewModelContainer`:

```
protocol ViewModelContainer {
    associatedtype VM: ViewModel

    var viewModel: VM { get set }
}
```

Our associated type must inherit from the `ViewModel` class, and that is the only constraint. Now, we can use this protocol on our view controller. So, switch gears, let's add this protocol to the `GroceryListsViewController`:

```
class GroceryListsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer {

    // MARK: - Properties

    var viewModel: GroceryListsViewModel = GroceryListsViewModel()

    // ...
}
```

Conforming to this protocol forces our view controller to have a `viewModel` property and we want it to be of type `GroceryListsViewModel`. Now, we can get back to the `willDisplayCell` method:

```
func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt
indexPath: IndexPath) {
    let list = viewModel.groceryList(at: indexPath)

    cell.textLabel?.text = list.name
    cell.detailTextLabel?.text = "\(list.itemCount) Items"
}
```

Thanks to our view model method `groceryList(at:)`, this method is really simple. We ask for the Grocery List object at the specific `indexPath` and then set the cell's main label and detail label to the correct text. Also, because we handled the optionals in our view model, we do not need to worry about string interpolation doing anything funny such as "Optional(10) items". One method left, `numberOfRowsInSection`. We want to use the count for the grocery lists in our view model:

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

The whole purpose of our grocery lists at the index path method and the view model is to keep the data away from our view controller. This makes the previous code block seem counterintuitive because we are directly accessing the grocery lists data in our view controller. I am doing this so we can be as efficient as possible, but also, it recognizes that iOS will always be MVC and accessing our data in the view controller is not necessarily wrong. We could definitely create a list count method, but I did not want to go that far. If you would like to create that method, go for it.

That does it for this section. We did all of our work up front, so implementing the Grocery Lists view controller was a piece of cake. Go ahead and run your app to see the results. Figure 10-12 shows our app running for the first time. It also shows data! Okay, fake data, but still, pretty cool.

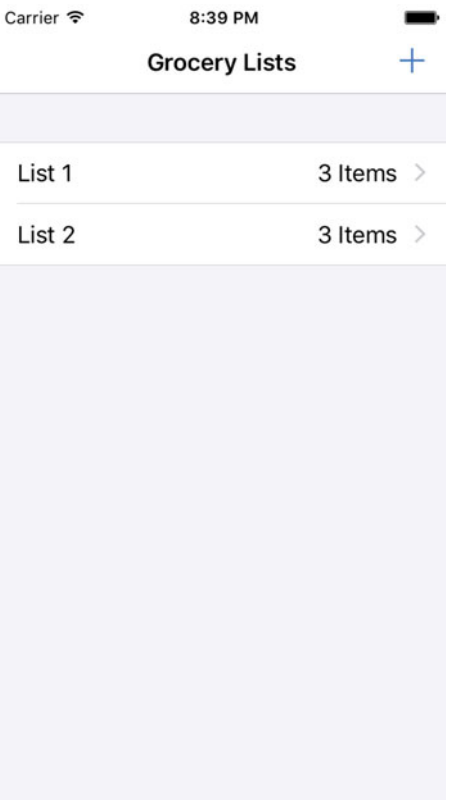


Figure 10-12. *The Grocery Lists app running for the first time with fake data*

If you are not seeing this when you run your app, there are a few things you can try. First, let's talk about what you should do if you are seeing a black screen with no view. When configuring your storyboard, you may have messed up the initial view controller setting. You want the navigation controller in your storyboard to be the initial view controller. If you select the navigation controller in the storyboard and navigate to the Attributes Inspector, there should be a setting for *Is Initial View Controller*. You want to make sure this is checked on. You can see this setting in Figure 10-13.

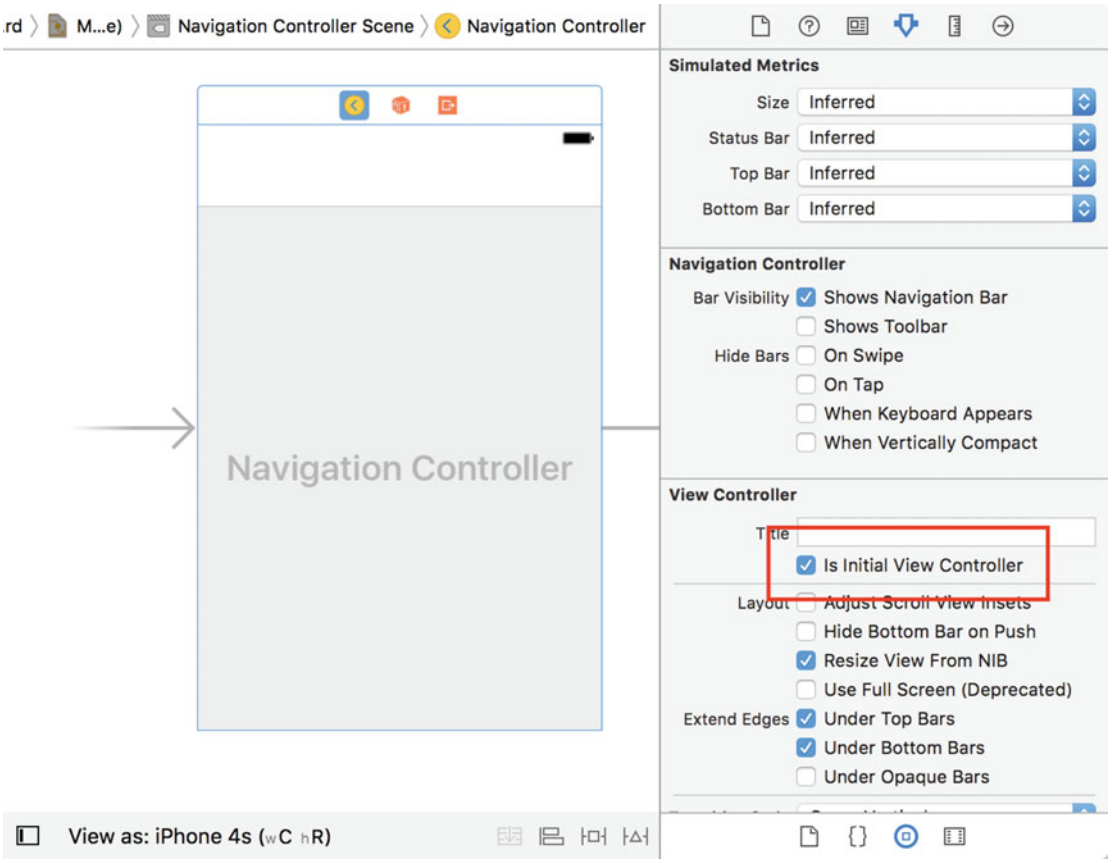


Figure 10-13. *The Is Initial View Controller setting on the Attributes Inspector. Make sure this setting is turned on for the Navigation Controller*

That should do it. If you are still experiencing issues, I would recommend going back through from the start. Make sure, everything is building and running smoothly before moving on to the next section.

Grocery Items

There was a lot of setup in the previous section. In this section, we can use all that work to move faster. Let's quickly cover our current progress. First, we implemented a fake data container that we put on our `AppDelegate`. We then built up a view model base class that allows the type containing our fake data to be injected. We then implemented the view model specific to our grocery list data. And finally, we finished the section by implementing the view controller for the grocery lists, which allowed us to see our fake data running in our app. This section is going to finish up the view controller work, and by the end, we should be able to see our fake data grocery lists and the items associated with those lists. First, let's see our progress in the Project Navigator in Figure 10-14.

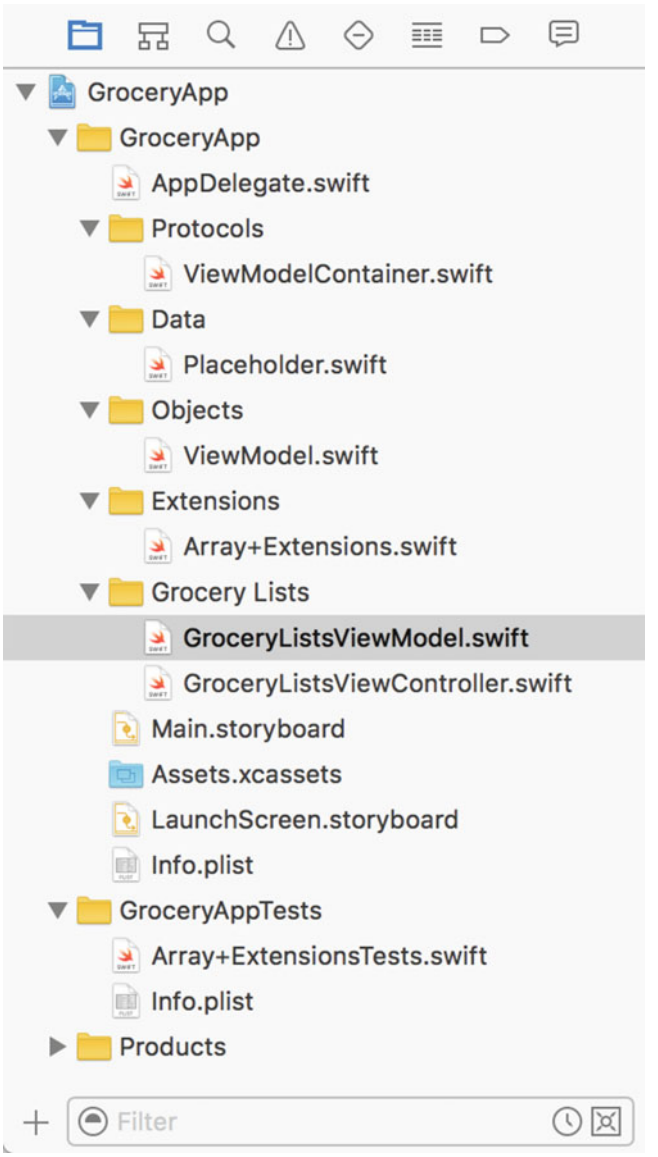


Figure 10-14. The progress up until this point in Xcode's Project Navigator

View Model

We are going to jump right into this section by creating our new view model. This view model is going to be responsible for the grocery items within a specific grocery list. That means this view model is going to look almost identical to the previous `GroceryListsViewModel`. First, create a new group and a new file called Grocery Items and `GroceryItemsViewModel.swift`:

```
import Foundation

class GroceryItemsViewModel: ViewModel {

}
```

We have our new ViewModel subclass, so now we can add some functionality to it. Let's start with an array of grocery items:

```
class GroceryItemsViewModel: ViewModel {

    // MARK: - Properties

    var groceryItems: [GroceryItem] {
        return []
    }
}
```

Just like our other view model, this array is a computed property, and I have just returned a default empty array for now. Before we make this work, let's write the grocery item data method. This method will be another method that returns a tuple of data representing a grocery item at a specific indexPath:

```
// MARK: - Grocery Item Data

func groceryItem(at indexPath: IndexPath) -> (name: String?, quantity: Int16) {
    let item: GroceryItem? = groceryItems.value(at: indexPath.row)
    return (item?.name, item?.quantity ?? 0)
}
```

See? Nothing to it. We are paying close attention to the Swift 3 guidelines, we are using our array extension to make sure this method has no inherit crashes, and we just return the data our view controller needs to display. Now, we need to refocus on the grocery items array. We know our items come from grocery lists, so we are going to need a grocery list property to make this work. Before we discuss how this data will be injected into our view model, let's just use it here. So in the Properties section of this view model, let's add the following code:

```
class GroceryItemsViewModel: ViewModel {

    // MARK: - Properties

    var groceryList: GroceryList?

    var groceryItems: [GroceryItem] {
        return []
    }

    // MARK: - Grocery Item Data

    // ...
}
```

We have our reference to a grocery list, so our computed property can now access the correct data. This is going to be a bit tricky. Remember our grocery list's items are an NSSet, not an array. We are going to need to do a bit of conversion here:

```
class GroceryItemsViewModel: ViewModel {

    // MARK: - Properties
```

```
    var groceryList: GroceryList?

    var groceryItems: [GroceryItem] {
        let result = groceryList?.items?.flatMap {
            $0 as? GroceryItem
        }
        return result ?? []
    }

    // MARK: - Grocery Item Data

    // ...
}
```

We are using one of the functional methods that sequence types have called `flatMap`. `flatMap` uses a closure to iterate over a sequence and asks for a new type back to convert the sequence. So, if we had an array of object A, we could use `flatMap` to covert the array of object A to an array of object B. The difference between `map` and `flatMap` is that `flatMap` will not allow the resulting type to be optional, whereas `map` does allow optionals to be returned. So in the previous code block, if we used `map`, the `result` would be `[GroceryItem?]?` instead of `[GroceryItem]?`.

All right, we have our new view model class. It can parse grocery items from a grocery list and it can give us displayable values of a grocery item at a specific `indexPath`. We can now implement our Grocery Items view controller.

Grocery Items View Controller

Let's create a new file called `GroceryItemsViewController.swift`, and we can put this file directly next to the `GroceryItemsViewModel.swift`. Just like our previous view controller, this `UIViewController` subclass will need to conform to three protocols. These protocols are `UITableViewDataSource`, `UITableViewDelegate`, and `ViewModelContainer`:

```
import UIKit

class GroceryItemsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer  {

}
```

Our new view controller will not compile until we have satisfied all of these protocols requirements. Let's start with out `ViewModelContainer` protocol:

```
class GroceryItemsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer  {

    // MARK: - Properties

    var viewModel: GroceryItemsViewModel = GroceryItemsViewModel()
}
```

We have added the `viewModel` property and made sure its type is a `GroceryItemsViewModel`. The next step for the Properties section is going to be a `@IBOutlet` to our table view. Our new outlet looks like the following:

```
class GroceryItemsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer  {

    // MARK: - Properties

    var viewModel: GroceryItemsViewModel = GroceryItemsViewModel()

    @IBOutlet var groceryItemsTableView: UITableView?
}
```

The next step is to wire op our view controller in the storyboard. You have already performed these steps in the previous section. First, we need to set up the identity of the view controller properly. You can do this through the Identity Inspector in the storyboard. Figure 10-15 shows the identity inspector for our `GroceryItemsViewController`.

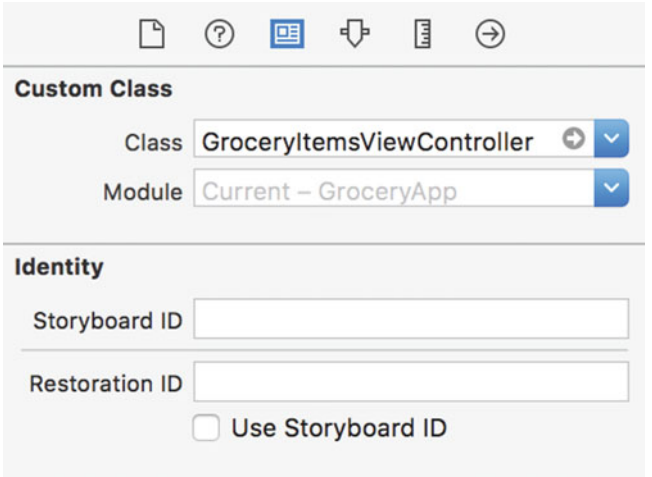


Figure 10-15. *The Identity Inspector for the Grocery Items view controller*

Next, we need to hook up the table view's `dataSource`, `delegate`, and `@IBOutlet` references in the storyboard. Trying not to be too repetitive, we can control-drag from our table view to the view controller reference (the little yellow icon), to set up the `dataSource` and `delegate`. We then want to control-drag from the view controller reference to our table view for our `@IBOutlet`. This should wire everything up correctly, and we should be ready to use the table view now.

Next, let's implement the table view methods we need in our view controller. We are going to use the same three methods our previous view controller used: `numberOfRowsInSection`, `willDisplayCell`, and `cellForRow`. We can start by using hardcoded data:

```
class GroceryItemsViewController: UIViewController, UITableViewDataSource,
```

```
UITableViewDelegate, ViewModelContainer  {

    // MARK: - Properties

    var viewModel: GroceryItemsViewModel = GroceryItemsViewModel()

    // MARK: - Table View

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return 0
    }

    func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt
indexPath: IndexPath) {

    }

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

Now our code should compile, but we still need to fill in these methods correctly. Starting at the bottom, let's create our table view cell:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    return tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
}
```

Back in Chapter 9, we made sure to configure both table view cells with the same reuse identifier, so we can use the same code as the previous controller to dequeue our cell here as well. Moving on, let's use our `groceryItem` method from the view model to set up our table view cell:

```
func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt
indexPath: IndexPath) {
    let item = viewModel.groceryItem(at: indexPath)

    cell.textLabel?.text = item.name
    cell.detailTextLabel?.text = "Quantity: \(item.quantity)"
}
```

We safely grab the grocery item and use it to display the correct text on our cell with the name of the item as well as the item quantity. Lastly, we need to get the count of the items to display. Just like before, we could create an item count method on the view model, or we could directly access the array. Here, I am going to directly access the array:

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

Now, for reference, here is the entire `GroceryItemsViewController.swift` file:

```
import UIKit

class GroceryItemsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer  {

    // MARK: - Properties

    var viewModel: GroceryItemsViewModel = GroceryItemsViewModel()

    @IBOutlet var groceryItemsTableView: UITableView?

    // MARK: - Table View

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return viewModel.groceryItems.count
    }

    func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt
indexPath: IndexPath) {
        let item = viewModel.groceryItem(at: indexPath)

        cell.textLabel?.text = item.name
        cell.detailTextLabel?.text = "Quantity: \(item.quantity)"
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        return tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
    }
}
```

There is one last thing we need to do before we can move on. It might seem small, but it could become unruly quickly. In our `GroceryListViewController` and `GroceryItemsViewController`, we are using the table view cell's reuse identifier. We have hardcoded this string in both places. It might not seem like a big deal, but the more hardcoded strings we use, the more complicated our app becomes to refactor. So, let's takes a break for a minute to create a constants file. This will also help us out later on.

I have called my constants file `GAConstants.swift` for grocery app constants. Let's create this new file at the top of our app above the `AppDelegate.swift` file. This file will contain a struct for all the constant values for the Grocery App. This is not limited to strings; we could also put colors here or other constants data. The following is how I have structured my `GAConstants` struct:

```
/// Grocery App Constants
struct GAConstants {

    /// Constants for our table views
    struct TableCell {
        /// Cell Identifier for our the grocery list and grocery item table views
        static let rightDetail = "Cell"
    }
}
```

We have our outer struct and I have also elected to use an inner struct, so this does not become a mess of data all at the root of the `GAConstants` struct. We know our table view cell has a right detail style, so I have named our constant `rightDetail`. Now, let's go back through our view controllers and use this constant instead of the hardcoded string starting with grocery lists:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    return tableView.dequeueReusableCell(withIdentifier: GAConstants.TableCell.rightDetail,
for: indexPath)
}
```

Now, grocery items:

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    return tableView.dequeueReusableCell(withIdentifier: GAConstants.TableCell.rightDetail,
for: indexPath)
}
```

Awesome! We have removed our magic strings and we can continue forging ahead. Building and running our app now, we should still see our grocery lists, but we are not quite there yet for the grocery items. This leads us to our last section, transferring data. Before we move on, Figure 10-16 shows our progress in the Project Navigator.

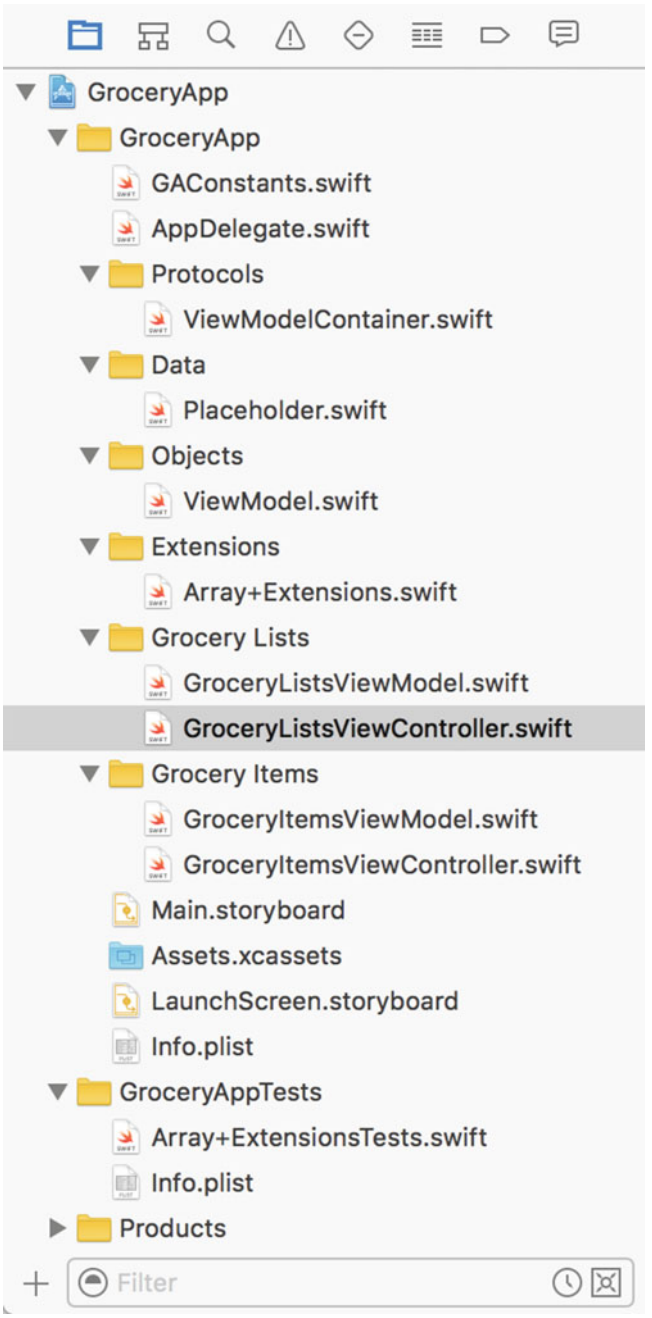


Figure 10-16. Our progress in Xcode's Project Navigator

Transferring Data

We have finally made it the last section for Chapter 10! The infrastructure for our app is almost complete. We have implemented the MVVM pattern and we have wired up our initial view controllers that display our fake data. It might not seem like much, but we have done a lot of work. This section is going to focus on the problem we reached at the end of our last section.

