



Community Experience Distilled

Reactive Programming with Swift

Leverage the power of the Functional Reactive Programming paradigm with Swift to develop robust iOS applications

Cecil Costa

[PACKT] open source 
PUBLISHING

Reactive Programming with Swift

Table of Contents

[Reactive Programming with Swift](#)

[Credits](#)

[About the Author](#)

[About the Reviewer](#)

[www.PacktPub.com](#)

[Why subscribe?](#)

[Free access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Downloading the color images of this book](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Introduction to Reactive Programming](#)

[What is reactive programming?](#)

[The history of reactive programming](#)

[Paradigms - declarative versus imperative](#)

[What is functional programming?](#)

[Choosing reactive programming](#)

[Swift - interactive, safe, and fast](#)

[The ReactiveCocoa project](#)

[ReactiveCocoa extensions](#)

[Migrating to ReactiveCocoa](#)

[The future of reactive programming](#)

[Summary](#)

[2. Installing ReactiveCocoa and Using It with Playground](#)

[The ReactiveCocoa website](#)

[Exploring ReactiveCocoa](#)

[Installing ReactiveCocoa via CocoaPods](#)

[Installing CocoaPods without administrator permission](#)

[Installing CocoaPods with Carthage](#)

[Using Playground](#)

[Summary](#)

[3. Performing UI Events with ReactiveCocoa](#)

[An overview of the project](#)

[Setting up the project](#)

[Creating a validator class](#)

[Validating text fields](#)

[Enabling and disabling the button](#)

[Using UIDatePicker](#)

[Selecting the gender of the user](#)

[Adding more information](#)

[Getting the right input type](#)

[Using bidirectional channels](#)

[Displaying your horoscope](#)

[Summary](#)

[4. Network and Change Propagation](#)

[Overviewing the project](#)

[Setting up the project](#)

[Searching for a movie](#)

[Creating signals](#)

[Handling errors](#)

[Filling in the table view](#)

[Model-View-ViewModel bindings](#)

[Displaying movie posters](#)

[Improving your code for a second scene](#)

[Filling in the movie form](#)

[Implementing the genre signal](#)

[Changing a few details in the first scene](#)

[Summary](#)

[5. Enhance Your Application Using RAC Extensions](#)

[An overview of the project](#)

[Setting up the project and installing extensions](#)

[Mocking up the first scene](#)

[Retrieving information from GPS](#)

[Signaling](#)

[Taking pictures with a camera](#)

[Using gesture recognizers](#)

[Storing pictures](#)

[Saving pictures to the photo library](#)

[Storing coordinates](#)

[Showing coordinates](#)

[Summary](#)

[6. Using the ReactiveCocoa 4 Style](#)

[An overview of the project](#)

[Setting up the project](#)

[Developing the Currency class](#)

[Creating the Currency Manager](#)

[Creating the Product class](#)

[Implementing a shopping cart](#)

[Resuming the ViewController class](#)

[Creating the checkout scene](#)

[Testing the application](#)

[Summary](#)

[7. Testing Your Application](#)

[Checking the expected results](#)

[Creating unit tests](#)

[Using signals for checking the results](#)

[Testing an asynchronous signal](#)

[Testing the UI](#)

[Profiling with Instruments](#)

[Summary](#)

[8. Migrating a Real Application to ReactiveCocoa](#)

[Knowing the application](#)

[Creating a new framework](#)

[Replacing the airplane delegate](#)

[Reorganizing the signals](#)

[Checking the dark side](#)

[Splitting the signal again](#)

[Waiting for 10 seconds](#)

[Reversing the geolocation](#)

[Avoiding repeated calls](#)

[Summary](#)

Reactive Programming with Swift

Reactive Programming with Swift

Copyright © 2016 Packt Publishing

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

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

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

First published: April 2016

Production reference: 1210416

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78588-426-9

www.packtpub.com

Credits

Author	Copy Editor
Cecil Costa	Sonia Cheema
Reviewer	Project Coordinator
Maksim Mikheev	Nidhi Joshi
Commissioning Editor	Proofreader
Neil Alexander Singh	Safis Editing
Acquisition Editor	Indexer
Rahul Nair	Mariammal Chettiar
Content Development Editor	Graphics
Aishwarya Pandere	Disha Haria
Technical Editor	Production Coordinator
Rahul C. Shah	Conidon Miranda

About the Author

Cecil Costa, also known as Eduardo Campos in Latin countries, is a Euro-Brazilian freelance developer who has been learning about computers since getting his first 286 in 1990. From then on, he kept learning about programming languages, computer architecture, and computer science theory.