If we run the app, our fake data displays for the grocery lists and not for the grocery items. This is because we are depending on the `GroceryItemsViewModel` to have a reference to the selected grocery list, but we have not set this up. To fix this problem, we are going to need a few different pieces in place. Before we jump right in, let's discuss the solution.

We are going to need a protocol that can allow us to set a grocery list property no matter what type we are dealing with. We can then use this protocol type to set the selected grocery list, which will allow us to give the selected grocery list to the Grocery Items view controller and its view model. Now, if we had used a singleton, in some form, we would not have to worry about this problem. However, we would have different issues to deal with, such as maintainability. Singletons can solve a lot of problems, but they must be treated with care. All right, now that we have an idea of what to do, let's start with our new protocol.

In the Protocols group, let create a new file called `SelectedGroceryListContainer.swift`. If you have not caught on by now, I like to name a lot of my protocols containers. I like naming some protocols this way because I am trying to describe a trait. The purpose of this protocol is to allow us to transfer data through the use of protocols instead of more concrete types. Here is the implementation for this file:

```
protocol SelectedGroceryListContainer {
    var selectedGroceryList: GroceryList? { get set }
}
```

It only requires a property for the selected grocery list. The next step is to conform to this protocol on our `GroceryItemsViewController`:

```
class GroceryItemsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer, SelectedGroceryListContainer {

    // MARK: - Properties

    var selectedGroceryList: GroceryList? {
        get {

        }
        set {

        }
    }

    // ...
}
```

We have our new protocol and the new property. You can see I have used the Swift property getter and setter syntax, but I don't do anything with them yet. We do not want to run into a situation in which the state of our app gets out of whack. So I am not going to save the selected grocery list on this view controller just to transfer it to the view model. What I, later on, we set it to something else without setting the view model's property? This could wreak havoc with the app, but what was wrong would not be obvious. So, let's implement the `get` and `set` to be a passthrough for the view model:

```
class GroceryItemsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer, SelectedGroceryListContainer {

    // MARK: - Properties

    var selectedGroceryList: GroceryList? {
        get {
            return viewModel.groceryList
        }
        set {
            viewModel.groceryList = newValue
        }
    }

    // ...
}
```

Pretty cool, now all that is left is to use this protocol and property to transfer the selected grocery list. Let's flip over to the `GroceryListsViewController` to accomplish this. To intercept the view controller that will be presented, we are going to override the `prepareForSegue` method on the `GroceryListsViewController`:

```
class GroceryListsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer {

    // MARK: - Properties

    // ...

    // MARK: - Segue

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        super.prepare(for: segue, sender: sender)
    }

    // MARK: - Table View

    // ...
}
```

You might think we are going to ask the segue for the GroceryItemsViewController, but we want to use our protocol to make this more generic. Let's see if the destination on the segue is our protocol type:

```
// MARK: - Segue

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    super.prepare(for: segue, sender: sender)

    if var groceryListContainer = segue.destination as? SelectedGroceryListContainer {
        // TODO: transfer grocery list
    }
}
```

We are only missing the last piece, which is the capability to transfer the selected grocery list. This is where we want to give more responsibility to our view model. Our view model holds our data; so let's make it transfer it as well. So we will need to add a new method to the GroceryListsViewModel. Keeping the API design guidelines in mind, we want this to read as a verb phrase since there will be side effects. Let's create our transfer method on GroceryListsViewModel:

```
class GroceryListsViewModel: ViewModel {

    // MARK: - Properties

    // ...

    // MARK: - Transfering Data

    func transferGroceryList(at indexPath: IndexPath, to container: inout
SelectedGroceryListContainer) {
        container.selectedGroceryList = groceryLists.value(at: indexPath.row)
    }

    // MARK: - Grocery List Data

    // ...
}
```

This method reads as a verb phrase exactly as it should. We then expect an indexPath for the selected grocery list and a container to transfer the grocery list to. I have also elected to not make the SelectedGroceryListContainer protocol a class protocol. This way, we need to explicitly say our property here is an inout property.

The inout keyword allows us to transfer the property into the method so it can be modified. The inout property is then transferred back to the caller with the potentially modified state. Since protocols are seen as value types regardless of the underlying implementation, we need this to set our property. If we go back to our GroceryListsViewController, let's finish this. So what's left to do? We need to use our table view property to access the selected index path, and we need to use our view model to transfer the selected grocery list. The prepareForSegue method should now look like the following code block:

```
// MARK: - Segue

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    super.prepare(for: segue, sender: sender)

    if var groceryListContainer = segue.destination as? SelectedGroceryListContainer,
        let indexPath = groceryListTableView?.indexPathForSelectedRow {

            viewModel.transferGroceryList(at: indexPath, to: &groceryListContainer)
    }
}
```

Now, drum roll please. If you run your app, you should still see the fake data on the first view controller, but if you select one of those lists, what happens? It now displays the items within the list! You should see the two screens from Figure 10-17.

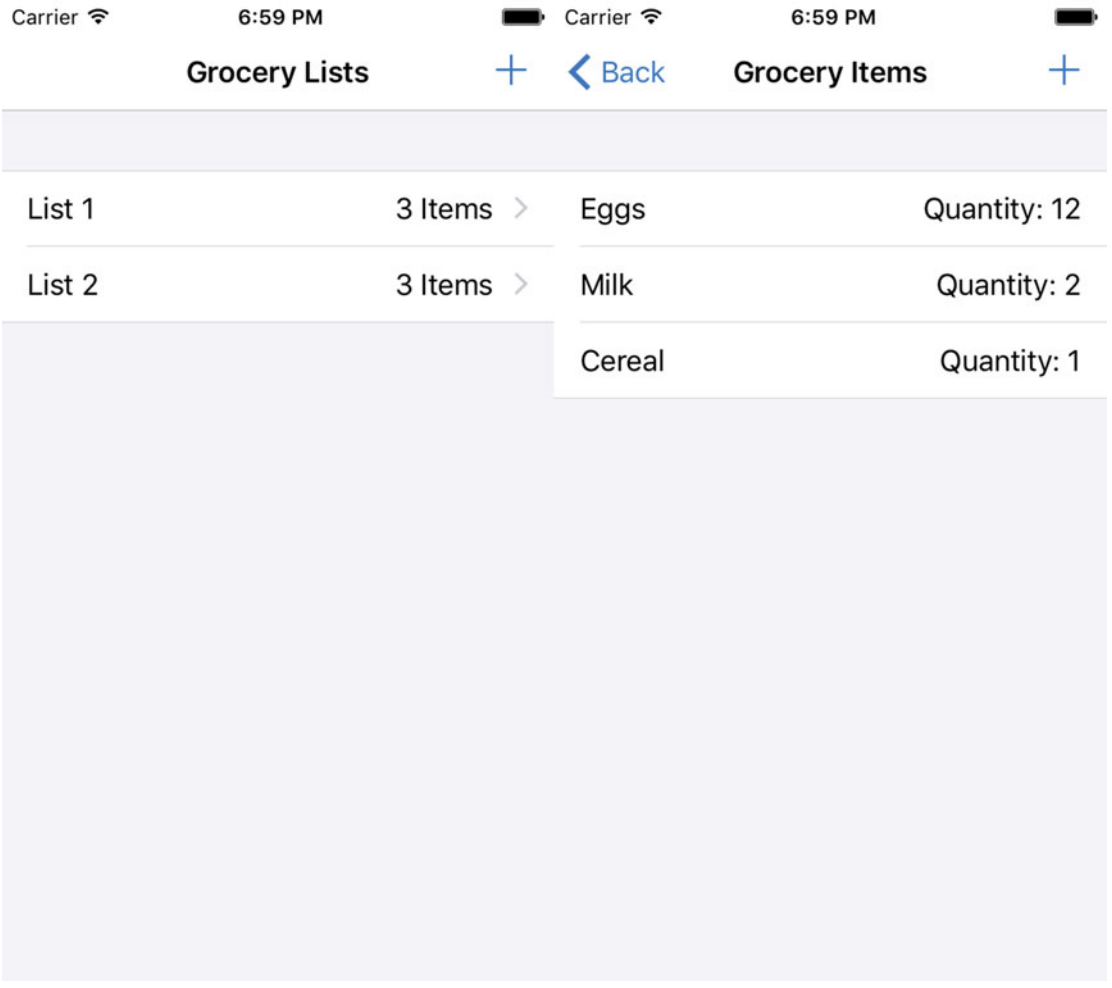


Figure 10-17. *The Grocery App running with fake data. The first two view controllers are set up to display fake data*

That does it for this chapter. I know there was a lot here, but we got through it. There is a lot to this simple app, especially with the architecture we have chosen, but I hope you can see how clear and concise everything we have written is. There are clear responsibilities for every piece, and there is no code in our app that was done without thought and reason. Figure 10-18 shows the progress of our app in the Project Navigator.

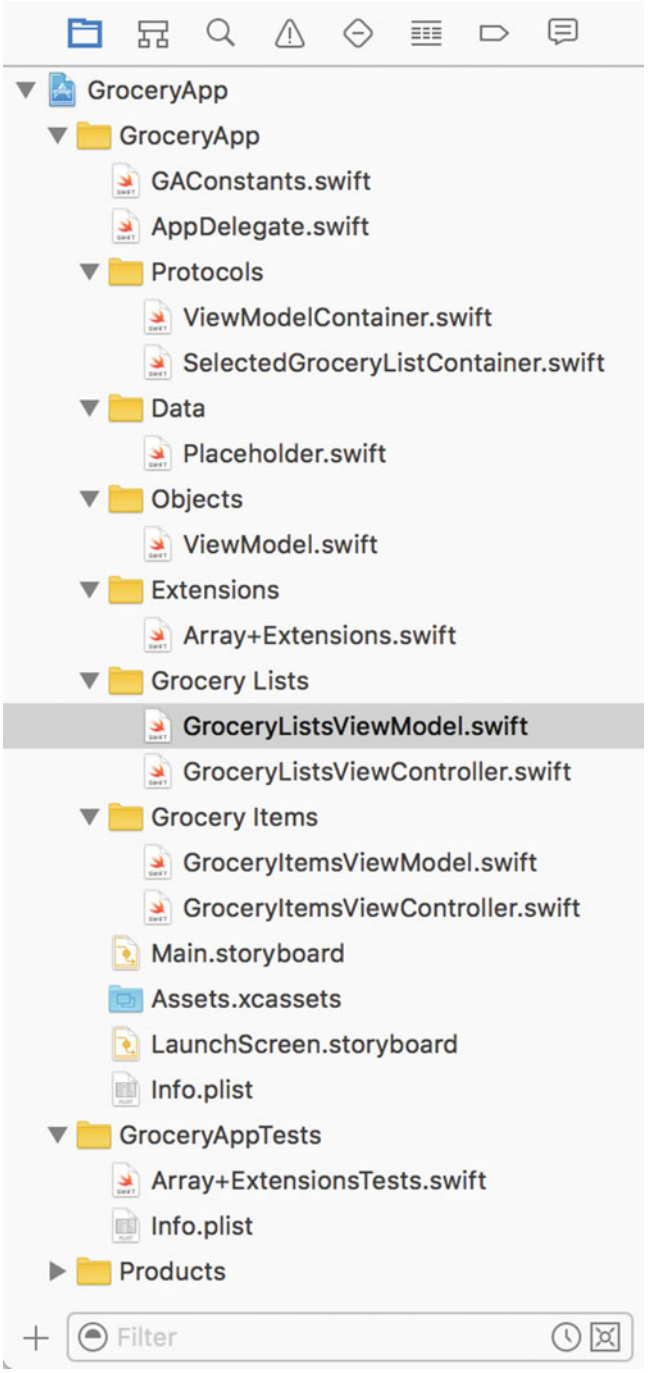


Figure 10-18. *The progress of the Grocery App in Xcode's Project Navigator*

We are a little light in the tests and that is mainly due to our fake data. I did not want to walk you through testing that would just have to be redone later. Don't worry; we are going to address this in the next chapter. Chapter 11 is going to be all about Core Data and integrating real data with our app.

Wrap Up

Wow! This has been a huge chapter. A lot happened here and I would recommend running the app a couple of times before moving on. You can throw some breakpoints in and walk through the code if you want. So, let's wrap this up. We started this chapter with almost a blank slate. We had the interface built, but everything else was brand new. We saw how to implement the MVVM pattern in a real-world scenario. This is one of the key points to this chapter. We made sure to use dependency injection to allow our view models to access the data for our app. By using dependency injection, we can, theoretically, use anything to hold onto our data. We could use some other object, a singleton, or anything we wanted. We are only restricted to the protocol we created.

Another key point to this chapter is how we worked with optionals. If you go back through our code, you can see how we are accounting for optionals in the appropriate locations. Our view model holds onto the data through optional properties and then makes sure we safely access that data. We then make sure the interface between the view model and view controller is clean. The view controller can just use the APIs of the view model without deciding what to do if something is optional. I cannot describe how important it is to account for optionals properly. This does not mean everything should expect optionals, but we should allow for optionals until it is absolutely necessary.

Finally, we used protocols to keep concrete types, such as the `SelectedGroceryListContainer` protocol, out of the equation. If we add more view controllers that conform to this protocol, we do not have to refactor our code to transfer the selected grocery list. We have accomplished a lot, but there is still a ways to go. Let's move on to the next chapter where we will integrate Core Data.

Chapter 11

Grocery App: Core Data

In the previous chapter, we implemented the MVVM design pattern and saw our app displaying fake data. This chapter is going to remove our fake data and integrate Core Data. I wanted to handle this before the final chapter. Chapter 12 is going to finish the functionality of our app by implementing the rest of the functionality, including creating grocery lists/items, reloading the view, and displaying errors to the user.

Through most of this chapter, our app is not going to be able to build. This will be due to the switch we are going to pull between our fake data and Core Data. Don't worry, though, because once we are done, we will be able to see our app using real data and persisting said data. Are you ready for the biggest switch since *Raiders of the Lost Ark*? (Didn't that now work? ☺)

What You'll Learn

This chapter is going to cover integrating Core Data. *Core Data* is Apple's framework for handling persistence within apps. Core Data is huge, so we are not going to discuss everything it can do for us in this chapter. I am going to highlight key points, but as always, I recommend you research Core Data more. We also have to think slightly differently because we are using MVVM. Remember, Core Data and the entire iOS stack are built on the idea of MVC. We have to consider compatibility with the features Apple gives us.

After you are done with this chapter, you should feel comfortable with both using Core Data in your applications and with your knowledge of how to set up and manage the Core Data stack. Xcode has a great interface for managing the data model, but it might not seem straightforward if this is your first experience with Core Data. After integrating Core Data, we are going to add functionality to our view model layer. This functionality will come into play in Chapter 12. Finally, we will finish this chapter by writing tests for our core functionality. It's the penultimate chapter; let's get to it.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_11

Persistent Container

Before we build our data model, let's make sure our app is set up to use Core Data. Way back in Chapter 9, during "Project Setup," we made sure to turn on Core Data. Just like with the unit tests, I elected to leave this setting turned off so I could show you how to integrate Core Data fresh. If you forgot to check this option during project setup, then you can follow along. Let's open the `AppDelegate.swift` file and look at the bottom. You should see the following code:

```
// MARK: - Core Data stack

lazy var persistentContainer: NSPersistentContainer = {
    let container = NSPersistentContainer(name: "GroceryApp")
    container.loadPersistentStores(completionHandler: { (storeDescription, error) in
        if let error = error as NSError? {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()

// MARK: - Core Data Saving support

func saveContext () {
    let context = persistentContainer.viewContext
    if context.hasChanges {
        do {
            try context.save()
        } catch {
            let nserror = error as NSError
            fatalError("Unresolved error \(nserror), \(nserror.userInfo)")
        }
    }
}
```

> **Caution** If you do not have this code at the bottom of your `AppDelegate.swift` file, make sure to add it. Also be sure to import Core Data at the top of the file. This is what drives the Core Data stack in our app, so we have to implement this code.

If you chose Core Data, then your `AppDelegate` will have a lot of comments in this code. I have omitted the comments for clarity. There are two pieces to this code; let's go through each. The first is this `persistentContainer`. It might look scary, but it is fairly simple. It is a lazily initialized variable that, once accessed, creates the app's persistent stores. Once accessed, this will load the app's data into memory. The second piece is this `saveContext` method. Again, all this method does is write out the current context.

You'll notice this `saveContext` method does not have to deal with the specific changes to save. One of the niceties of Core Data is that everything is managed through the idea of a

context. This context then just writes out the current context when instructed. That way, there is no need to say, “Save this new grocery list I just added.” You can just say, “Save whatever you got.”

You might notice that there are `fatalErrors` in this code. This is what the Core Data template provides us with, but the omitted comments say the error should be handled appropriately and that it is not a fatal error. It even goes so far as to say the fatal errors should not be included in a shipping application. We are not going to change this functionality for our app. If you choose to continue development and eventually ship this app to the App Store, make sure these fatal errors are resolved first.

The last thing you should take notice of is the `applicationWillTerminate` method. If you used the template, there should be a call to `saveContext` like the following:

```
func applicationWillTerminate(_ application: UIApplication) {
    // Called when the application is about to terminate. Save data if appropriate. See also
applicationDidEnterBackground:.
    saveContext()
}
```

Every time we go to terminate the app, the current context will be written out. We are not going to rely on this functionality. We are going to handle saving ourselves when we modify or add new data, but this can stay here as a catch all just in case.

All right, we now have the necessary code in our `AppDelegate` to move on to some new code. To start integrating Core Data, we need to remove our placeholder data. This means our app is not going to build for a little while. It is important to follow along closely, since we will not be capable of running the app. Before we start this process, let’s create a new file called `PersistentContainer.swift` and write a new protocol called `PersistentContainer`. Be sure to put this file in the Protocols group. Let’s write a protocol that looks like the following code block:

```
import CoreData


protocol PersistentContainer {
    var persistentContainer: NSPersistentContainer { get set }
    var managedObjectContext: NSManagedObjectContext { get }
}
```

This protocol is meant to replace the `PlaceholderDataContainer` protocol in the `AppDelegate`. The `persistentContainer` property is already on the `AppDelegate`. The `managedObjectContext` is the current context of the persistent store. Before we add this, let’s add an extension for this protocol:

```
extension PersistentContainer {
    var managedObjectContext: NSManagedObjectContext {
        return persistentContainer.viewContext
    }
}
```

This extension just gives us a little extra sugar for accessing the managed object context. We do not actually need this property because the `persistentContainer` has a reference

called `viewContext`. We're using this instead to save us from typing all that. Now, let's start the switch. Remember, this means our app is not going to compile for a while, but we will work through it. The first step is to delete the `Placeholder.swift` file. Figure 11-1 shows the Project Navigator after deleting this file.

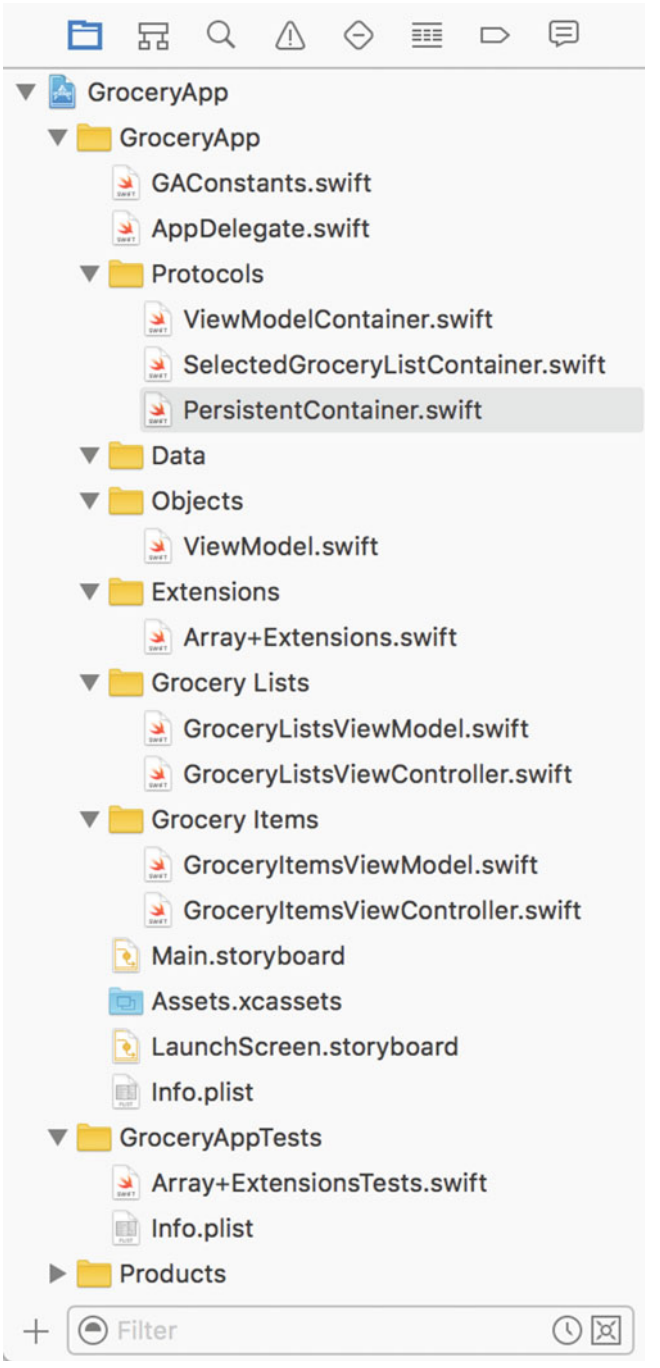


Figure 11-1. *The app's progress in Xcode's Project Navigator after deleting the Placeholder.swift file*

All right, now let's remove our old PlaceholderDataContainer protocol from the AppDelegate. We can also add our new PersistentContainer protocol at the same time. The previous AppDelegate looked like this:

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, PlaceholderDataContainer {

    var window: UIWindow?
    var container: DataContainer = DataContainer()

    // ...
}
```

Now, our new AppDelegate should like the following:

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, PersistentContainer {

    var window: UIWindow?

    // ...
}
```

We have removed our DataContainer property and the AppDelegate now conforms to our new protocol. Since we already have a persistentContainer property on the AppDelegate, there should be no more work required. The next step is to use this new protocol in our view model just like the PlaceholderDataContainer protocol was used. Navigate to the ViewModel.swift file. Here is our current implementation:

```
class ViewModel {

    // MARK: - Properties

    var placeholderContainer: PlaceholderDataContainer?
    var dataContainer: DataContainer? {
        return placeholderContainer?.container
    }

    // MARK: - Initializer

    init(placeholderContainer: PlaceholderDataContainer? = UIApplication.shared.delegate as?
    AppDelegate) {
        self.placeholderContainer = placeholderContainer
    }
}
```

This is basically all wrong now. Let's make this work with our new protocol. There are three pieces of functionality we need to add. First, we need to hold a reference to the object conforming to the PersistentContainer protocol. Second, we need to allow the conforming object to be injected using the AppDelegate as our default. Lastly, we need to add another computed property for the managedObjectContext. This is just another convenience property that is not required. Here is the entire ViewModel class after all of our updates:

> **Caution** Be sure to import `CoreData` at the top of the `ViewModel.swift` file.

```
import UIKit
import CoreData

class ViewModel {

    // MARK: - Properties

    var persistentContainer: PersistentContainer?
    var managedObjectContext: NSManagedObjectContext? {
        return persistentContainer?.managedObjectContext
    }

    // MARK: - Initializer

    init(persistentContainer: PersistentContainer? = UIApplication.shared.delegate as?
    AppDelegate) {
        self.persistentContainer = persistentContainer
    }
}
```

Not much has really changed. We are just using our new protocol the same way we used the old protocol. We are still using the `AppDelegate` as our default. Let's take a step back because I want to discuss why we are injecting this instead of an initializer that looks like the following:

```
// Do not implement this!
init() {
    self.persistentContainer = UIApplication.shared.delegate as? AppDelegate
}
```

As I have stated previously, iOS is designed with MVC in mind. This means that the Core Data stack is meant to be used in view controllers. However, as we discussed in Chapter 4, we do not want to bloat our view controllers with too much responsibility. Core Data is a huge responsibility. On many occasions I myself have created singletons and other manager type classes to hold the responsibility of Core Data so my view controller does not have to.

However, right in Apple's documentation, they discuss how managed object contexts should be injected into view controllers. This documentation can be found at https://developer.apple.com/library/content/documentation/DataManagement/Conceptual/CoreDataSnippets/Articles/stack.html. It also states the `AppDelegate` should be responsible for the first injection point, but after that, the managed object context should be passed around from controller to controller.

If we create a singleton or manager to hold on to this responsibility, we can miss out on this functionality by having only one point for Core Data. We then tie our implementation to this single context and single implementation of the Core Data stack. After reading through the documentation, it also sounds like we have violated iOS patterns through our use of MVVM. I don't believe this is the case. iOS is allows going to be MVC by nature. We cannot avoid this. However, I see the view model as an extension of the view controller.

In true MVVM, there is no idea of a controller. The view model actually takes the place of the view controller. Since we cannot avoid the view controller in iOS, the view model is just half of the view controller. The view model half is responsible for communicating with the data. The view controller half is meant to communicate with the view. Putting these two pieces together, we achieve the same result we would by having our view controller access Core Data. Except with this separation, we have defined clear lines for injecting more functionality and higher testability.

The preceding theory is why I am okay allowing the view controller to access members such as the grocery lists and the selected grocery list property. These two halves are separated but cannot exist without each other. All right, that was a long explanation, so what was the point? We want to use dependency injection when working with the Core Data stack so we can allow for other contexts to be injected. This also makes our view model layer extensible and testable. Our default is the `AppDelegate`, but we could just as easily inject another object that gives us different data and a different context, but our view model layer, which work exactly the same.

So what have we done in this section? We have started the switch to Core Data from our fake data. Our app is currently not compiling, but never fear—we are going to fix this. We created a new protocol, the `PersistentContainer` protocol, to hold onto the `NSPersistentContainer` property and give us access to the `managedObjectContext`. We then used dependency injection in our view model to inject this new protocol type with the default set to our `AppDelegate`. Then finally, I hope I have cleared up any confusion you might have about using the MVVM pattern since iOS is by nature MVC.

XCDataModel

We get to take a little break from coding in this section. We will set up our data model file and create our two entities `GroceryList` and `GroceryItem`. We will then use Xcode's Core Data code generation to create our real models. Let's start with the `GroceryApp.xcdatamodel` file. Since I did not choose the Core Data option when I created my project, I do not have this file. Let's start here.

If you do not have the data model file, go to File ➤ New ➤ File. This will open the new file dialog, where we want to search for *Data Model*. Figure 11-2 shows the new file chooser.

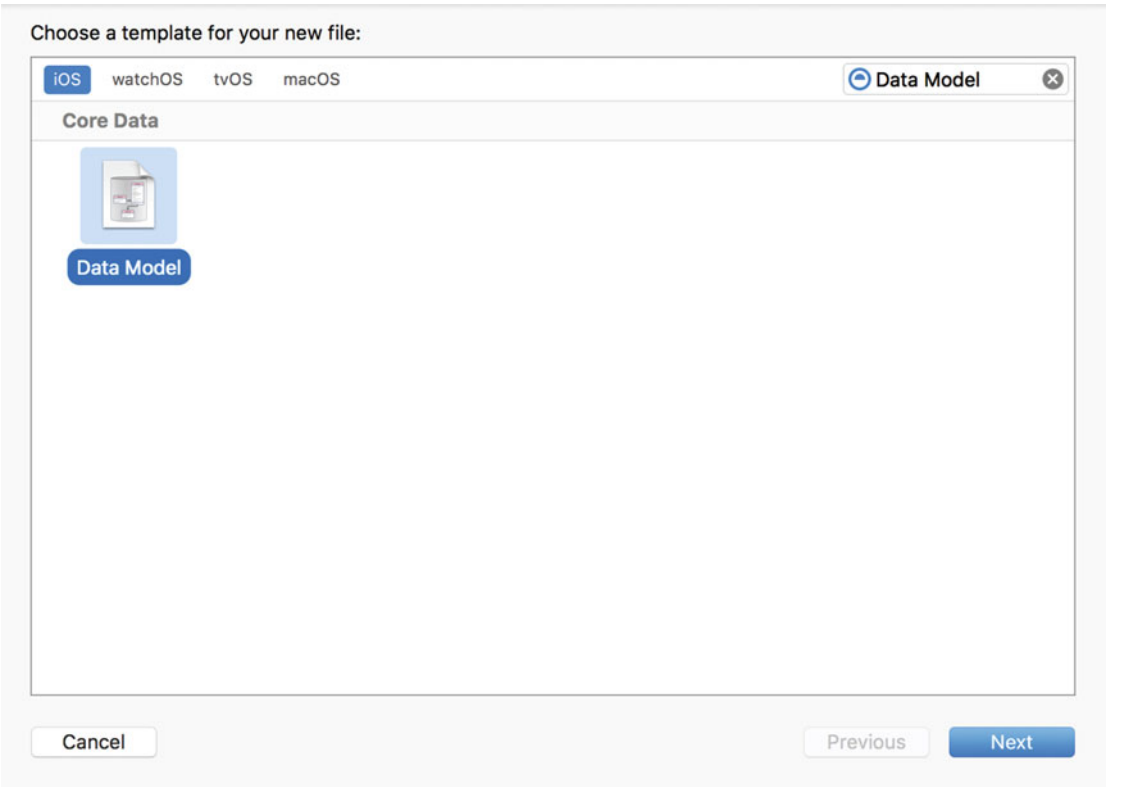


Figure 11-2. The new file chooser with a search for Data Model

Let's call this new file `GroceryApp` and make sure its target membership is the GroceryApp target and not our tests. Once you create this file, open it and you will see Figure 11-3. This is the default screen for a Core Data model file. It shows us a list of our entities.

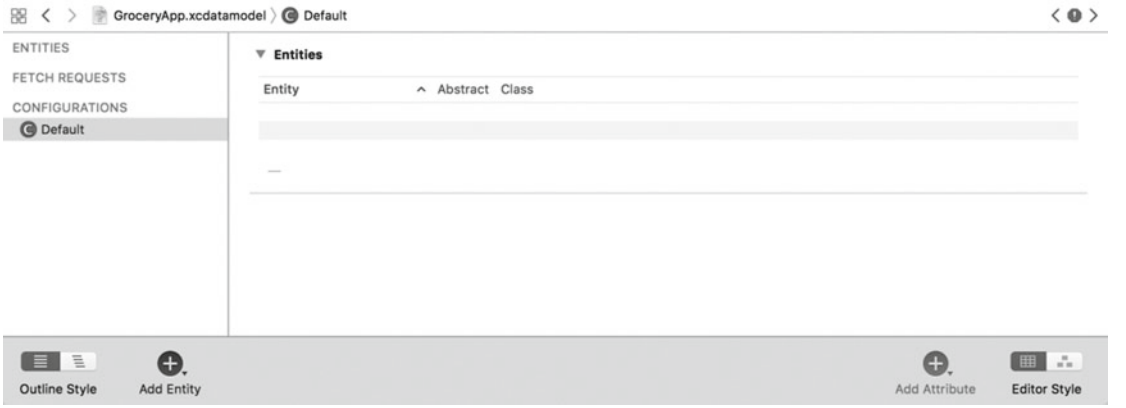


Figure 11-3. The GroceryApp.xcdatamodel file. This screen will contain all of the entities for our app.

Whether you just created this file or you already had it in your project, it's time to get to work. If you remember our placeholder data, we had those tuples representing our models. We want to match those tuples' properties and types. The following code block is our old tuples for reference:

```
typealias GroceryList = (name: String, items: NSSet)
typealias GroceryItem = (name: String, quantity: Int16)
```

Let's create our first entity and call it `GroceryList`. This entity is going to contain one attribute and one relationship. Let's add the attribute to the `GroceryList` called `name` and make it of type `String`. Figure 11-4 shows our new attribute along with its type.

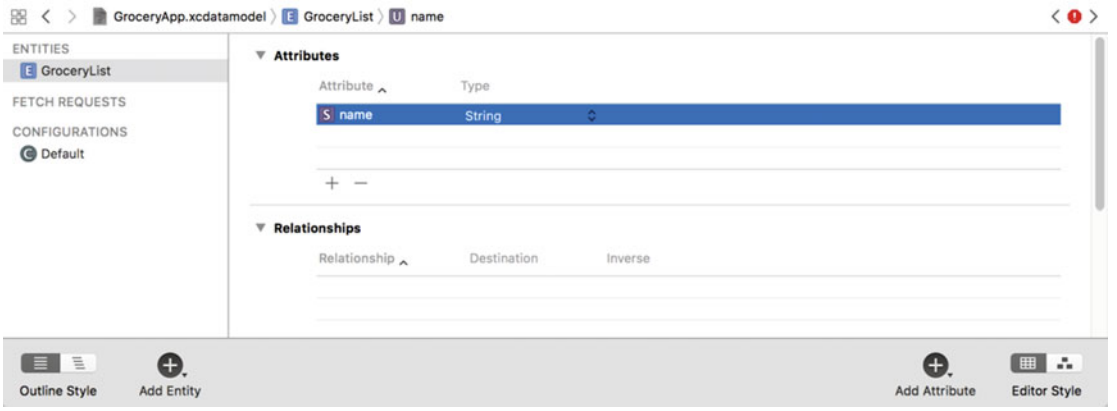


Figure 11-4. *The name attribute on our GroceryList entity with a type of String*

Now before we add our relationship to property, let's add our second entity, `GroceryItem`. This entity is going to have two attributes, a name and a quantity, and a relationship property called `list`. The name attribute should be a `String` type and the quantity attribute should be an `Int16` type. These attributes match our tuple from the previous code block. The *Relationship* property should have a destination of `GroceryList` and currently *No Inverse*. Figure 11-5 shows the fully configured `GroceryItem` entity.

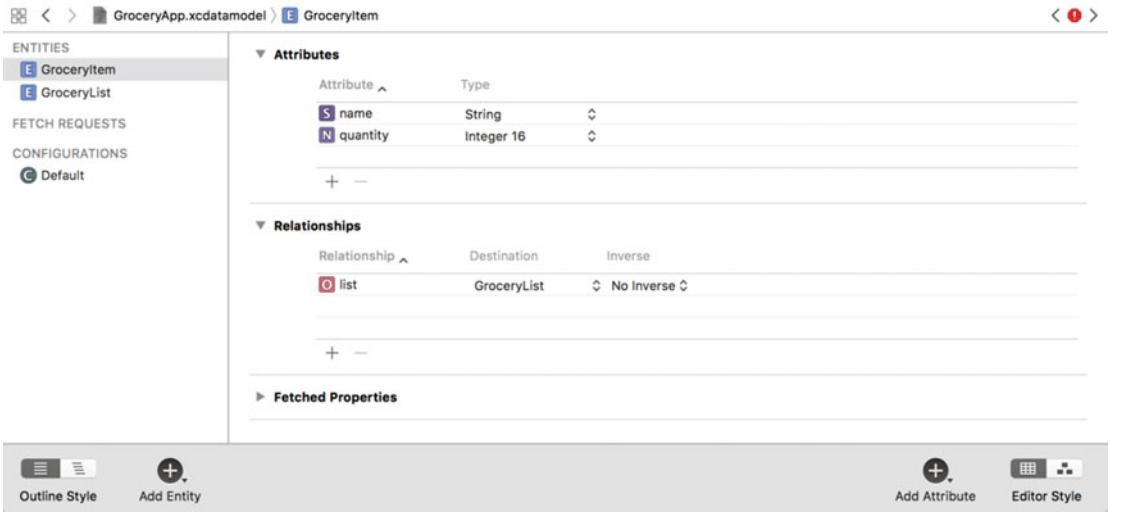


Figure 11-5. The GroceryItem entity with a name attribute, a quantity attribute, and a list relationship

Now, let's get back to our `GroceryList` entity. We need a relationship property that points to our `GroceryItem` entity. Let's define a new relationship on the `GroceryList` entity called `items`. We want to configure our `items` relationship to have a *Destination* of `GroceryItem` and an *Inverse* of `list`. You can see this configuration in Figure 11-6.

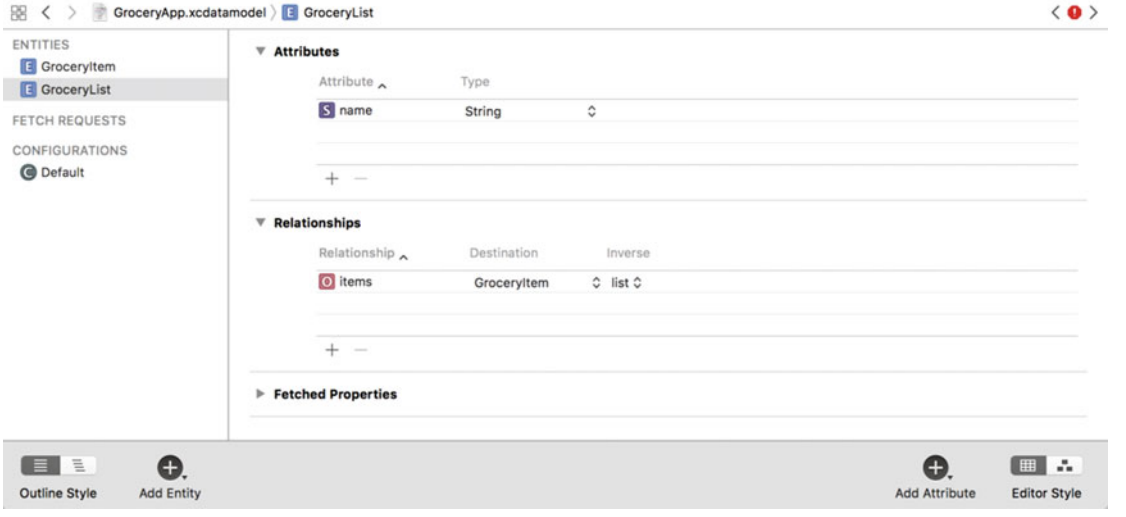


Figure 11-6. The new configuration for the items relationship on the GroceryList entity. The Destination should be GroceryItem with an Inverse of list

The next step is to set up the type of relationship for the items on `GroceryList`. Open up the Utility Pane on the right side of Xcode and navigate to the Data Model Inspector. You will see multiple properties, but let's focus on the *Type*. Right now, it is configured to be *To One*. Let's change this to be configured to be *To Many*. This configuration can be seen in Figure 11-7. By configuring the items to be To Many, we can have a set of items per grocery list. The list relationship on `GroceryItem` should remain To One. That way, each grocery item only corresponds with one grocery list.

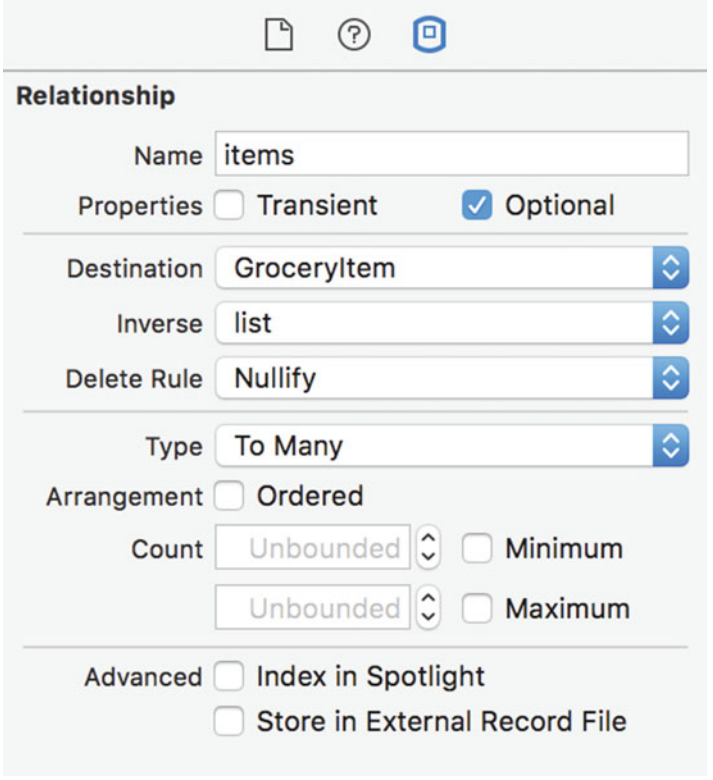


Figure 11-7. *The Data Model Inspector for the items relationship property on GroceryList*

All right, we should now have a fully configured data model for our app. We have a `GroceryList` entity with a name attribute and a relationship property. This relationship describes the connection with the `GroceryItem` entity. Each grocery item has a name attribute and a quantity attribute. We also made sure these entities match our previously defined tuples that acted as the fake data.

We are now going to let Xcode generate our model files for these entities. To generate your model, make sure you have the data model file selected, then go to Editor ➤ Create NSManagedObject Subclass. You can see this menu option in Figure 11-8.