Learning and teaching are his passions; this is the reason why he worked as a trainer and an author. He has been giving on-site courses for companies such as Ericsson, Roche, TVE (a Spanish television channel), and lots of others. He is also the author of *Swift Cookbook First Edition* and *Swift 2 Blueprints*, both by Packt Publishing. He will soon publish an iOS 10 programming video course.

Nowadays, Cecil Costa teaches through online platforms, helping people from across the world.

In 2008, he founded his own company, Conglomo Limited (<http://www.conglomo.es>), which offers development and training programs both on-site and online.

Throughout his professional career, he has created projects by himself and also worked for different companies from small to big ones, such as IBM, Qualcomm, Spanish Lottery, and DIA%.

He develops a variety of computer languages (such as Swift, C++, Java, Objective-C, JavaScript, Python, and so on) in different environments (iOS, Android, Web, Mac OS X, Linux, Unity, and so on) because he thinks that a good developer needs to learn all kinds of programming languages to open their mind; only after this will they really understand what development is.

Nowadays, Cecil is based in the UK, where he is progressing in his professional career as an iOS team lead.

*I would like to thank Mr. Isaac Newton for discovering his third law:
For every user action there is a ReactiveCocoa reaction.*

I would like to thank Mr. Robert William Bemer for creating the escape key, and my son, Gabriel Campos Oliveira, for bringing happiness into my life.

About the Reviewer

Maksim Mikheev is an experienced iOS developer, who specializes in native development with Swift and Objective-C. He has worked with all major iOS releases starting with iOS 4. He has gathered a solid expertise in all aspects of developing apps for Apple's mobile platform, starting from the idea to the development of the app to publishing the app in the App Store. Maksim has worked in different types of companies—small-scale businesses, start-ups, and large-scale companies. He also owned a web development consultancy company. He developed mobile apps for major Russian companies and government bodies, including a Russian leading consulting company, Strategy Partners Group, and the largest Russian bank, Sberbank.

Maksim is a longtime blogger, writing primarily about mobile development and other IT topics. He is also the host of two successful Russian podcasts concerning different kinds of hobbies. In his free time, Maksim is either reading or programming for one of his pet projects. He is a huge fan of science and follows the latest scientific and technological news and developments. Being a geek, he is always interested in modern gadgets and devices. He tweets a lot about mobile development, IT, and science. You can reach him on Twitter at [@maxmikheev](https://twitter.com/maxmikheev).

He aspires to reach the next level as a software developer, which is why he is constantly working on improving his skills and expertise.

www.PacktPub.com

For support files and downloads related to your book, please visit www.packtpub.com.

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.packtpub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

Apps nowadays are not just sequences of code that are executed synchronously; they are more like a collection of code that is executed asynchronously, making it difficult to follow one order. Reactive Programming (or even better, Functional Reactive Programming) is trending due to it being prepared for this new way of app development. This book will show to you how to use ReactiveCocoa, the most extended Reactive Programming framework for iOS and OS X.

What this book covers

[Chapter 1](#), *Introduction to Reactive Programming*, gives you a brief introduction to what Reactive Programming and Functional Programming are, where they come from, and why you should used them.

[Chapter 2](#), *Installing ReactiveCocoa and Using It with Playground*, shows you three different ways of installing ReactiveCocoa: using the git submodule, CocoaPods, and Carthage. Finally, it also explains how to use it on Playground for performing fast tests.

[Chapter 3](#), *Performing UI Events with ReactiveCocoa*, gives you the first approach of using Reactive Programming. You will see how you can validate a form with ReactiveCocoa.

[Chapter 4](#), *Network and Change Propagation*, starts with the concepts of using asynchronous calls and introduces you to creating your own signals and controlling them.

[Chapter 5](#), *Enhance Your Application Using RAC Extensions*, shows you how to create a small app that uses Reactive Cocoa extensions. A good feature of this chapter is that it uses different versions of ReactiveCocoa and shows you how to deal with them.

[Chapter 6](#), *Using the ReactiveCocoa 4 Style*, shows you how to use ReactiveCocoa in a safer way. Also, you will learn how Signals and SignalProducers replaced RACSignal.

[Chapter 7](#), *Testing Your Application*, teaches you how to create unit tests, debug an app, and use Instruments with ReactiveCocoa.

[Chapter 8](#), *Migrating a Real Application to ReactiveCocoa*, shows you how to convert an app that was developed without Reactive Programming into an app that uses it.