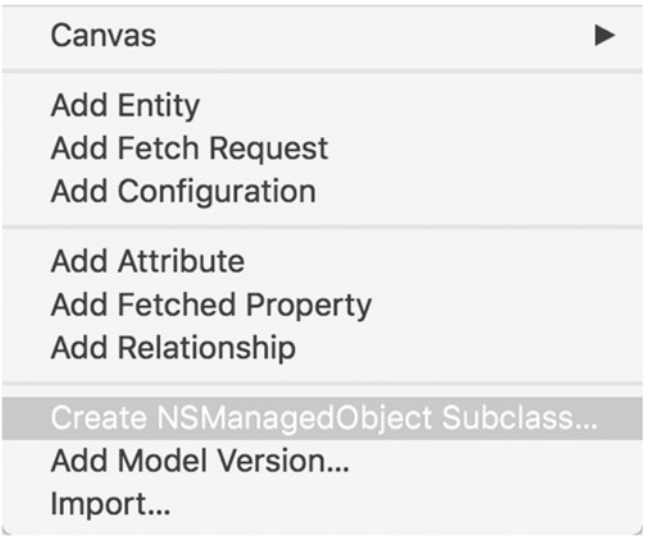


Figure 11-8. *The Editor menu and how to create NSManagedObject subclasses*

This will open a menu to select the data model you want to manage. The only option here should be our `GroceryApp` data model. Click *Next* and then we will see another menu for selecting the individual entities to manage; this menu can be seen in Figure 11-9. We want to make sure we have both entities, `GroceryList` and `GroceryItem`, selected.

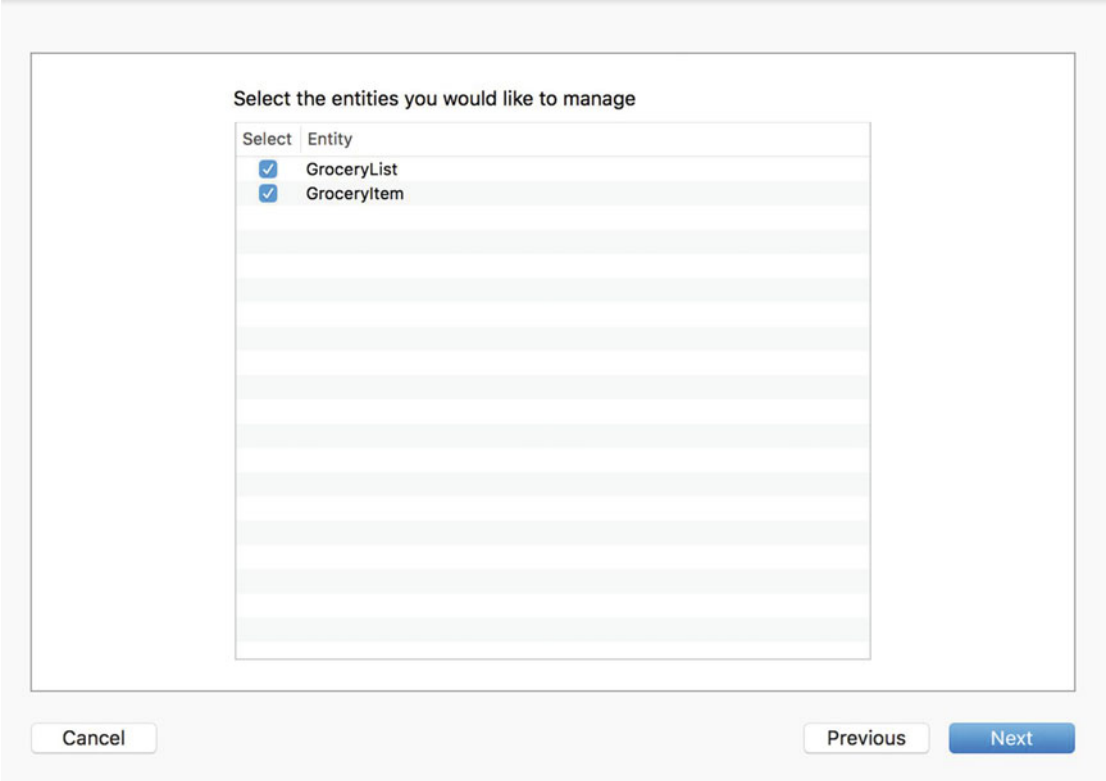


Figure 11-9. *The menu for selecting the entities to manage via NSManagedObject subclasses*

After selecting *Next* in Figure 11-9, you will have to choose where to save the new files and Xcode will create four new files. Let's move these files to our Data group. These are the four files that should have been created:

1. `GroceryList+CoreDataClass.swift`
2. `GroceryList+CoreDataProperties.swift`
3. `GroceryItem+CoreDataClass.swift`
4. `Groceryitem+CoreDataProperties.swift`

The following code block shows the code for the first two files. The Core Data class file defines the actual class with no properties or methods:

```
import Foundation
import CoreData

public class GroceryList: NSManagedObject {

}
```

The properties file has two extensions describing the properties and the Core Data methods:

```
import Foundation
import CoreData

extension GroceryList {

    @nonobjc public class func fetchRequest() -> NSFetchRequest<GroceryList> {
        return NSFetchRequest<GroceryList>(entityName: "GroceryList");
    }

    @NSManaged public var name: String?
    @NSManaged public var items: NSSet?

}

// MARK: Generated accessors for items
extension GroceryList {

    @objc(addItemsObject:)
    @NSManaged public func addToItems(_ value: GroceryItem)

    @objc(removeItemsObject:)
    @NSManaged public func removeFromItems(_ value: GroceryItem)

    @objc(addItems:)
    @NSManaged public func addToItems(_ values: NSSet)

    @objc(removeItems:)
    @NSManaged public func removeFromItems(_ values: NSSet)

}
```

There is a lot of code here, so take a minute to look through it. We have our two attributes, name and items, in the first extension. The second extension defines methods that allow us to add and remove items from the grocery list. These methods look a bit weird since they have no implementation, but they are marked with an `@NSManaged`. These methods work because Core Data automatically generates the code for these actions and our extension methods using Key-Value coding. These methods then use the underlying implementation provided by Core Data. Let's see the two files for the `GroceryItem`:

```
import Foundation
import CoreData

public class GroceryItem: NSManagedObject {

}
```

Then we have the extension for the `GroceryItem` class:

```
import Foundation
import CoreData

extension GroceryItem {

    @nonobjc public class func fetchRequest() -> NSFetchRequest<GroceryItem> {
        return NSFetchRequest<GroceryItem>(entityName: "GroceryItem");
    }

    @NSManaged public var name: String?
    @NSManaged public var quantity: Int16
    @NSManaged public var list: GroceryList?

}
```

This is simpler than the previous implementation, but it uses the same `@NSManaged` attribute to define the three attributes here. You also might have noticed the `fetchRequest` method on both extensions. This is the method we are going to use to fetch the grocery lists from Core Data. This will be in our next section.

All right, our code still does not compile, but we are almost there. This section has set up our data model for the Grocery App. We have created two entities representing a grocery list and a grocery item. We then have Xcode generate our models for us, so we can access Core Data through our model classes. The next section is going to expand on our base view model for a more comprehensive Core Data API and fixing our build issues. Before we move on, take a look at Figure 11-10, which shows our progress in Xcode's Project Navigator.

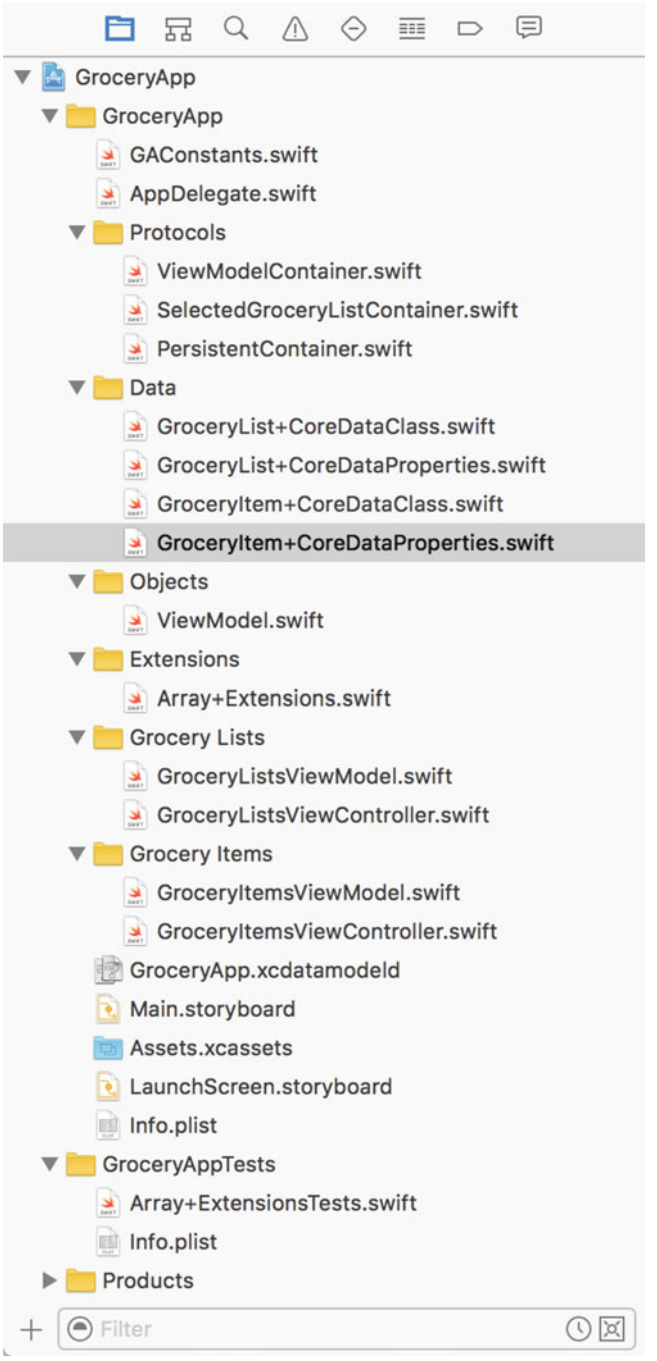


Figure 11-10. Our progress in Xcode's Project Navigator

View Model and Core Data

We have our new data models, but our view `GroceryListsViewModel` is still using the old `dataContainer` property. Once we fix this issue, our app should start compiling again. The following code block is the broken code:

```
class GroceryListsViewModel: ViewModel {

    // MARK: - Properties

    var groceryLists: [GroceryList] {
        return dataContainer?.data ?? []
    }

    // MARK: - Transfering Data

    // ...

    // MARK: - Grocery List Data

    // ...
}
```

Let's remove the reference to the `dataContainer` and allow our app to compile again. Make sure the `groceryLists` computed property looks like the following:

```
var groceryLists: [GroceryList] {
    return []
}
```

This is not what we want, but it will allow us to build the new Core Data APIs we need to access our actual data. For now, we can leave this as an empty array.

So, now that we are compiling again, let's focus on the `ViewModel` base class. We need two new methods to work with Core Data. Let's start with the `executeRequest` method.

In Chapter 6, we discussed Swift generics. We need our `executeRequest` method to use the `fetchRequest` object returned by our data models to get our data from Core Data. However, we have two different types; so let's make this a generic method so it will work for any Core Data model:

```
func executeRequest<Type: NSManagedObject>() throws -> [Type]? {
    let request = Type.fetchRequest()
    return try request.execute() as? [Type]
}
```

We have our new method. It is generic and we constrain the `Type`, so it must inherit from `NSManagedObject`, like our `GroceryList` and `GroceryItem`. We then get the `NSFetchRequest` and call `execute`. The `execute` method can throw an error, so we want to mark our method as `throws` and allow the user of this method to determine how to handle the error.

There is one method left to implement. The NSFetchRequest object expects to be run within a managed object context. That means the previous code block will not work just anywhere. So, let's create a new fetch method on our base ViewModel class. The signature of this method should be identical to the executeRequest:

```
func fetch<Type: NSManagedObject>() -> [Type]? {
    var result: [Type]? = nil

    managedObjectContext?.performAndWait { [weak self] in
        do {
            result = try self?.executeRequest()
        }
        catch {
            print(error)
        }
    }
    return result
}
```

We have our second generic method called fetch. This method uses the managedObjectContext on the ViewModel class and calls performAndWait. This method synchronously executes the closure within the context of the managed object context. This allows our NSFetchRequest to execute properly.

You can see we have accounted for the error here with a do catch block. In our case, we are ignoring the error. If this were a production app, however, it would probably be a good idea to have this display something to the user. The last important piece to notice is that we are capturing self as weak in our closure. In Chapter 2, while discussing the Memory Graph Debugger, we talked about the importance of using weak references with closures to make sure we do not retain memory. Odds are, this would not leak memory, but it is still good practice to use weak references. Here is the entire ViewModel base class for reference:

```
class ViewModel {

    // MARK: - Properties

    var persistentContainer: PersistentContainer?
    var managedObjectContext: NSManagedObjectContext? {
        return persistentContainer?.managedObjectContext
    }

    // MARK: - Initializer

    init(persistentContainer: PersistentContainer? = UIApplication.shared.delegate as?
    AppDelegate) {
        self.persistentContainer = persistentContainer
    }

    // MARK: - Core Data

    func fetch<Type: NSManagedObject>() -> [Type]? {
```

```
        var result: [Type]? = nil

        managedObjectContext?.performAndWait { [weak self] in
            do {
                result = try self?.executeRequest()
            }
            catch {
                print(error)
            }
        }
        return result
    }

    func executeRequest<Type: NSManagedObject>() throws -> [Type]? {
        let request = Type.fetchRequest()
        return try request.execute() as? [Type]
    }
}
```

Now, let's flip back to our `GroceryListsViewModel` and fix the empty array. It should currently look like the following code:

```
class GroceryListsViewModel: ViewModel {

    // MARK: - Properties

    var groceryLists: [GroceryList] {
        return []
    }

    // ...
}
```

Now, we can call `fetch` and make sure we still return an empty array if the result is `nil`:

```
class GroceryListsViewModel: ViewModel {

    // MARK: - Properties

    var groceryLists: [GroceryList] {
        return fetch() ?? []
    }

    // ...
}
```

This is the cool part about Swift generics. The value of our computed property is enough information to tell the `fetch` method what type we are attempting to retrieve. This will use the grocery list's `fetchRequest` method on the Core Data class. Now, we can build and run our app, but we won't see anything. We have no data saved in Core Data to retrieve. Let's manually add a grocery list so we can see our app work with Core Data. The following code block in the application did finish launching method on our `AppDelegate`:

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

```
    if let entity = NSEntityDescription.entity(forEntityName: "GroceryList", in:
    persistentContainer.viewContext) {
        let list = GroceryList(entity: entity, insertInto: persistentContainer.viewContext)
        list.name = "Real List #1"
        saveContext()
    }

    return true
}
```

Here we created a reference to the `GroceryList` entity and used the entity and the managed object context on the `persistentContainer` to create a new `GroceryList` object. We then set the name to `Real List #1` and called `saveContext`. Now, when you build and run the app, you should see Figure 11-11. This confirms that our code is still working and we have successfully integrated Core Data! Be sure to remove this code after running the app once. If you rerun the app without this code, however, your list should still be there because it is being persisted.

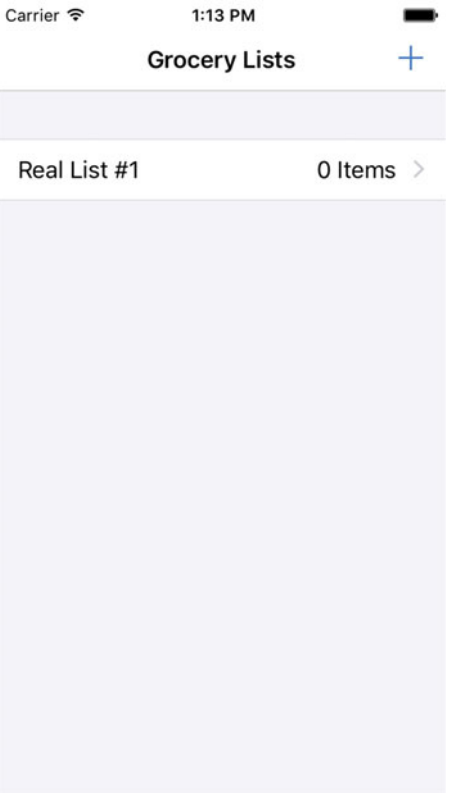


Figure 11-11. *Our app running after we configure Core Data correctly*

We have successfully integrated Core Data and we have confirmed our app is running and talking with Core Data correctly. In the next section, we are going to write tests for the code in our base `ViewModel` class. The functionality we gave to our base view model is pretty core to how we interact with Core Data, so we should make sure we have it covered in unit tests.

View Model Tests

This section is going to add unit tests for our base view model class. We are going to pull concepts and code from Chapter 8 to get through this section. Currently the base view model class consists of an initializer that allows a `PersistentContainer` to be injected and two methods for executing an `NSFetchRequest`. Let's start with our `executeRequest` method. Here is the code for reference:

```
func executeRequest<Type: NSManagedObject>() throws -> [Type]? {
    let request = Type.fetchRequest()
    return try request.execute() as? [Type]
}
```

This is a fairly simple method. It instantiates an `NSFetchRequest` and then calls the `execute` method. The problem here is that the `fetchRequest` method on our `Type` is a class method. We have no way to test this, or if we do, it will have to actually access Core Data, and this would not be a good unit test. So, let's not worry about this method and just make sure the `fetch` method is tested. I have provided the `fetch` method here for reference:

```
func fetch<Type: NSManagedObject>() -> [Type]? {
    var result: [Type]? = nil

    managedObjectContext?.performAndWait { [weak self] in
        do {
            result = try self?.executeRequest()
        }
        catch {
            print(error)
        }
    }
    return result
}
```

Yep, this method, we can test. We can spy on `executeRequest` and mock that method. We can then test to make sure `performAndWait` is called and we get the expected results. It might not seem like much, but our foundation will be covered and we can build on this. First things first, we need a couple new files under the GroceryAppTests group. These two files should be called `Mocks.swift` and `Mockable.swift`. These files are going to contain the fake objects from our app and the mock work we did from Chapter 8, respectively. Figure 11-12 shows these new files in the Project Navigator.

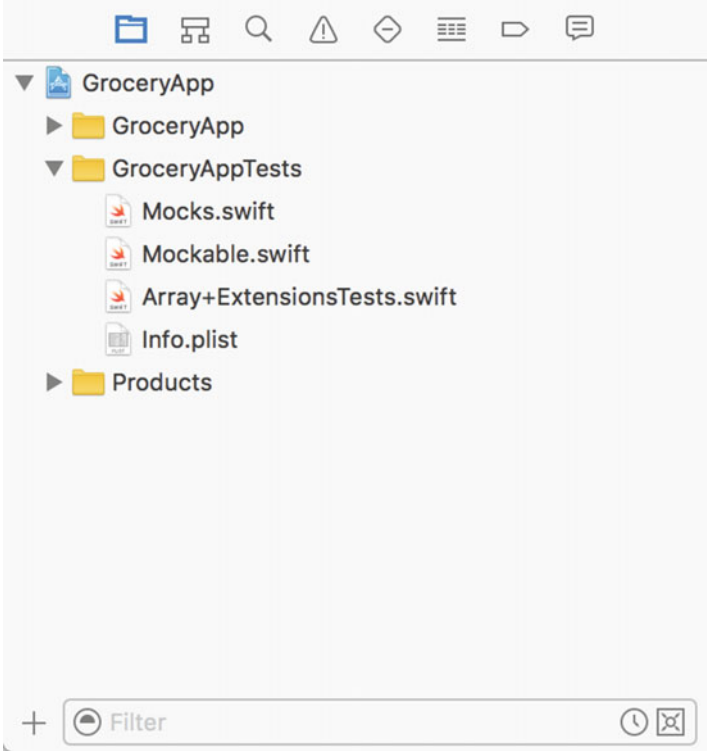


Figure 11-12. Xcode's Project Navigator after creating two new files in for testing

> **Note** Be sure to add both of these files under the test target and not the app's target.

Let's start with the `Mockable.swift` file. We have already written the code for this in our playground on page ***Ch08—Mocks***. I have included the code from this section here. This code is going to help us create mock objects for our tests:

```
struct MockParameter {
    var param: Any

    init(value: Any) {
        param = value
    }

    func value<T>() -> T? {
        return param as? T
    }
}

class Mocked {
    typealias Invocation = (signature: String, parameters: [Any])
    typealias ReturnValue = (signature: String, value: Any)
```

```
    var calls: [Invocation]
    var returnValues: [ReturnValue]

    init() {
        calls = []
        returnValues = []
    }

    func record(method call: String, with parameters: Any...) {
        calls.append( Invocation(signature: call, parameters: parameters) )
    }

    func set(value: Any, for call: String) {
        returnValues.append( ReturnValue(signature: call, value: value) )
    }

    func value<T>(for call: String) -> T? {
        return returnValues.first {
            $0.signature == call
        }?.value as? T
    }

    func invocations(for call: String) -> Int {
        return calls.filter {
            $0.signature == call
        }.count
    }

    func parameters(for call: String) -> [MockParameter] {
        let invocation = calls.first {
            $0.signature == call
        }
        return invocation?.parameters.flatMap {
            MockParameter(value: $0)
        } ?? []
    }
}
```

This is the class that drives the mocks, but we also implemented a protocol in Chapter 8. Here is the `Mockable` protocol and its extension:

```
protocol Mockable {
    var mocked: Mocked { get set }

    func record(method call: String, with parameters: Any...)
    func set(value: Any?, for call: String)
    func value<T>(for call: String) -> T?
    func invocations(for call: String) -> Int
    func parameters(for call: String) -> [MockParameter]
}

extension Mockable {
```

```
    func record(method call: String, with parameters: Any...) {
        mocked.record(method: call, with: parameters)
    }

    func set(value: Any?, for call: String) {
        mocked.set(value: value, for: call)
    }

    func value<T>(for call: String) -> T? {
        return mocked.value(for: call)
    }

    func invocations(for call: String) -> Int {
        return mocked.invocations(for: call)
    }

    func parameters(for call: String) -> [MockParameter] {
        return mocked.parameters(for: call)
    }
}
```

All right, now we can start building the mock objects we need to test the view model's `fetch` functionality. First, we need a mock of `NSManagedObjectContext`. Let's place the following code in our `Mocks.swift` file:

```
class MockManagedObjectContext: NSManagedObjectContext, Mockable {
    var mocked: Mocked = Mocked()

    override func performAndWait(_ block: @escaping () -> Void) {
        record(method: "performAndWait", with: block)
        block()
    }
}
```

This fake context conforms to our protocol `Mockable`, and we are able to fake the `performAndWait` method. We record the method with the block parameter and then we execute the block. Executing the block closure allows the code within our closure to run. Otherwise we would need to get the parameter and manually call the block. Now, we need a fake object that conforms to our `PersistentContainer` protocol. We can then inject this object into our view model under test so it will contain the mocked objects. Here is the code for that object:

```
import CoreData
@testable import GroceryApp

class MockPersistantContainer: PersistentContainer {
    var persistentContainer: NSPersistentContainer
    var managedObjectContext: NSManagedObjectContext = MockManagedObjectContext(concurrencyT
ype: .mainQueueConcurrencyType)

    init() {
        persistentContainer = NSPersistentContainer(name: "fake", managedObjectModel:
NSManagedObjectModel(byMerging: [])!)
    }
}
```

This might look a little scary at first. We did not mock the `NSPersistentContainer` because I found no way to mock it. If you look at our code, we only use `NSPersistentContainer` to get a reference to the managed object context in our protocol extension. We are subverting this and what really matters here is the mocked managed object context.

Now, we are finally set up to write our view model test. Let's create a new file called `ViewModelTests.swift` under the GroceryAppTests group. This file will need to import `XCTest` and the `GroceryApp` using the `@testable` attribute. Here is the new test file setting up our mock object:

```
import XCTest
import CoreData
@testable import GroceryApp

class ViewModelTests: XCTestCase {

    var mockPersistentContainer: MockPersistantContainer!

    override func setUp() {
        super.setUp()

        mockPersistentContainer = MockPersistantContainer()
    }

}
```

We cannot set up our subject yet because we need a view model that can spy on the executeRequest method. This will require a view model subclass, but instead of placing the class in the Mocks.swift file, let's create it right here in our test class:

```
class ViewModelTests: XCTestCase {

    var mockPersistentContainer: MockPersistantContainer!

    override func setUp() {
        super.setUp()

        mockPersistentContainer = MockPersistantContainer()
    }

    class MockViewModel: ViewModel, Mockable {
        var mocked: Mocked = Mocked()

        override func executeRequest<Type : NSManagedObject>() throws -> [Type]? {
            record(method: "executeRequest")
            return value(for: "executeRequest")
        }
    }
}
```

Now, let's write our test for the fetch method:

```
func testFetchReturnsArrayOfManagedObjects() {
    let spy = MockViewModel(persistentContainer: mockPersistentContainer)
    let expected = [NSManagedObject()]
    spy.set(value: expected, for: "executeRequest")

    // Action
    let result: [NSManagedObject]? = spy.fetch()

    // Asserts
    let objectContext = mockPersistentContainer.managedObjectContext as?
    MockManagedObjectContext
    XCTAssertEqual(1, objectContext?.invocations(for: "performAndWait"))
    XCTAssertEqual(1, spy.invocations(for: "executeRequest"))
    XCTAssertEqual(expected, result ?? [])
}
```

And that's it. Now, before we go through this code, recall that there was a lot of setup to get it to work right. This is where the discussion from Chapter 8 has to come into play. We have to ask ourselves how much value this test is providing. I cannot answer this for you; I can only show you how.

So, let's walk through the code. We first create our mock view model that can spy on the executeRequest method by injecting the mock persistent container. We then set up our expected values for the aforementioned mock method and run our action fetch. We then confirm that the performAndWait method on the mock managed object context was called and everything is what we expected it to be. Run this test and let's see what happens.

It should pass and if you have code coverage turned on; you will see we have covered this method in our view model. And that does it for this section. We have only scratched the surface of the amount of testing we could apply to our app. Hopefully, this section has given you some insight into using Chapter 8's `Mockable` code and how you would mock these objects to only test your code.

Wrap Up

That's a wrap for the Core Data chapter, and I think we handled it better than Indiana Jones. The last section was on testing, but Figure 11-11 proves our app works correctly with Core Data, which is an awesome feat considering we started with completely fake data and tuples. We could have used Core Data from the start, but I wanted us to switch between the fake data and Core Data once we were ready because starting Core Data too soon can make things complicated.

I have developed apps with Core Data from the start and I usually end up changing my data model a couple times throughout the process. There is nothing necessarily wrong with this, but I find the process slow when I am trying to solidify my ideas. By using the tuples and the fake data, we were able to get the core concepts of our app developed. Then once we knew we were ready to use Core Data, there was hardly any work involved in switching the fake data out. Once we had our protocol in place on the `AppDelegate`, the only thing we had to change was the `GroceryListsViewModel`. It was trying to access the fake data container and we switched it to call `fetch`. Everything else was net new code.

Finally, we finished this chapter up on the subject of testing. We saw the lengths we were required to go to to test the core functionality of our Core Data integration. Right or wrong, this is what it took to test our code. Maybe it is worth it to you, maybe it isn't. We are almost done. The next chapter is our last chapter and all we have left is to complete the functionality for the last two view controllers. Chapter 12 is going to bring this all together, so let's keep going.

Chapter 12

Grocery App: Finish Line

Welcome to the last chapter! It's been a long road getting here. In this chapter, we are going to take our app, which has successfully integrated Core Data, and finish it. We are going to finally implement our view controllers that are responsible for adding grocery lists and items. We will also be implementing more protocols and more tests. We are going to pull in code and tests from Chapter 8. By the end of this chapter, our app should be capable of displaying/adding grocery lists and grocery items and persisting the information.

What You've Learned

This chapter is going to use the knowledge you have learned throughout this book. We have implemented protocol-oriented programming techniques. We have implemented complex iOS architecture patterns to avoid common AntiPatterns. We have seen how to use generics in Swift, how to build interfaces through storyboards, and the ins and outs of Xcode. And we have also discussed how to test our code through the use of mocks and protocols. There is nothing left to learn in this chapter. We are just going to bring this app across the finish the line, so let's talk about what's left.

We have two view controllers we need to finish, the `AddGroceryListViewController` and the `AddGroceryItemViewController`. These view controllers will be responsible for adding new lists and new items. After we finish these, we then need a way to reload the UI once a user has created a new list or item. We are also going to bring code in from Chapters 5, 7, and 8. This code will allow us to block the UI and display alerts to the user in the case of errors. We still have a lot left to accomplish, so let's get to it.

Adding Grocery Lists

This section is going to cover the first half of missing functionality. By the end of this section, we should be able to add new grocery lists to our app and see the UI update. We are going to work bottom up, so by the time we reach the view controller, we will be set up for success. Let's start with our data models.

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5_12

Core Data

Let's open our `GroceryList+CoreDataClass.swift` file. The following code block is the state of this class:

```
import Foundation
import CoreData

public class GroceryList: NSManagedObject {

}
```

Currently, all the functionality for this class is held within extensions. What we need to do is add a convenience initializer. In Swift, there are two types of initializers, designated and convenience. The *designated initializer* is meant to be the main source of initialization for the object. A *convenience initializer* is meant to handle some specific case to make initialization easier. The following code block is a reference for how we have to instantiate a `GroceryList` object:

```
let managedObjectContext = persistentContainer.viewContext


if let entity = NSEntityDescription.entity(forEntityName: "GroceryList", in:
managedObjectContext) {
            let list = GroceryList(entity: entity, insertInto: managedObjectContext)
}
```

We need a reference to an `NSEntityDescription` object. We then use the entity description and a managed object context to create the `GroceryList` object. The `entityForEntityName` class method also returns an optional value and the `GroceryList` designated initializer does not allow an optional value. Let's encapsulate this logic in a convenience init on `GroceryList`. Let's also make this a failable initializer, so we do not need to use any implicit unwrapping. Here is the new `GroceryList+CoreDataClass.swift`:

```
import Foundation
import CoreData

public class GroceryList: NSManagedObject {

    convenience init?(managedObjectContext: NSManagedObjectContext?) {
        guard   let mObjCtx = managedObjectContext,
                    let entity = NSEntityDescription.entity(forEntityName: "GroceryList",
in: mObjCtx)
        else {
            return nil
        }
        self.init(entity: entity, insertInto: mObjCtx)
    }
}
```

By making this initializer failable, we can return `nil` if we cannot construct the necessary pieces to create a `GroceryList`. We make this initializer failable by using the `?` character directly after the `init` keyword. To construct our `GroceryList` object, we need a non-optional managed-object context and a non-optional entity. If either of these fail, then we return `nil` and our `GroceryList` is not constructed. Otherwise, we call the designated initializer on `self`.

Before we move on, let's move the entity name to our `GAConstants.swift` file. Currently our constants look like the following:

```
/// Grocery App Constants
struct GAConstants {

    /// Constants for our table views
    struct TableCell {
        /// Cell Identifier for our grocery list and grocery item table views
        static let rightDetail = "Cell"
    }
}
```

Let's add another inner struct called Entities and add our GroceryList entity name:

```
/// Grocery App Constants
struct GAConstants {

    /// Constants for our table views
    struct TableCell {
        /// Cell Identifier for our grocery list and grocery item table views
        static let rightDetail = "Cell"
    }

    struct Entities {
        static let groceryList = "GroceryList"
    }
}
```

Now our initializer can use this constant:

```
convenience init?(managedObjectContext: NSManagedObjectContext?) {
    guard   let mObjCtx = managedObjectContext,
                let entity = NSEntityDescription.entity(forEntityName: GAConstants.Entities.
                groceryList, in: mObjCtx)
    else {
        return nil
    }
    self.init(entity: entity, insertInto: mObjCtx)
}
```

All right, we have added a convenient way to construct our `GroceryList` object, so now we need to use this functionality. Next, we need to give more functionality to our grocery lists view model. Currently, the grocery lists view model holds onto the grocery list data, returns the information for a specific grocery list, and can transfer a grocery list to the protocol type `SelectedGroceryListContainer`. We need a new method for creating grocery lists and making sure they are persisted once they're created. We are going to start with the base view model.

View Model

Let's add a new save method to our base view model class. This method is going to use the managed object context we have and call `save`. The `save` method on the managed object context can throw an error, so let's make a higher method in the chain worry about the error. Navigate to the `ViewModel.swift` file and let's add the following functionality:

> **Note** I have omitted the rest of the `ViewModel` class due to code size. I have placed this method directly under the `// MARK: - Core Data`.

```
func save() throws {
    try managedObjectContext?.save()
}
```

This gives our view models the ability to uniformly save the managed-object context. Let's move up the chain to the `GroceryListsViewModel`. We now need a way to create a new grocery list object. Before we implement this functionality, let's think about it for a second. We already know this method will be capable of throwing errors since it will call the previous `save` method. Another thing we need to consider is the view controller side. The view has a text field to allow the user to enter the list name. When we pull text from a text field, it is an optional string, so let's make sure we allow optional strings to make this easier on the view controller.

Also, constructing our `GroceryList` object could fail. There are a number of issues to deal with here, so let's start by creating our own `Error`. At the top of the `GroceryListsViewModel.swift` I have created the following enum:

```
enum GroceryDataError: Error {
    case Saving(String)
}
```

This enum conforms to the *Error protocol*. This was renamed in Swift 3 from *ErrorType*. We have one case and it is the saving case that holds onto a string. This is called an *associated value* and it allows the case to store a given value. You can see how useful this is for errors. Not only can we have a save error, but it can store more information about the specific error that occurred.

All right, we know there is a bit of work to do, so let's build this method up a piece at a time. First, let's start by creating a new list, setting the name of the list, and then calling `save`:

```
func createGroceryList(with name: String?) throws {
    let newList = GroceryList(managedObjectContext: managedObjectContext)
    newList?.name = name
    try save()
}
```

Cool, but there are a couple things that seem wrong with this implementation. First, if the app fails to create a grocery list, we should probably not continue on as if nothing happened. We do not want to crash, just account for this error. To account for this, let's add a guard statement for our `newList` property and then throw our new `GroceryDataError`.

```
func createGroceryList(with name: String?) throws {
    guard let newList = GroceryList(managedObjectContext: managedObjectContext) else {
        throw GroceryDataError.Saving("There was an error creating new grocery list \
(name)")
    }
    newList.name = name
    try save()
}
```

Now the code will not continue unless it successfully creates a new grocery list object. I also just noticed one last issue. What if the user accidentally taps the *New List* button before entering a name? We need one more condition to our guard that keeps us from creating a new grocery list if the name is empty. Let's add this now:

```
func createGroceryList(with name: String?) throws {
    guard   name?.isEmpty == false,
                let newList = GroceryList(managedObjectContext: managedObjectContext) else {
                    throw GroceryDataError.Saving("There was an error creating new grocery
list \(name)")
    }
    newList.name = name
    try save()
}
```

This will prevent us and users from making any mistakes while creating a new grocery list. And that is all the functionality our grocery lists view model needs. Let's use all of the work we just did to complete the first view controller, the `AddGroceryListViewController`.

View Controller

We've made it to the view controller! This section is going to implement the `AddGroceryListViewController`. I have included Figure 7-2 again here; it shows the interface for this view controller. Based on the interface, our view controller is going to have one `@IBOutlet` for the text field. We are then going to have two `@IBAction` methods for the *Cancel* button and the *Add List* button.

Interface for adding a grocery list in our future app

First, we need to create a new file called `AddGroceryListViewController.swift` under the Grocery Lists group. Figure 12-1 shows the Project Navigator after creating this new file.

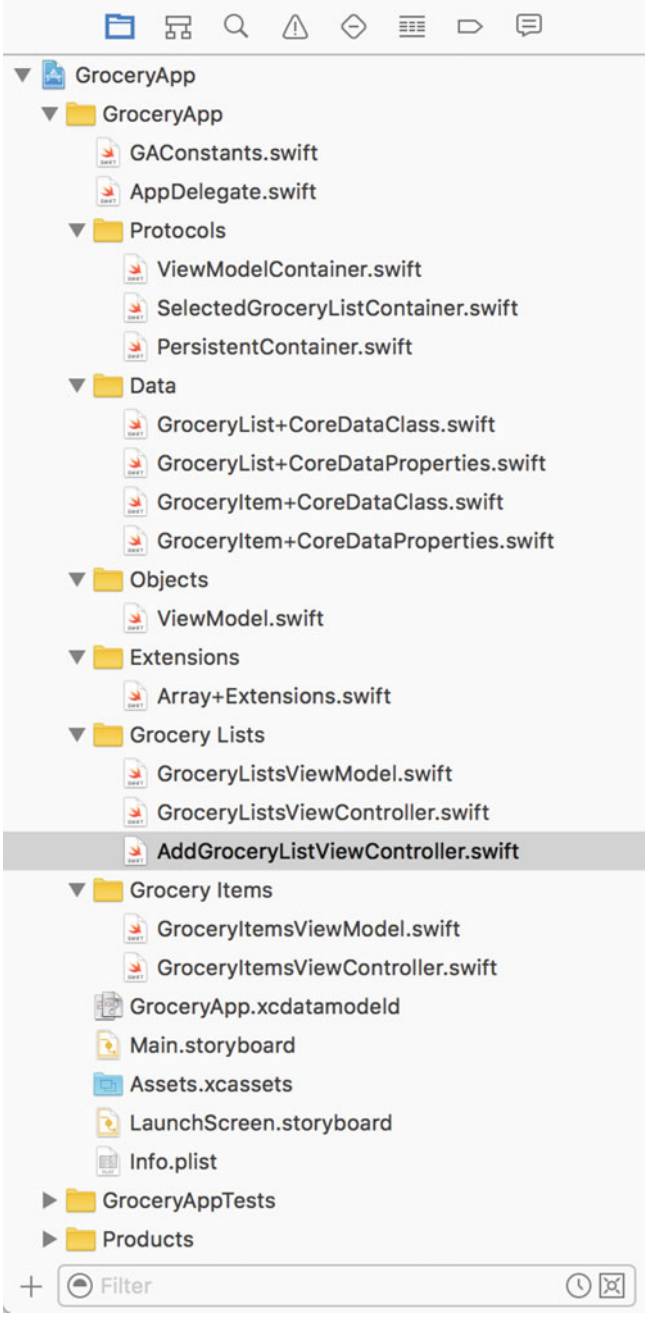


Figure 12-1. *Xcode's Project Navigator after creating the AddGroceryListViewController.swift file*

Once we have the file created, let's add our new class that inherits from `UIViewController`. We also know it is going to need a view model, so let's make it conform to our `ViewModelContainer` protocol and give it a new `GroceryListsViewModel` object:

```
import UIKit

class AddGroceryListViewController: UIViewController, ViewModelContainer {

    // MARK: - Properties

    var viewModel: GroceryListsViewModel = GroceryListsViewModel()
}
```

Let's add the `@IBOutlet` for the text field:

```
class AddGroceryListViewController: UIViewController, ViewModelContainer {

    // MARK: - Properties

    var viewModel: GroceryListsViewModel = GroceryListsViewModel()

    @IBOutlet var groceryListName: UITextField?
}
```

Let's now add our first `@IBAction`. When the user taps *Cancel*, we want the view to dismiss. We also want to reuse this functionality, so let's name it `dismiss` instead of `cancel`:

```
class AddGroceryListViewController: UIViewController, ViewModelContainer {

    // MARK: - Properties

    var viewModel: GroceryListsViewModel = GroceryListsViewModel()

    @IBOutlet var groceryListName: UITextField?

    // MARK: - Actions

    @IBAction func dismiss() {
        dismiss(animated: UIView.areAnimationsEnabled, completion: nil)
    }
}
```