What you need for this book

This book was written using Xcode 7.2 and ReactiveCocoa 4; however, its code should work with newer versions.

Only a few parts use a physical device; therefore, having one is optional but an advantage.

Who this book is for

This book is for Swift developers who want to start making more powerful and efficient applications. You need a basic understanding of Swift to follow along. This book takes a first-principles approach to what Reactive Programming is and how you can start implementing it in your future iOS applications.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "This can include unsubscribing the observer using the `deinit` method."

A block of code is set as follows:

```
func changeSingletonValue() {
    let singleton = Singleton.instance()
    singleton.setValue("New Value", forKey: "MyKey")
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
private func signalForQuery(query:String) -> RACSignal{
    return RACSignal.createSignal{
        { (subscriber:RACSubscriber!) -> RACDisposable! in
            })
    }
}
```

Any command-line input or output is written as follows:

```
cd ReactiveCocoa
script/bootstrap
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Now, recompile your project, and click on **Playground**."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

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

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

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

Downloading the color images of this book

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

Errata

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

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

Piracy

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

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

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

Questions

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

Chapter 1. Introduction to Reactive Programming

Every day, you'll find something new to explore in terms of computer programming languages. These can be in the form of a new programming language, framework, methodology, or even a new paradigm. You can work with all of these to solve a problem or improve a development process. Reactive programming is a new paradigm that is no exception to this. The philosophy behind this new paradigm is that an application needs to focus on what to do and not how to do it. This chapter introduces what this paradigm is about, where it came from, and what ReactiveCocoa is.

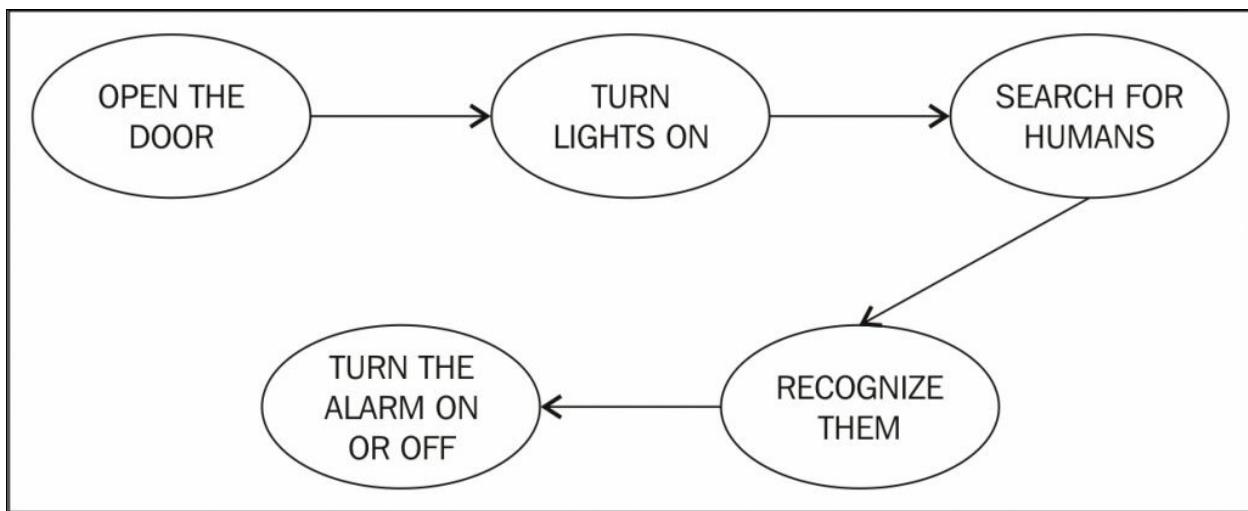
In this chapter we will cover:

- What is reactive programming?
- The history of reactive programming
- Paradigms – imperative versus declarative
- What is functional programming?
- Choosing reactive programming
- Swift – interactive, safe, and fast
- ReactiveCocoa extensions
- Migrating to ReactiveCocoa
- The future of reactive programming

What is reactive programming?

Reactive programming is a computer programming paradigm that's based on the propagation of change. What does this mean? A short answer to this can be that when something changes (the value of a text field, variable value, and so on), other objects that depend on this value must be notified and react according to the new value.

To visualize it, let's take a look at an example. Imagine that you have a home automation application. Your application must react according to the device's state; for example, if the front door is open, your application must know that someone is coming in. Therefore, it can then switch on the light. When the light is switched on (it doesn't matter whether it is done because someone has opened the door or manually switched it on), the security camera must search for the person in the room and try to recognize them. If the person has not been recognized, the alarm should ring, but if the person is recognized, the alarm must react by shutting down. The following diagram explains this idea to you in a more visual way:



Keep in mind that everything works like a chain. When the lights are on, your application must also react by changing its switch state and icon, the camera

preview button must be enabled, and so on.

Once you have understood the previous example, you will have a basic idea of what reactive programming is; however, you may think that this application can also be used without including reactive programming. You can, of course, do this, but the problem lies in how you do it.

The traditional style of programming is based on how to do things. This means that you have to create many observers, making development more difficult and fragile as you have to be aware of every part of the chain, especially when adding new requirements.