We can now use this functionality when the user successfully creates a new grocery list. The last `@IBAction` is going to be slightly more complex. The API for creating a new grocery list can throw, so we are going to need a `do catch` block. We then just call `createGroceryList` with the text field's data and then call `dismiss` once it completes successfully. The following is the entire `AddGroceryListViewController` class:

```
class AddGroceryListViewController: UIViewController, ViewModelContainer {

    // MARK: - Properties
```

```
    var viewModel: GroceryListsViewModel = GroceryListsViewModel()

    @IBOutlet var groceryListName: UITextField?

    // MARK: - Actions

    @IBAction func dismiss() {
        dismiss(animated: UIView.areAnimationsEnabled, completion: nil)
    }

    @IBAction func addList() {
        do {
            try viewModel.createGroceryList(with: groceryListName?.text)

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

All right, we have implemented our `AddGroceryListViewController`. We now need to hook up our outlets and actions in the storyboard. First, we need to change the identity of the view controller on the storyboard to match the correct one, so Figure 12-2 shows the Identity Inspector for the Add Grocery List view controller.

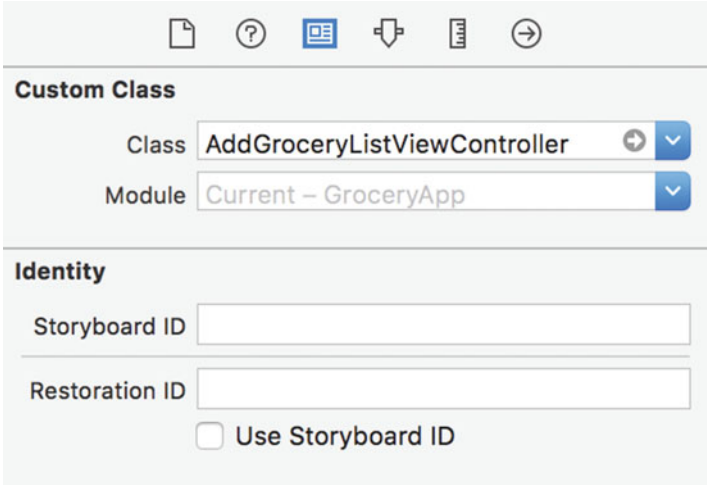


***Figure 12-2.** The Identity Inspector for the AddGroceryListViewController*

Let's now set the outlet and actions on our view controller. Figure 12-3 shows the Connections Inspector for the Add Grocery List view controller. You can see in Figure 12-3 we have one outlet connection for our text field and two actions: one for `dismiss`, one for `addList`.

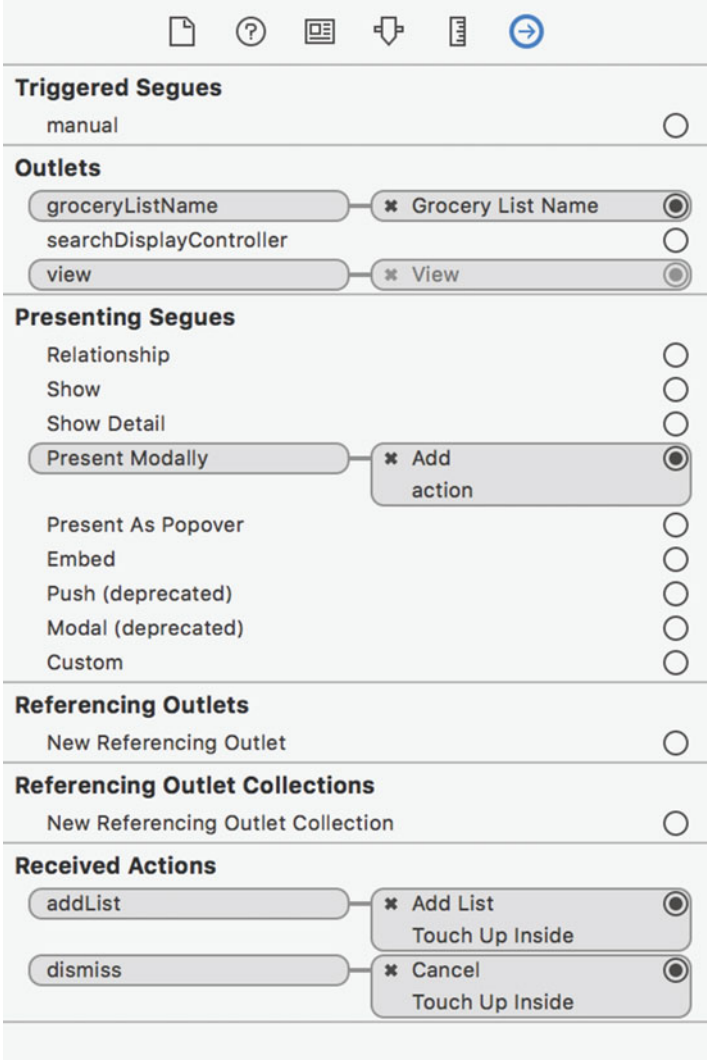


Figure 12-3. *Xcode's Connections Inspector for the AddGroceryListViewController. The connections show an outlet to the text field and two actions including dismiss and addList.*

We have set up our view controller with the storyboard. We have implemented the code behind the `AddGroceryListViewController`. We have also implemented code for the creation of grocery lists on our view model. If you build and run the app, we should be able to navigate to the Add Grocery List view and cancel out of the view. Now, let's try our error-handling logic. Before typing anything into the text field, tap *Add List*. The app should not do anything, but if you look in Xcode's console, you should see the following error: *Saving("There was an error creating new grocery list Optional(\"\")")*. Our error handling is working perfectly.

Now, type something into the text field, tap *Add List*, and then the view should dismiss and nothing happens. Before you start adding more lists, close the app and rerun it. After restarting, you should be able to see your new list on the first screen. Nothing happened after tapping *Add List* because we have not set up our app to reload. This is going to be the focus of our next section.

Refreshing the UI

We just saw our app actually add a real grocery list! The problem was that nothing reloaded once we added our list. We had to shut down the app and restart it to reload the data. This is not how we want our app to work, and real users would see this as a horrible bug. Let's discuss why the table view did not reload its data.

We are using a *modal* presentation style to display the `AddGroceryListViewController`. When this controller is dismissed, the `GroceryListsViewController` runs the `viewDidAppear` lifecycle method, but our table view is a subview of our view controller, so nothing tells it to reload. The `GroceryListsViewController` does not do a full load like it does on app start. We also have a brand new view model on the `AddGroceryListViewController`, so there is no way to connect this back to the `GroceryListsViewController`.

So, how can we solve this problem? On the `GroceryListsViewController`, we already have an outlet to our Grocery Lists table view. We could override the `viewDidAppear` method like the following code block. Do *not* implement the following code:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)

    groceryListTableView?.reloadData()
}
```

This will work and we could move on, except this is inefficient. Every time this view controller's view appears, it will reload the table view whether there was new data or not. So how can we implement this efficiently? Let's use something called an *unwind segue*. The segues we have used up until now have taken us forward to a new view controller. The purpose of an unwind segue, is to take us backward. These can be a bit difficult to understand and set up for the first time, so let's take it slow.

Navigate to the `GroceryListsViewController.swift` file and let's add a new `@IBAction` method under the `// MARK: - Segue` section. Let's call this new action `unwindToGroceryLists`. The following code block defines this method. I have omitted the rest of the code from this view controller:

```
@IBAction func unwindToGroceryLists(segue: UIStoryboardSegue) {
    groceryListTableView?.reloadData()
}
```

It is important to make sure we have a parameter called `segue` and that it is of type `UIStoryboardSegue`. Then the implementation of this method calls `reloadData` on the `groceryListTableView`. The unwind segue is difficult to explain and understand because we have implemented this action on our `GroceryListsViewController`, but it is available to us within our entire app. Let's navigate back to our storyboard and use this unwind segue.

Figure 12-4 shows the Add Grocery List view controller on our storyboard. The *Exit* icon is highlighted in Figure 12-4. If you right click on the *Exit* icon, you should see our `unwindToGroceryListsWithSegue` action. The menu displaying our unwind action can be seen in Figure 12-5.

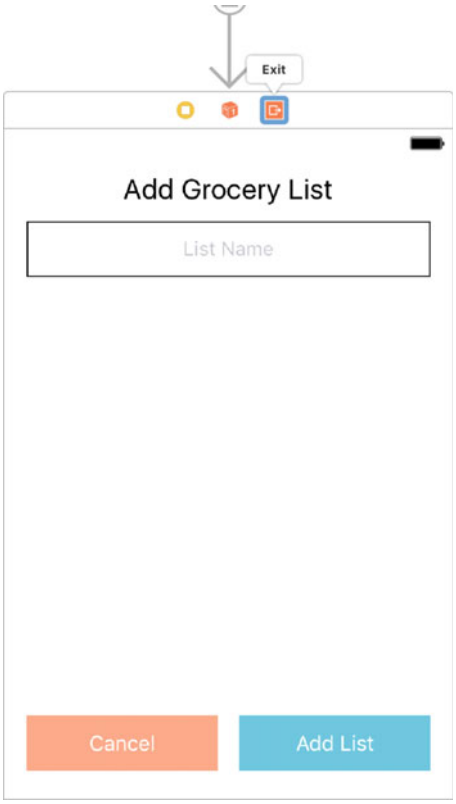


Figure 12-4. *The AddGroceryListViewController in our storyboard. The Exit icon, that holds the unwind segue, is highlighted.*

Figure 12-5. *The menu after right clicking on the Exit icon from Figure 12-4*

Figure 12-5 shows the new manual segue created by using this unwind segue. To create this manual segue, we need to control-drag from the yellow view controller icon to the *Exit* icon and select our unwind action. Once you have successfully created the manual segue, a reference to it should show up in the left pane of our storyboard. This can be seen in Figure 12-6. Manual segues are no different than automatic segues, except we have to manually add code in our view controller to invoke them. We can use the `performSegueWithIdentifier` method to execute the unwind segue.

The last part of our storyboard we have to configure is the identifier for this manual unwind segue. Figure 12-6 shows the two steps to configuring the identifier. In the left pane of our storyboard, we must select the manual segue reference that was created and then we can configure this segue in the Attributes Inspector.

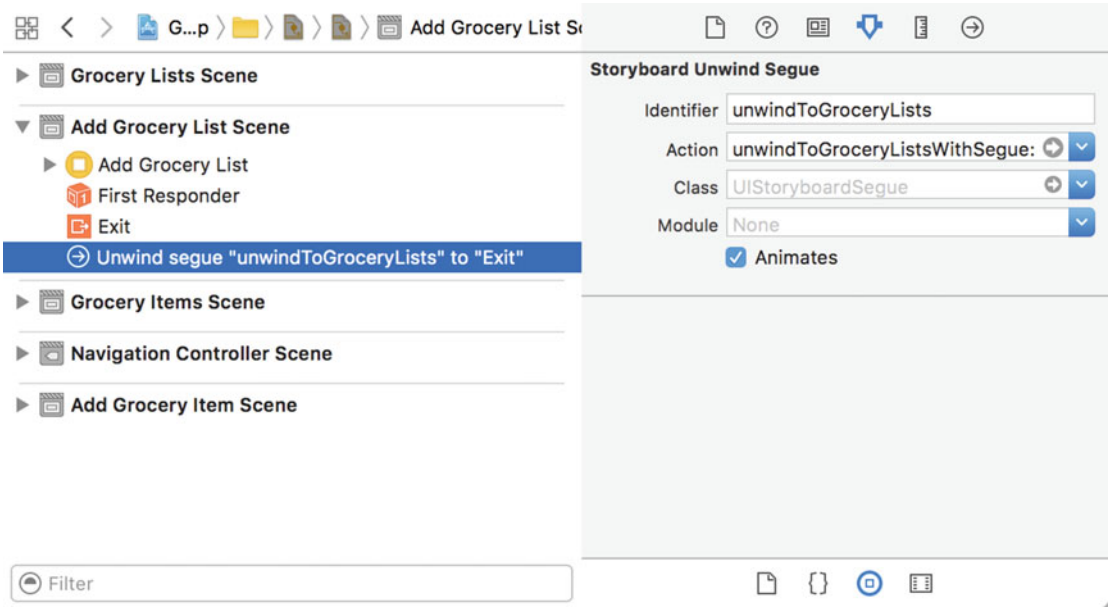


Figure 12-6. *The two steps to configuring the segue identifier in our storyboard. The left-hand side displays the hierarchy menu with the manual segue reference. The right-hand side displays the Attributes Inspector with the configuration for the unwind segue.*

Whew! That was a lot of work for one segue. We still have one last step before this can work. Navigate to the `AddGroceryListViewController.swift` file and let's change one line of code. Previously the `addList` method looked like the following:

```
@IBAction func addList() {
    do {
        try viewModel.createGroceryList(with: groceryListName?.text)

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

To manually invoke our segue, let's replace the call to the `dismiss` method with `performSegueWithIdentifier`:

```
@IBAction func addList() {
    do {
```

```
            try viewModel.createGroceryList(with: groceryListName?.text)

            performSegue(withIdentifier: "unwindToGroceryLists", sender: nil)
        }
        catch {
            print(error)
        }
    }
```

Now, if we rerun our app and add a new list, the Add Grocery List view controller should dismiss and our table view should have the new list! Before we move on, there is one last thing left to do here. We want to refactor our previous code block to use our `GAConstants` instead of a hardcoded string. Here is the `GAConstants.swift` file:

```
/// Grocery App Constants
struct GAConstants {

    /// Constants for our table views
    struct TableCell {
        /// Cell Identifier for our the grocery list and grocery item table views
        static let rightDetail = "Cell"
    }

    /// Constants for Core Data entities
    struct Entities {
        /// Entity Identifier for the GroceryList entity in Core Data
        static let groceryList = "GroceryList"
    }

    /// Constants for App Segues
    struct Segues {
        /// Unwind Segue Identifier
        static let unwindToGroceryLists = "unwindToGroceryLists"
    }
}
```

The previous code block has added a new inner struct called `Segues`, which contains the identifier for our unwind segue. Our UI now refreshes without the need to restart the app. What have we done in this section? We have seen how to hook up manual segues and unwind segues. We then used the unwind segue functionality to dismiss the `AddGroceryListViewController` and reload our `groceryListTableView` at the same time. We are also reloading our table view efficiently because we reload on demand instead of every time we navigate to our Grocery List table view. The next section is going to cover alerting the user on error and blocking the view.

Alerts and Blocking

We have already covered most of the code that we are going to write in this section. This section is going to take protocols and functionality from Chapters 5 and 8. We will also write more tests in this section. In the “Balanced Testing” section from Chapter 8, we looked at

how we can use protocols in our tests to handle repeated test code. We are going to build on this concept as well in this section. Let's get started with a protocol. Figure 12-7 shows the progression of our app in the Project Navigator. We are going to focus on the Protocols group for now.

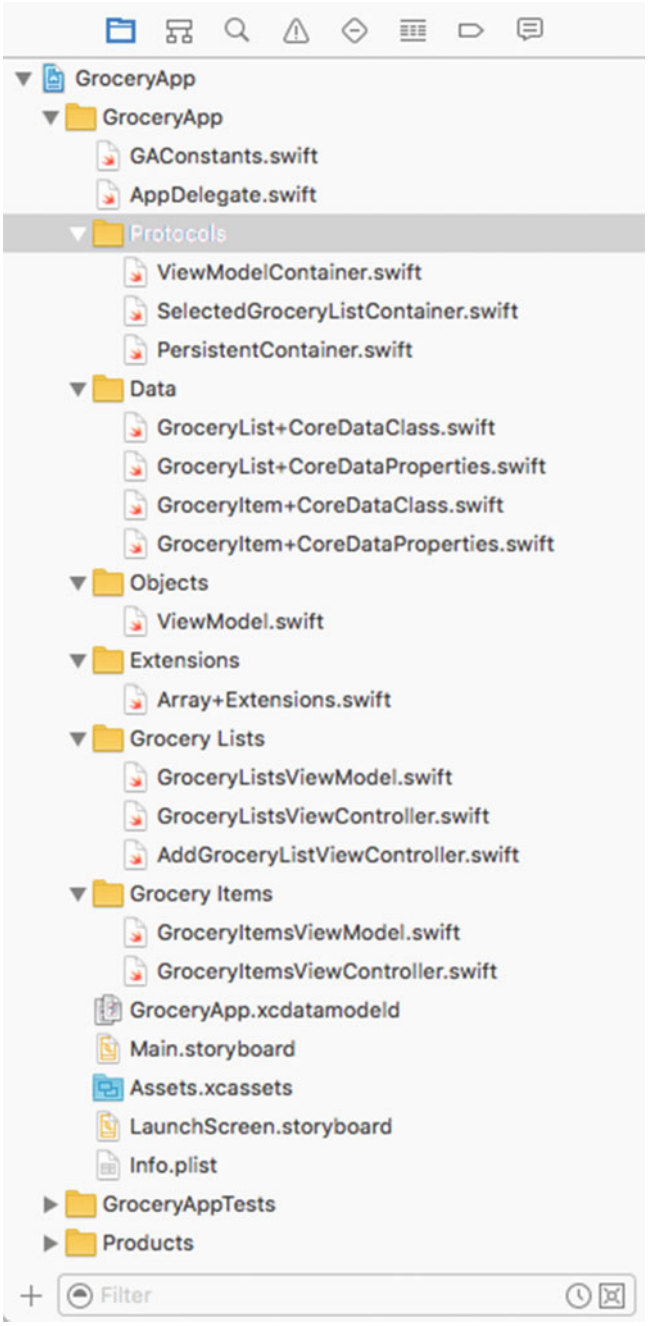


Figure 12-7. *The progress of the GroceryApp in the Project Navigator*

Let's create a new file called `AlertProtocols.swift` in the Protocols group. This file will hold all of the protocols responsible for presenting alerts. The following code has been taken from our playground on pages *Ch05—Protocols in UIKit* and *Ch08—DRY vs WET Testing*. The following protocol defines the trait protocol for presenting view controllers:

```
/// Trait Style Protocol for presentation of view controllers
protocol canPresentViewControllers {
    func present(_ viewControllerToPresent: UIViewController, animated flag: Bool,
    completion: (() -> Void)?)
}
```

The next code block then defines our interface protocol for the actual `AlertDisplayer` functionality:

```
/// Interface Style Protocol that represents the concept of an AlertDisplayer
protocol AlertDisplayer {
    var viewControllerPresenter: canPresentViewControllers { get set }

    func displayAlert(withTitle title: String?, message: String?, withHandler handler: @
    escaping (UIAlertAction) -> Void)
}
```

The previous interface style protocol has a property to present view controllers via the protocol `canPresentViewControllers`. We defined our protocol to facilitate mocking in our tests. The next protocol and extension defines the trait that allows other objects to use the `AlertDisplayer`:

```
/// Trait Style Protocol for displaying alerts through the use of an AlertDisplayer
protocol canDisplayAlerts {
    var alertDisplayer: AlertDisplayer { get set }
    func displayAlert(withTitle title: String?, message: String?, withHandler handler: @
    escaping (UIAlertAction) -> Void)
}

/// Extension that allows any canDisplayAlerts to communicate with the AlertDisplayer via a
method mixed into the conforming object
extension canDisplayAlerts {
    func displayAlert(withTitle title: String?, message: String?, withHandler handler: @
    escaping (UIAlertAction) -> Void) {
        alertDisplayer.displayAlert(withTitle: title, message: message, withHandler:
        handler)
    }
}
```

In Chapter 8, we defined this protocol, so our use of the `AlertDisplayer` can be tested via the `canDisplayAlerts` protocol. The other purpose of this protocol is to illustrate composition over inheritance. Using this protocol, the conforming object, such as a view controller, is forced to use this object to display alerts instead of holding all of that responsibility itself. This makes our code much more scalable and testable. We have one final piece to our alert functionality. We need to use retroactive modeling so all `UIViewControllers` conform to our `canPresentViewControllers` protocol:

```
/// Retroactive Modeling: Any UIViewController has the traits
/// - canPresentViewControllers
extension UIViewController: canPresentViewControllers {}
```

The previous code block allows any of our view controllers to be used with the `AlertDisplayer`. All right, let's now create one more file under the Objects group called `ErrorAlertDisplayer.swift`. The following block is the code for the `ErrorAlertDisplayer` from Chapter 8:

```
import UIKit


/// Struct used to present UIAlertControllers using the canPresentViewControllers protocol
struct ErrorAlertDisplayer: AlertDisplayer {
    var viewControllerPresenter: canPresentViewControllers

    init(viewControllerPresenter: canPresentViewControllers) {
        self.viewControllerPresenter = viewControllerPresenter
    }

    func displayAlert(withTitle title: String?, message: String?, withHandler handler: @
    escaping (UIAlertAction) -> Void) {
        let alert = UIAlertController(title: title, message: message, preferredStyle:
        .alert)
        alert.addAction( UIAlertAction(title: "OK", style: .cancel, handler: handler) )

        viewControllerPresenter.present(alert, animated: UIView.areAnimationsEnabled,
        completion: nil)
    }
}
```

This code might be slightly different than the code from Chapter 8, so let's step through this. We still have our `canPresentViewControllers` property. The `displayAlert` method, however, now has a `handler`, so the caller can supply a closure for the alert action.

Since we are working in Swift 3, we must mark our closure as `@escaping`. This tells the compiler the closure will be saved for execution later. This is the opposite of the `@noescape` attribute in Swift 2. This attribute restricted the escaping behavior of closures so they could not be saved for later. Essentially, Swift 3 has reversed the default behavior for closures in parameters. This gives us greater safety, because we do not need to remember to mark a closure as `@noescape` anymore. We are also forced to think about the behavior of our closure since we need to mark escaping closures with `@escaping`.

Awesome, we have just implemented our `ErrorAlertDisplayer` struct. This struct can be used to present `UIAlertControllers` anywhere we conform to `canPresentViewControllers`, like our view controllers. Now, let's navigate to our `AddGroceryListViewController.swift` file. Previously, when we tried to add a new grocery list and an error occurred, we only printed it out to the console. This is great for debugger, but horrible for users. Let's use our new alert displaying capabilities to give the user an idea of what happened. So in our `AddGroceryListViewController` class, let's start by conforming to our protocols:

```
class AddGroceryListViewController: UIViewController, ViewModelContainer, canDisplayAlerts {

    // MARK: - Properties

    lazy var alertDisplayer: AlertDisplayer = {
        ErrorAlertDisplayer(viewControllerPresenter: self)
    }()

    // ...
}
```

On the first line, we have the `AddGroceryListViewController` class conform to the `canDisplayAlerts` protocol. This will force us to have an `alertDisplayer` property. I have then used the lazy keyword so our `ErrorAlertDisplayer` struct can be instantiated with a reference to this view controller. Cool, now let's use this in our `addList` method whenever an error occurs:

```
@IBAction func addList() {
    do {
        try viewModel.createGroceryList(with: groceryListName?.text)

        performSegue(withIdentifier: GAConstants.Segues.unwindToGroceryLists, sender: nil)
    }
    catch {
        alertDisplayer.displayAlert(withTitle: "Error", message: "\(error)", withHandler: {_
in })
    }
}
```

Now, the `catch` block uses the `alertDisplayer` object to display the error. Let's run our app and see what happens. Be sure to leave the grocery list text field blank as this will generate an error. The result of this work can be seen in Figure 12-8.

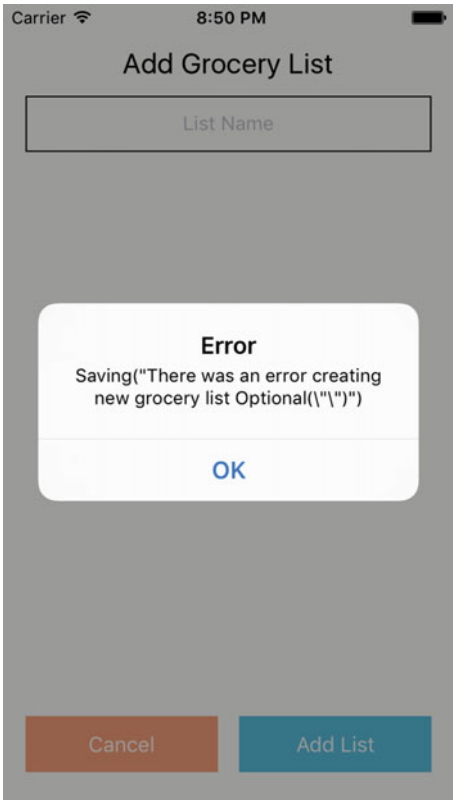


Figure 12-8. *The alert on the add grocery list view controller with our error*

Now, before we can move on to the view blocking functionality, we need to make sure we have tested our previous functionality. This is where we can bring in our concepts from Chapter 8. Chapter 8 discussed WET vs. DRY testing as well as the idea of balance. We want to use the balanced concept here because we are going to have very similar functionality when we finish adding grocery items.

We are not going to implement any tests for the actual `ErrorAlertDisplayer` code. This section is going to cover how we can unit test our `AddGroceryListViewController`. Unit testing view controllers can be a somewhat controversial topic. Some argue we should write UI tests and not unit tests, whereas others are okay with unit tests for view controllers. I leave this up to you and I will just show you what to consider when unit testing view controllers.

All right, so let's start by creating two new files, one called `AddGroceryListViewControllerTests.swift` and the other called `TestBehaviors.swift` in our GroceryAppTests group. Figure 12-9 shows the Project Navigator for the GroceryAppTests group.

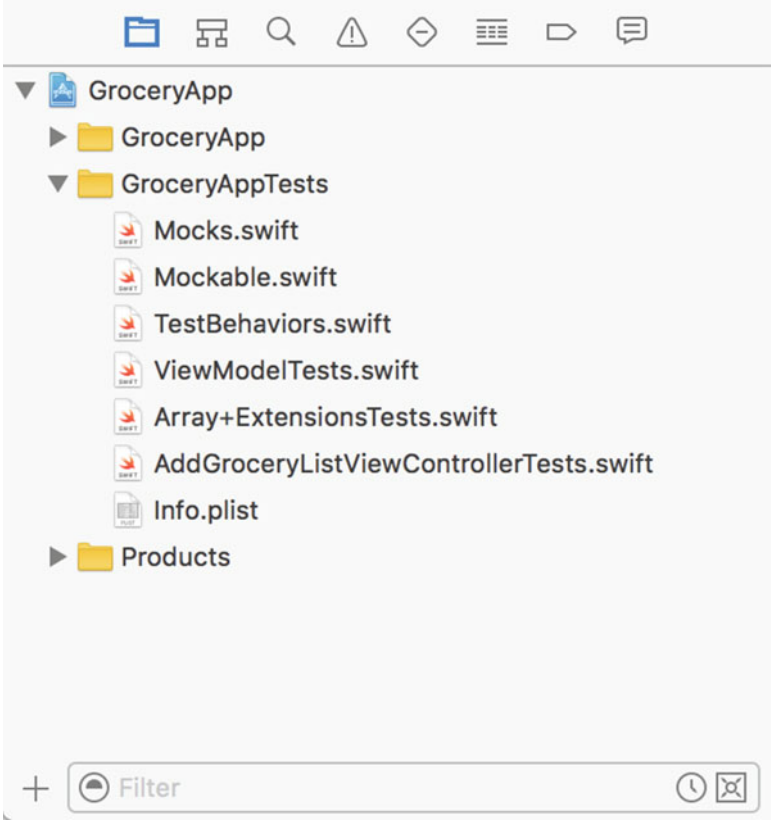


Figure 12-9. *The Project Navigator for the GroceryAppTests group*

The test behaviors file is going to contain the protocols and extensions for our tests. Before we start writing our tests, let's pull in the code from Chapter 8 on the playground page, *Ch08—Balanced Testing*. The following code block first defines the `FakeDisplayer` class we are going to need for tests. I have added this class to the `Mocks.swift` file:

> **Note** Make sure to import UIKit in the `Mocks.swift` file. We need a reference to this framework for the `UIAlertAction` type.

```
import UIKit

/// Used for testing purposes only
class FakeDisplayer: AlertDisplayer, Mockable {
    var mocked: Mocked = Mocked()

    static var displayAlert = "displayAlertWithTitleMessageWithHandler"

    var viewControllerPresenter: canPresentViewControllers
```

```
    init(viewControllerPresenter: canPresentViewControllers) {
        self.viewControllerPresenter = viewControllerPresenter
    }

    func displayAlert(withTitle title: String?, message: String?, withHandler handler: @
escaping (UIAlertAction) -> Void) {
        record(method: FakeDisplayer.displayAlert, with: title, message)
    }
}
```

So, we now have our `FakeDisplayer` class that conforms to two protocols. The first protocol is our `AlertDisplayer` and the second is the `Mockable` protocol. I have added a `static` string variable here so we do not have to hardcode our string in test. I have used the same initializer from the `ErrorAlertDisplayer` and we have the `displayAlert` method just record the method call with the `title` and `message`. I have omitted the `handler` closure, as we cannot assert closures.

Cool, we have our mock object, so let's flip back over to the `TestBehaviors.swift` file and create our protocol. We are going to create one protocol for testing the use of the `AlertDisplayer` object:

```
import UIKit
import XCTest
@testable import GroceryApp

protocol AlertDisplayerTestBehavior {
    func assert<T: canDisplayAlerts & canPresentViewControllers>(
        file: StaticString, line: UInt,
        subject: inout T,
        displaysAlertWithTitle title: String?,
        andMessage message: String?,
        onAction action: (T) -> Void
    )
}
```

There are a lot of parameters to this protocol's method. Let's go through each line. First, we have declared a method called `assert`. This method is a generic method with two constraints on our type `T`. We are using the new Swift 3 protocol composition syntax here to make sure our type conforms to the `canDisplayAlerts` protocol and the `canPresentViewControllers` protocol. These are the protocols necessary for the subject of the test.

The first two parameters are for XCTest. As discussed in Chapter 8, these two parameters are the debug identifiers. They will allow our test to fail on the correct line instead of inside our protocol extension method. The next parameter is the subject of the test and is the generic type `T`. We also need to make sure this parameter is labeled `inout`. This will allow us to modify the internal members of this type even if it is a value type.

The next two parameters describe the values for our assertions. We want to make sure the alert is displayed with the correct title and message. The last parameter is the closure that represents the test action. Whew, let's build the extension for this protocol now:

```
extension AlertDisplayerTestBehavior {
    func assert<T: canDisplayAlerts & canPresentViewControllers>(
        file: StaticString = #file, line: UInt = #line,
        subject: inout T,
        displaysAlertWithTitle title: String?,
        andMessage message: String?,
        onAction action: (T) -> Void
    ) {

        let fakeDisplayer = FakeDisplayer(viewControllerPresenter: subject)
        subject.alertDisplayer = fakeDisplayer

        action(subject)

        let parameters = fakeDisplayer.parameters(for: FakeDisplayer.displayAlert)
        XCTAssertEqual(title, parameters.first?.value(), file: file, line: line)

        let messageParameter: String? = parameters.last?.value()
        XCTAssertEqual(true, messageParameter?.contains(message ?? ""), file: file, line:
line)
    }
}
```

> **Note** Make sure the file and line parameters are defaulted to `#file` and `#line` respectively. This will reference the correct file and line where this method is used.

This extension looks a little scary, but this is the meat of the work. After this, the tests are going to be cake. So what does this extension do? It creates a `FakeDisplayer` object, sets the `alertDisplayer` property on the subject, and then runs the action closure, injecting the subject under test. Finally, we get the list of parameters from the `fakeDisplayer` object and then assert that the `title` and `message` are the correct values. I have used string's `contains` method, so we do not have to include the entire message in our test. You will also notice we have used the file and line parameters in our asserts. This is what allows the test to fail on the correct line and file.

We can finally implement the tests for our `AddGroceryListViewController`. Let's navigate to the `AddGroceryListViewControllerTests.swift` file. Let's set up our test class:

```
import XCTest
@testable import GroceryApp

class AddGroceryListViewControllerTests: XCTestCase, AlertDisplayerTestBehavior {

}
```

We have a subject and it is an `AddGroceryListViewController` type. The test class also conforms to our `AlertDisplayerTestBehavior` protocol. Now, using the functionality the protocol provides, let's write a test for our view controller:

```
class AddGroceryListViewControllerTests: XCTestCase, AlertDisplayerTestBehavior {


    func testShouldDisplayError() {
        var subject = AddGroceryListViewController()

        assert(subject: &subject, displaysAlertWithTitle: "Error", andMessage: "There was an
        error creating new grocery list nil") {
            $0.addList()
        }
    }
}
```

That's very simple! We create a `subject` property and use our protocol `assert` method. We pass in the subject, "`Error`", and the partial message. Then the action closure just uses the injected `subject` parameter to call the `addList` method. This is where view controller testing can be a bit hairy. I have included the code for the `addList` method here for reference:

```
@IBAction func addList() {
    do {
        try viewModel.createGroceryList(with: groceryListName?.text)

        performSegue(withIdentifier: GAConstants.Segues.unwindToGroceryLists, sender: nil)
    }
    catch {
        alertDisplayer.displayAlert(withTitle: "Error", message: "\(error)", withHandler: {_
        in })
    }
}
```

There are a number of things going on here, but one of the more important things I would like to point out is that this method uses the `groceryListName` text field property. It asks for the text on the text field. Now, we made sure to make our `@IBOutlet` an optional property; however, if it is implicitly unwrapped, our test would crash. This is why testing view controllers can be awkward. We have to make sure the view controller object has everything it expects, like `@IBOutlets`.

Another important concept is that we have not written a unit test here. We have technically written an integration test. We are using functionality of the view model, which will throw an error to cause the alert displayer to get called. This is not what we want because we are not isolating our view controller under test. If the behavior of the view model changes, this test would break, even though nothing on the view controller changed. So, the next step is to properly isolate our subject by creating a fake view model. At the bottom of the `AddGroceryListViewControllerTests` class, add the following code:

```
private class FakeGroceryListsViewModel: GroceryListsViewModel {
    private override func createGroceryList(with name: String?) throws {
        throw GroceryDataError.Saving("An error occurred.")
    }
}
```

We have overridden the `GroceryListsViewModel` class and the `createGroceryList` method. We made this method just throw an error. Now, our view controller will be properly isolated. If you add this fake view model object to your test and rerun the test, it will fail because the error has changed. Let's fix this issue. Here is the entire `AddGroceryListViewControllerTests.swift` file for reference:

```
import XCTest
@testable import GroceryApp

class AddGroceryListViewControllerTests: XCTestCase, AlertDisplayerTestBehavior {

    func testShouldDisplayError() {
        var subject = AddGroceryListViewController()
        subject.viewModel = FakeGroceryListsViewModel()

        assert(subject: &subject, displaysAlertWithTitle: "Error", andMessage: "An error
        occurred.") {
            $0.addList()
        }
    }

    private class FakeGroceryListsViewModel: GroceryListsViewModel {
        private override func createGroceryList(with name: String?) throws {
            throw GroceryDataError.Saving("An error occurred.")
        }
    }
}
```

Awesome! That was a lot of work, but it was definitely worth it. From now on, anywhere we use alert displayers we can use this protocol to test the behavior. This is the last testing I wanted us to do for the app. We would be here for days if I covered how to test all the pieces of the app. This is where I want you to take these concepts and run with them. If you want to add more tests, go for it; if not, that's okay too. So, let's wrap up testing for this book.

In Chapter 8, we explored the concept of testing. There are many aspects to this subject. There are many schools of thought such as DRY vs. WET. We also looked at how we can use Swift's protocols to clean up tests and we implemented this protocol-oriented idea in the tests for our app here. We finished Chapter 8 with a short discussion on TDD and iOS. I have talked about the concept of balance when it comes to testing, but what does this mean exactly? This concept boils down to do what's right for you and your situation at the time. I am not for or against practices such as DRY, WET, or TDD, but I want to use the appropriate practice at the correct time. I believe this can be fluid based on the situation. That is what balance is about and what I believe creates high-quality tests. Hopefully this has cleared up the whole discussion on testing.

We are now only missing one piece of functionality—blocking the view. We discussed and implemented this functionality in Chapters 5 and 7. The purpose for this functionality is to not allow the user to enter any more information or tap more buttons before we finish processing our actions. So, in our case, this is not completely necessary because the action is so fast the user has no opportunity to do anything else. However, if our data was stored off of the device and we had to send a network request and wait for a response, there would be enough time. In this case, we would want to block the view.

So, I want you to implement this functionality. We have already covered it and you have the knowledge. This work should include the code from the playground page: *Ch07—Blocking a View*. It should entail the `canBlockView` protocol and an extension. Once you have this protocol and extension implemented, the `addList` method can look like the following:

```
@IBAction func addList() {
    block()
    do {
        try viewModel.createGroceryList(with: groceryListName?.text)

        performSegue(withIdentifier: GAConstants.Segues.unwindToGroceryLists, sender: nil)
    }
    catch {
        unblock()
        alertDisplayer.displayAlert(withTitle: "Error", message: "\(error)", withHandler: {_
        in })
    }
}
```

We make sure to block the view before we start processing the data. We only have to unblock the view if an error occurs. If this completes successfully, the view will disappear, so we only have to unblock if we allow the user to try again. And that does it for this section.

This has been a very big section, but we have seen how to display alerts via protocol-oriented programming and how to test our view controller, and we have put the discussion on testing started in Chapter 8 to bed. We finally wrapped this section up with blocking and unblocking our view controller's view when we need to process data. In the next section, we will implement the same behavior from this section for grocery items. There is going to be less explanation since we have already seen everything we will need to complete our app. We are in the home stretch, so let's finish strong.

Adding Grocery Items

This is the last major section of the book. We are going to finish implementing our Grocery App with the functionality for adding grocery items. We are going to work the same way as in the previous section, bottom up. This section is also going to have fewer explanations for the code. You have followed along through the entire book and you know what to do. Before we start implementing the rest of our app, Figure 12-10 shows our progress in Xcode's Project Navigator.

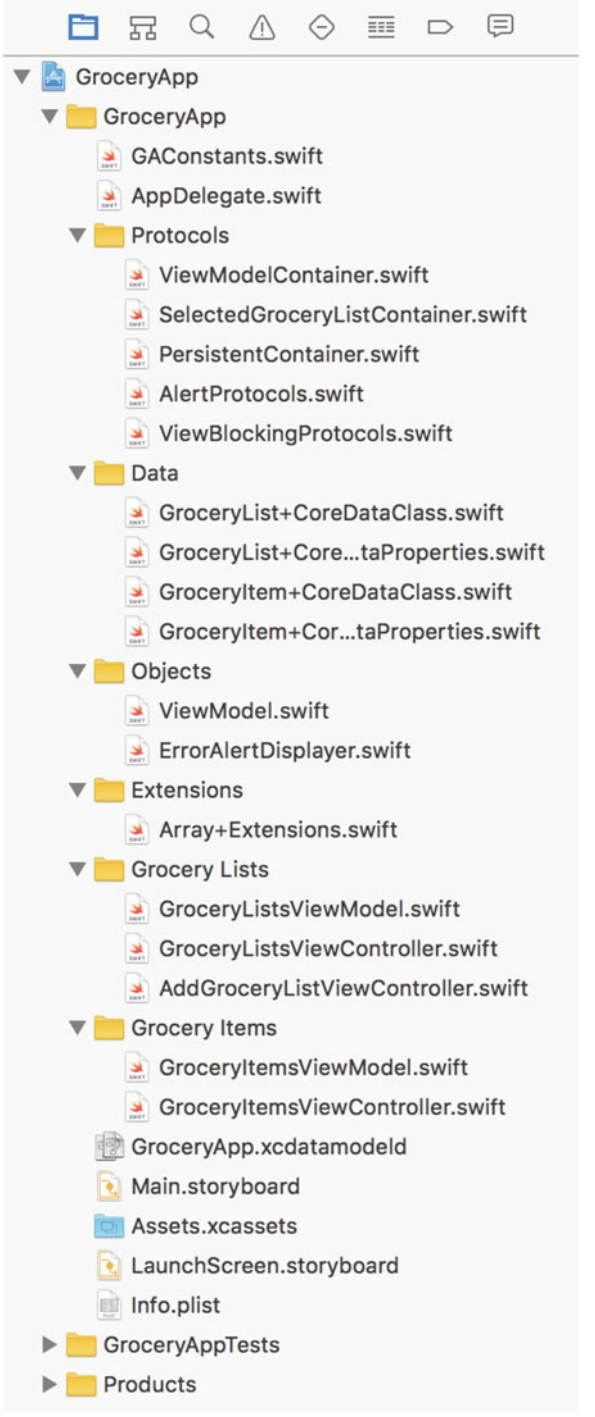


Figure 12-10. Xcode's Project Navigator for our current progress on the Grocery App

Core Data

Let's navigate to the `GroceryItem+CoreDataClass.swift` file and open it. The current state of this file should be an empty class like the following code block:

```
import Foundation
import CoreData

public class GroceryItem: NSManagedObject {

}
```

We are going to add another convenience initializer that has the optional-managed object context as a parameter. Before we add this new initializer, let's create a new constant string in `GAConstants.swift`. Let's put it under the `Entities` struct:

```
/// Grocery App Constants
struct GAConstants {

    /// Constants for our table views
    struct TableCell {
        /// Cell Identifier for our grocery list and grocery item table views
        static let rightDetail = "Cell"
    }

    /// Constants for Core Data entities
    struct Entities {
        /// Entity Identifier for the GroceryList entity in Core Data
        static let groceryList = "GroceryList"
        static let groceryItem = "GroceryItem"
    }

    /// Constants for App Segues
    struct Segues {
        /// Unwind Segue Identifier
        static let unwindToGroceryLists = "unwindToGroceryLists"
    }
}
```

Under the `Entities` struct, we have a new string property called `groceryItem`. This is the reference to our Core Data entity. Next, let's use this constant and implement our failable convenience initializer.

```
public class GroceryItem: NSManagedObject {


    convenience init?(managedObjectContext: NSManagedObjectContext?) {
        guard   let mObjCtx = managedObjectContext,
                    let entity = NSEntityDescription.entity(forEntityName: GAConstants.
Entities.groceryItem, in: mObjCtx)
        else {
            return nil
```

```
        }
        self.init(entity: entity, insertInto: mObjCtx)
    }
}
```

This failable convenience initializer unwraps the managed-object context and the entity description for `GroceryItem`. It returns `nil` if either of these items are not there. Then, once it has the correct information, it calls the designated initializer on self to create our grocery item object. Now both of our data models are complete and `GroceryItem` is ready to use. We will use this functionality in our `GroceryItemsViewModel`.

View Model

Just as was the case with our `GroceryListsViewModel`, we need a way to create new grocery item objects. This functionality will be housed in our `GroceryItemsViewModel`. Let's navigate to the `GroceryItemsViewModel.swift` file and quickly cover the functionality we want to add. The following code block is what our `GroceryItemsViewModel` class looks like currently:

```
class GroceryItemsViewModel: ViewModel {

    // MARK: - Properties

    var groceryList: GroceryList?

    var groceryItems: [GroceryItem] {
        let result = groceryList?.items?.flatMap {
            $0 as? GroceryItem
        }
        return result ?? []
    }

    // MARK: - Grocery Item Data

    func groceryItem(at indexPath: IndexPath) -> (name: String?, quantity: Int16) {
        let item: GroceryItem? = groceryItems.value(at: indexPath.row)
        return (item?.name, item?.quantity ?? 0)
    }
}
```

We currently have a way to get specific grocery item objects based on their index, but that is all the functionality we have. We want to add a new `createGroceryItem` method. This method will work exactly like the `createGroceryList` method. We need to make sure the data is consumable, instantiate our grocery item object, and then call save, or throw an error. Easy peasy. Let's start with the method and our check for bad data:

```
func createGroceryItem(with name: String?, and quantity: Int?) throws {
    guard   name?.isEmpty == false,
                let itemQuantity = quantity,
                let newItem = GroceryItem(managedObjectContext: managedObjectContext)
    else {
```

```
        throw GroceryDataError.Saving("There was an error creating new grocery item \(name)
with quantity: \(quantity)")
    }
}
```

All right, first we make sure the name for the item is not empty and then we unwrap two values. We need to unwrap the `quantity` value. We are going to pull this value from a text field, so it will have to come in as optional. We then create our new `GroceryItem` with the `managedObjectContext`. If any of these steps fail, we just throw our `GroceryDataError` again. Now, let's finish the functionality for this method. We need to use the data we have unwrapped and call `save` on view model:

```
func createGroceryItem(with name: String?, and quantity: Int?) throws {
    guard   name?.isEmpty == false,
                let itemQuantity = quantity,
                let newItem = GroceryItem(managedObjectContext: managedObjectContext)
    else {
        throw GroceryDataError.Saving("There was an error creating new grocery item \(name)
with quantity: \(quantity)")
    }
    newItem.name = name
    newItem.quantity = Int16(itemQuantity)
    try save()
}
```

There is one piece we are missing from the previous code block. We have not associated the new grocery item with the selected grocery list. We have a reference to the selected grocery list, so let's use it. If you look at the `GroceryList+CoreDataProperties.swift` file, you will see all the functionality Core Data gives us. One of those methods is the `addToItems` method. Here is the code for the extension that defines this method:

```
extension GroceryList {


    @objc(addItemsObject:)
    @NSManaged public func addToItems(_ value: GroceryItem)

    @objc(removeItemsObject:)
    @NSManaged public func removeFromItems(_ value: GroceryItem)

    @objc(addItems:)
    @NSManaged public func addToItems(_ values: NSSet)

    @objc(removeItems:)
    @NSManaged public func removeFromItems(_ values: NSSet)

}
```

We can use these methods to add our new grocery item to the selected grocery list. So, our new `createGroceryItem` method will look like the following:

```
func createGroceryItem(with name: String?, and quantity: Int?) throws {
```

```
    guard   name?.isEmpty == false,
            let itemQuantity = quantity,
            let newItem = GroceryItem(managedObjectContext: managedObjectContext)
    else {
        throw GroceryDataError.Saving("There was an error creating new grocery item \(name)
with quantity: \(quantity)")
    }
    newItem.name = name
    newItem.quantity = Int16(itemQuantity)
    groceryList?.addToItems(newItem)
    try save()
}
```

You will also notice we have the quantity come in as an `Int?` type instead of an `Int16?`. The view controller does not need to convert the text field data to the specific Core Data type. It should only need to know it is an `Int` type. And that does it for the Grocery Items view model. I think it is important to take a step back for a second. I am discussing the new functionality more quickly than I did in the previous section on grocery lists, but we have done so much work and kept our code clear, so implementing the last of our behaviors is so easy. I hope you can see the power of what we have built. The next section is going to start bringing everything together and we are going to start creating grocery items.

View Controller

This section is going to use the new functionality of the Grocery Items view model, along with the alert displayer and view blocking protocols, to create the `AddGroceryItemViewController`. We currently do not have this file, so let's create it now. This file should be under the Grocery Items group. There is going to be a bit more work here than in the last couple of sections. We need to create our new view controller and we need to hook up the outlets and actions in our storyboard. We then need to implement the code to create new grocery items.

Let's start with our new view controller. Once you have the new `AddGroceryItemViewController.swift` file created, let's add the following code:

```
import UIKit

class AddGroceryItemViewController: UIViewController {

    // MARK: - Properties

    @IBOutlet var groceryItemName: UITextField?
    @IBOutlet var groceryItemQuantity: UITextField?

    // MARK: - Actions

@IBAction func dismiss() {
        dismiss(animated: UIView.areAnimationsEnabled, completion: nil)
    }
```

```
    @IBAction func addItem() {

    }
}
```

We have our new view controller class now and it contains the two `@IBOutlets` we need for the two text fields on our storyboard. We have also defined the two actions for this view controller, `dismiss` and `addItem`. Let's hop on over to our storyboard and hook these items up.

First, we need to use the Identity Inspector to set the class identity of our view controller to `AddGroceryItemViewController`. Figure 12-2 showed the Identity Inspector for the `AddGroceryListViewController`. We want to repeat this action here. Once we have set up the identity for the `AddGroceryItemViewController`, we want to add the connections for the outlets and the actions. Figure 12-11 shows the *Connections Inspector* for this view controller.

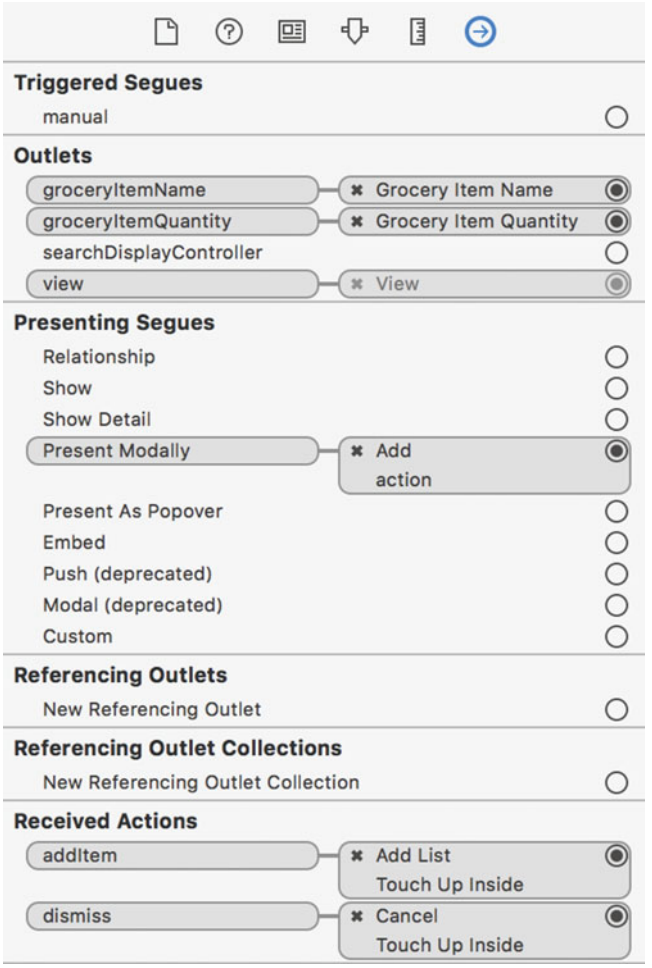


Figure 12-11. *The Connections Inspector for the AddGroceryItemViewController*

All right, we have the base for our view controller ready to go and we have configured our storyboard correctly. If we run the app, we should be able to navigate to the Add Grocery Items view controller, and tapping *Cancel* should dismiss our view controller. Tapping *Add Item* should do nothing yet, so let's implement this method. Just like our Add Grocery List view controller, we need to block the UI; then we can try to create our grocery item, and if anything bad happens, we'll throw an alert onto the screen. Let's start with the protocols we are going to need. The updated class declaration for the `AddGroceryItemViewController` will look like the following:

```
class AddGroceryItemViewController: UIViewController, ViewModelContainer, canDisplayAlerts,
canBlockView {

    // ...

}
```

These protocols will then force us to have all the necessary components to finish this view controller. Remember, we want to lazily initialize our alert displayer property, so we can inject a reference to `self`. The updated `AddGroceryItemViewController` should now look like the following:

```
class AddGroceryItemViewController: UIViewController, ViewModelContainer, canDisplayAlerts,
canBlockView {

    // MARK: - Properties

    var viewModel: GroceryItemsViewModel = GroceryItemsViewModel()

    lazy var alertDisplayer: AlertDisplayer = {
        ErrorAlertDisplayer(viewControllerPresenter: self)
    }()

    @IBOutlet var groceryItemName: UITextField?
    @IBOutlet var groceryItemQuantity: UITextField?

    // MARK: - Actions

    @IBAction func dismiss() {
        dismiss(animated: UIView.areAnimationsEnabled, completion: nil)
    }

    @IBAction func addItem() {

    }
}
```

Exactly what we expected, except there is a somewhat hidden problem with our implementation. In the previous section, when we implemented the functionality for our grocery items view model, we needed the selected grocery list. However, since we create a new `GroceryItemsViewModel` here, it will not have a reference to the selected grocery list.

To solve this, we need to implement a little more code on our GroceryItemsViewController. First, let's conform to the SelectedGroceryListContainer on our AddGroceryItemViewController. This will require a new property and the updated implementation will be as follows:

```
class AddGroceryItemViewController: UIViewController, ViewModelContainer, canDisplayAlerts,
canBlockView, SelectedGroceryListContainer {

    // MARK: - Properties

    var selectedGroceryList: GroceryList? {
        get {
            return viewModel.groceryList
        }
        set {
            viewModel.groceryList = newValue
        }
    }

    // ...
}
```

Now our AddGroceryItemViewController is set up exactly the same as our GroceryItemsViewController. Let's navigate to the GroceryItemsViewController.swift file and implement the functionality to transfer the selected grocery list. We are going to override the prepare-for-segue method on the GroceryItemsViewController. We will then use the segue reference to find the destination view controller and set the selected grocery list if it conforms to our SelectedGroceryListContainer protocol:

```
// MARK: - Segue


override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    super.prepare(for: segue, sender: sender)

    if var groceryListContainer = segue.destination as? SelectedGroceryListContainer {
        groceryListContainer.selectedGroceryList = selectedGroceryList
    }
}
```

Now both our GroceryItemsViewController and the AddGroceryItemViewController transfer the selected grocery list object appropriately. There is only one piece left to this section. We now need to use our UI blocking, view model, and the alert displayer to add new grocery items. Let's navigate back to the AddGroceryItemViewController.swift file and finish this guy. Once we are on this file, let's implement the addItem method as follows:

```
@IBAction func addItem() {
    block()
    do {
        let quantity = Int( groceryItemQuantity?.text ?? "0" )
        try viewModel.createGroceryItem(with: groceryItemName?.text, and: quantity)
```

```
            // TODO: Unwind to the previous view controller
        }
        catch {
            unblock()
            alertDisplayer.displayAlert(withTitle: "Error", message: "\(error)", withHandler: {
            _ in })
        }
    }
```

There is not much to explain from the last code block. We have to create an `Int` for the item quantity, but other than that, this is the same functionality as the `AddGroceryListViewController`. All right, a lot happened in this section and it happened pretty quickly, so let's recap before we move on. First, we created our new `AddGroceryItemViewController` class. This view controller is almost identical in code and purpose to our `AddGroceryListViewController`.

This section also modified the `GroceryItemsViewController`. We found that we needed to transfer the selected grocery list to add our new grocery item. To accomplish this we used the `SelectedGroceryListContainer` protocol and the prepare-for-segue method to transfer the selected grocery. I think how we implemented this is worth talking about for a minute. By using the `SelectedGroceryListContainer` protocol, we made this functionality generic enough to expand if our app expands. Let's say you take this app to the next level and add more screens and a lot of new functionality. If you found you need a reference to the selected grocery list, you can conform to the protocol and the code we put in place in this section would automatically handle transferring this data.

The last thing I want you to do is run your app. Navigate to the Add Grocery Item view controller and create a new grocery item. Nothing should happen and you should be stuck on a screen that is currently blocked. Quit the app and reopen it. You should be able to see your new grocery item. We still need our UI to refresh, but first I wanted to discuss architecture.

In Chapter 4, I introduced multiple design patterns including MVVM. This has been the driving pattern behind our app. I chose this pattern because it creates a nice separation between the data and the view. This separation has allowed us to write clear code for each section. We then use protocol-oriented programming to bridge the gap. Had we implemented the singleton pattern instead of MVVM, I believe we could have finished this app sooner, but we would not have necessarily had the most clear and concise code. All of our view controllers would access the same object and modify the state of that object. Then our singleton would have to be responsible for grocery lists and grocery items. Based on the single responsibility principle from Chapter 4, this would not be a good idea.

We have run into a few issues because we used MVVM. When our data is updated, we have to use unwind segues to inform the view controllers containing the lists of data to reload. We also have to transfer state back and forth. This is not a bad thing, but it is more code. The last point I will leave you with on the subject of architecture is that I like to treat architecture like the new API design guidelines. We want our app to be clear and concise, yet we do not want to sacrifice clarity for brevity. The next section is going to cover the last functionality we are missing. We need our UI to update when we add new grocery items.

Refreshing the UI

Welcome to the last section in this book. Here we are going to make sure our UI updates correctly when we add new grocery items and grocery lists. All of the other functionality for our app is complete. In this section, we are going to create an unwind segue for adding grocery items. The last case we need to handle is when items are added, we need to also update the table view on the first view controller. Let's get started.

First things first, we need to create a new unwind segue. Before we can add any connections in our storyboard, we need a new unwind action in the `GroceryItemsViewController`. Navigate to the `GroceryItemsViewController.swift` file and add the following functionality under the `// MARK: Segue` section:

```
@IBAction func unwindToGroceryItems(segue: UIStoryboardSegue) {
    groceryItemsTableView?.reloadData()
}
```

This new `@IBAction` should now be available in our storyboard. Let's open up our storyboard and configure this manual unwind segue. Navigate to the Add Grocery Item view controller on the storyboard and control-drag from the yellow icon to the *Exit* icon. We should now see two actions available to us. The two actions should be the `unwindToGroceryListsWithSegue` and `unwindToGroceryItemsWithSegue`. We want to connect our Add Grocery Item view controller to the `unwindToGroceryItemsWithSegue`. Once this connection shows up in the left pane under the Add Grocery Item view controller, we need to set the segue identifier. Figure 12-12 shows the Attributes Inspector for the unwind segue. We want to make sure the identifier is `unwindToGroceryItems`.

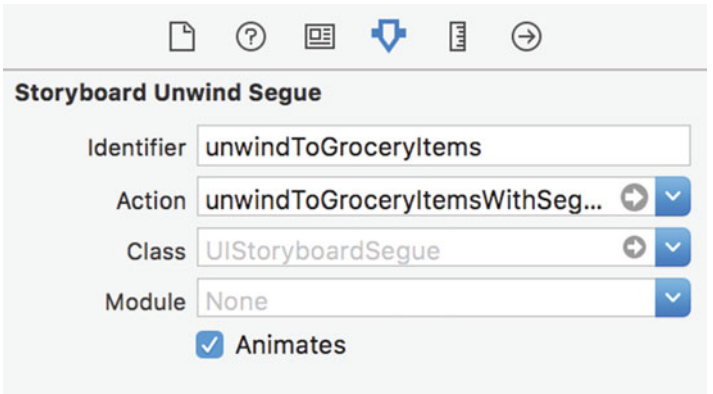


Figure 12-12. *The Attributes Inspector for the unwindToGroceryItemsWithSegue action*

All right, our unwind segue is configured in our storyboard and the action behind it reloads our groceryItemsTableView. Let's navigate to the AddGroceryItemViewController and call this manual segue once we have successfully created the new grocery item. The final implementation of the addItem method is as follows:

```
@IBAction func addItem() {
    block()
    do {
        let quantity = Int( groceryItemQuantity?.text ?? "0" )
        try viewModel.createGroceryItem(with: groceryItemName?.text, and: quantity)

        performSegue(withIdentifier: GAConstants.Segues.unwindToGroceryItems, sender: nil)
    }
    catch {
        unblock()
        alertDisplayer.displayAlert(withTitle: "Error", message: "\(error)", withHandler: {
        _ in })
    }
}
```

We have removed the TODO from the last implementation and we are now calling performSegue with our unwindToGroceryItems segue identifier. Just like our last unwind segue identifier, we want to have this string in our GAConstants.swift file. The GAConstants.swift file should now look like the following:

```
/// Grocery App Constants
struct GAConstants {

    /// Constants for our table views
    struct TableCell {
        /// Cell Identifier for our the grocery list and grocery item table views
        static let rightDetail = "Cell"
    }

    /// Constants for Core Data entities
    struct Entities {
        /// Entity Identifier for the GroceryList entity in Core Data
        static let groceryList = "GroceryList"
        static let groceryItem = "GroceryItem"
    }

    /// Constants for App Segues
    struct Segues {
        /// Unwind Segue Identifier
        static let unwindToGroceryLists = "unwindToGroceryLists"
        static let unwindToGroceryItems = "unwindToGroceryItems"
    }
}
```

Now build and run your app. We should be able to add grocery lists. We should be able to select grocery lists and add new items to the selected grocery list! And our UI should update now when we create new lists and items. Well, almost. You will notice there is a gap between the grocery list and grocery item. Once we create a new item, it will show up in the `groceryItemsTableView`, but the items label does not update on the `groceryListsTableView`. This is the final puzzle piece before we can call our app complete.

So how can we implement this? We want to keep our app reloading efficiently and there would be no purpose to our unwind segues if we gave up on this now. I want you to think about a solution for this for a minute. This one took me a second to figure out. In fact, I could only think of one way to reload the data if a new item was created. We need to pass a closure to the `GroceryItemsViewController`. This closure will then house the reload logic for the first view controller. Up until now we have also been able to keep our view controllers ignorant of each other. Let's continue this by creating one more protocol.

In the Protocols group, let's create a new file called `ReloadContainer.swift`. In this file we are going to create a protocol that will contain a single property. This property will be a closure that we can execute every time the unwind segue for grocery items fires. The implementation for this new protocol is as follows:

```
protocol ReloadContainer {
    var reloadData: ((Void) -> Void)? { get set }
}
```

Let's go back to our `GroceryItemsViewController` and conform to this protocol. Once we conform to this protocol, we will have access to the closure and we can execute it in the `unwindToGroceryItems` method. The following code block is the `unwindToGroceryItems` method:

```
@IBAction func unwindToGroceryItems(segue: UIStoryboardSegue) {
    groceryItemsTableView?.reloadData()
    reloadData?() // Call the arbitrary closure from the ReloadContainer protocol
}
```

The following code block is the final implementation for the entire `GroceryItemsViewController` class:

```
class GroceryItemsViewController: UIViewController, UITableViewDataSource,
UITableViewDelegate, ViewModelContainer, SelectedGroceryListContainer, ReloadContainer {

    // MARK: - Properties

    var selectedGroceryList: GroceryList? {
        get {
            return viewModel.groceryList
        }
        set {
            viewModel.groceryList = newValue
        }
    }
```

```
    var reloadData: ((Void) -> Void)? = nil

    var viewModel: GroceryItemsViewModel = GroceryItemsViewModel()

    @IBOutlet var groceryItemsTableView: UITableView?

    // MARK: - Segue

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        super.prepare(for: segue, sender: sender)

        if var groceryListContainer = segue.destination as? SelectedGroceryListContainer {
            groceryListContainer.selectedGroceryList = selectedGroceryList
        }
    }

    @IBAction func unwindToGroceryItems(segue: UIStoryboardSegue) {
        groceryItemsTableView?.reloadData()
        reloadData?() // Call the arbitrary closure from the ReloadContainer protocol
    }

    // MARK: - Table View

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return viewModel.groceryItems.count
    }

    func tableView(_ tableView: UITableView, willDisplay cell: UITableViewCell, forRowAt
indexPath: IndexPath) {
        let item = viewModel.groceryItem(at: indexPath)

        cell.textLabel?.text = item.name
        cell.detailTextLabel?.text = "Quantity: \(item.quantity)"
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        return tableView.dequeueReusableCell(withIdentifier: GAConstants.TableCell.
rightDetail, for: indexPath)
    }
}
```

We now have one final piece. We need to give our reload functionality to the `GroceryItemsViewController` from the `GroceryListsViewController`. We are going to add one more `if` block to our prepare-for segue-method on the `GroceryListsViewController`. Since we want the closure to reload our `groceryListTableView`, let's set the closure equal to this method. The following code block is the prepare-for-segue method on the `GroceryListsViewController`. We try to cast the `destination` view controller as a `ReloadContainer` type. If this is successful, we set the `reloadData` closure to the `groceryListTableView.reloadData` method reference.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
    super.prepare(for: segue, sender: sender)

    if var groceryListContainer = segue.destination as? SelectedGroceryListContainer,
        let indexPath = groceryListTableView?.indexPathForSelectedRow {

            viewModel.transferGroceryList(at: indexPath, to: &groceryListContainer)
    }

    if var reloadContainer = segue.destination as? ReloadContainer {
        reloadContainer.reloadData = groceryListTableView?.reloadData
    }
}
```

And that's a wrap. If you build and run your app, you should be able to see your grocery lists and add more. Once you select a grocery list, you will see the items held within the selected list. You can then add more items to the selected grocery list and the entire UI reloads appropriately at the correct time. Next, we have the final wrap up to discuss the ultimate meaning and goals to take away from this book.

Final Wrap Up

Wow! This has been a long road getting here, for me too. I wanted to start this off by thanking you! I really appreciate you following along and not throwing this book away. This has been as much a learning experience for me as it has been for you. So what does this book all mean? What do we want to take away?

Throughout this entire book, I have started threads like architecture, protocol-oriented programming, and testing. I have already wrapped up architecture and testing earlier in this chapter. There is still one loose end, protocol-oriented programming. I think this is the biggest topic of the book. The entire book has been a holistic look at Swift and iOS, but one of the biggest paradigms in Swift is protocol-oriented programming. This paradigm encompasses so much of what it means to develop Swift code. From the ground up, Swift was meant to be safe and efficient. Protocol-oriented programming illustrates these ideas perfectly. Protocols force us to think about the surface of our code. What should be public, what should be optional, should it be restricted to class types or open to values types?

A value type offers us a lot of safety and keeps our code lightweight. In multiple chapters including this one, we saw how we can use value types in our apps. Through the use of protocols, we were able to use a struct for displaying errors to the user and we were able to test this functionality with ease. Another aspect of protocol-oriented programming is the abstraction it offers us. In this last chapter, we were able to keep our view controllers from knowing anything about each other. We used protocols to drive how our data flows and how our UI refreshes without using any concrete types. I think this is a huge win for our app and allows it to scale.

And finally, way back in Chapter 5, I introduced the concept of traits and abilities and the differences between these ideas and interfaces. I believe these distinctions in concepts can help us decide if a protocol should be extended with functionality or not. I believe this is key to understanding what protocol-oriented programming is at its core. Ultimately, I believe to successfully employ protocol-oriented programming requires a shift in thinking. Here are my final thoughts: I believe protocol-oriented programming is an architectural decision, I believe it is a testing strategy, and that it encompasses ***all*** of Swift. Thank you for reading!

Index

A

E. Downey, *Practical Swift*, DOI 10.1007/978-1-4842-2280-5

B

C

D

E, F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U

V

W

X, Y, Z