Reactive programming is based on a different way of development. Basically, you just have to create rules and when anything happens, the rules are to be followed. It is different from the traditional way as you have to think about how everything works and then create rules. Continuing with the previous example, in the traditional way of programming, you have to worry about whether the light switch icon is on the screen or not. Therefore, when you return to the screen where this icon shows up, you have to check its status using the `viewDidAppear` method. With reactive programming, you usually don't have to worry about this.

When developing on iOS, you have to remember a few patterns, such as MVC, KVO, and notifications, making the developer's life a bit complex as it is necessary to remember every detail of each pattern. This can include unsubscribing the observer using the `deinit` method. Reactive programming tries to merge these patterns and make everything work in the same way.

In a nutshell, an application can have a lot of states. With traditional programming, you have to be aware of all of these states, making development very complex and hard to follow. Reactive programming creates a new approach where the developer doesn't need to be worried about the complexity of states and their internal details; they just need to know the main idea behind the application flow. Here's one definition of reactive programming: it involves programming around data flows and the propagation of change.

This propagation of change is mainly based on two concepts, **signals**

and **streams**, which are going to be explained in the upcoming chapters.

The history of reactive programming

Since the time that computers were invented, human beings have started looking for better ways of programming with the understanding that this would solve their problems. Some of the traditional problems in computer programming include performance, maintenance, development speed, and so on.

When some programming languages (mainly object-oriented programming languages) were being created with abstraction as their main feature, at that time it was very questionable because some people couldn't accept that hiding the internal implementation would create a better software. Nowadays, developers and companies understand that abstraction is important as most of the time, there is no sense developing at a very low-level. Besides this, a function or method implementation can be changed completely without changing its header or class definition.

In the 90s, design patterns were presented to the world in a book called *Design Patterns*, written by the "Gang of Four" (*Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*) by *Pearson*. The idea behind these patterns was to create templates that could speed up development and make software maintenance easier. An example of a famous design pattern is (Model-View-Controller (**MVC**)).

The MVC pattern could also mean **Massive View Controller**; therefore, a new pattern had to be created. Microsoft created a new pattern called **Model-View-ViewModel (MVVM)** in the late noughties. This pattern was based on binding data with a view.

After MVVM was created, Microsoft created a framework called (Reactive Extensions (**Rx**)). The idea behind this framework was to make it possible to perform reactive programming with .NET mainly by propagating it with the changes made using LINQ. This is the reason why searching for reactive programming on the Internet might take you to some websites that use the following logo:



Nowadays, you can develop software using reactive programming in different languages; for example, you can use ReactiveX and Rx.PHP if you develop on PHP, RxPY if you develop on Python, or even Reflex if you develop on Perl.

Tip

Some frameworks are based on Rx and some are not. Before using a specific reactive framework for your programming language, it is a good idea to check its features and other users' opinions. Remember that reactive programming is a relatively new programming concept; therefore, some frameworks are still quite young.

Paradigms - declarative versus imperative

When we talk about programming, it is mostly understood that we are talking about the imperative paradigm, where a developer tells the machine how to perform a task step by step. C, for example, is an imperative programming language as you have to write every instruction in order to retrieve the desired result.

The declarative paradigm, in contrast, tells you that you have to code what you want in order to make the machine do what you want it to do, but it doesn't matter how you do this. You can look at SQL as a declarative programming language as you don't really know how data is stored. All you have to do is write a statement, such as the `SELECT` statement, and then you receive your result.

For a more visual result, let's compare some code: one using the imperative paradigm and the other one using the declarative paradigm. Imagine that you want to apply VAT to the prices that are in an array using imperative programming. To do this, you may have to iterate through every element in the array and remember some details; for example, an array in Swift starts at position `0`, and the `count` property is beyond the bounds of the array. Have a look at the following code:

```
let prices: [Double] = [10, 8.1, 20.15]
var pricesVAT: [Double] = [Double]()

for i in 0..
```

Now, let's create the equivalent code to this using the declarative programming approach. Basically, we are going to use the `map` function to make the final code more readable:

```
var pricesVAT = [10, 8.1, 20.15].map { (element) -> Double in
```

```
    return element * 1.2
}
```

In the preceding code, we just describe what we want to do, rather than how to do it, how an array works, and so on.

If you've understood the difference between the code, remember that reactive programming is a branch of declarative programming. This means that it can be simple for a new developer or a bit hard for a developer who has been working with imperative programming for a long time.

Note

If you explore declarative programming and reactive programming further, you will see that between them there's another category called **dataflow**.

This paradigm is based on values that change over time. A typical example of this is the comparison of variables with a spreadsheet. Let's assume we have the following code:

```
var a = 9
var b = 15
var c = a + b
```

As you see, the value of variable `c` is `24`. What happens to the value of variable `c` if we change the value of `a` or `b`? The answer, as you know, is: it remains the same. Let's pretend that our application has a shopping cart: variables `a` and `b` represent the prices of two products and `c` is the amount to be paid. Variables `a` and `b` can change their value over time: this can be due to a change in product, discount voucher, or some other reason. In such a case, we have to remember to update the value of variable `c`, which can make development a bit fragile.

Now, let's visualize this sample on a spreadsheet. Let's add value `9` to cell `A1` and `15` to cell `B1`. These cells, therefore, represent variables `a` and `b` from our previous code. Cell `C1` has a formula that represents the sum of `A1` with `B1`, something that looks like this: $C1=A1+B1$. Any changes made in `A1` or `B1` will automatically update the result of `C1`, and if any other cells reference `C1`, they

will also be updated. The following screenshot shows this example in a visual way:

	A	B	C
1	9	15	24
2			
3			
4			
5			

Any changes on A1 or B1 will change C1

The diagram illustrates a data flow from cells A1 and B1 to cell C1. Any changes made to either A1 or B1 will result in an update to C1.

After comparing the first code with this spreadsheet example, you can now understand what dataflow is all about. Today's applications change values very frequently over a period of time. This can be because a device has changed its state (it's lost a network connection, for example), some information was received from the network, an animation has finished, and so on. Each of these events may change the application's status and update the UI.

Some people think that declarative programming has a lower rate of performance than imperative programming, but the reality is a bit different. Let's perform some benchmarking to check which one is faster. Create a new, single view project, and add the following code to the `viewDidLoad` method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let array1:[Double] = [9,15.5, 3, 18, 7.9, 5.5, 2.2,
1.89, 995, 123.3, 4, 6.2, 12.12, 7,45, 6.61, 7.1, 2.9, 3.5,
52.1, 90, 82.2]
    var array2:[Double]

    let start = CFbsoluteTimeGetCurrent()
    array2 = [Double]()
    for i in 0..
```

```

        array2.append(element)
    }
    print(CFAbsoluteTimeGetCurrent() - start)
}

```

Execute this code, and take note of the execution time when it's in imperative mode. On a MacBook Pro with a 2.6 GHz i5 processor, it took 0.000124990940093994 seconds. Now, update the code to be a declarative one by replacing your current code with the highlighted one:

```

override func viewDidLoad() {
    super.viewDidLoad()
    let array1:[Double] = [9,15.5, 3, 18, 7.9, 5.5, 2.2,
1.89, 995, 123.3, 4, 6.2, 12.12, 7,45, 6.61, 7.1, 2.9, 3.5,
52.1, 90, 82.2]
    var array2:[Double]

    let start = CFAbsoluteTimeGetCurrent()
array2 = array1.map({ (element) -> Double in
        return element * 3.3
    )

    print(CFAbsoluteTimeGetCurrent() - start)
}

```

Execute the code again using the same simulator, and take note of the execution time. Using the machine and simulator we saw previously, it took 0.0000889897346496582 seconds.

Why? The reason for this is that a declarative function might be optimized by the compiler or built-in libraries, while when using an imperative function, you have to optimize yourself.

This reasoning behind this is simple logic: if you compare the execution of a SQL `SELECT` statement with code that's built from scratch you can think about which one has a better performance. How long does the `SELECT` statement take to filter 50,000 records from a table of 1,000,000 records? How long would it take if you had to write an equivalent filter using imperative programming?

What is functional programming?

Once you read up on reactive programming, you will also find something called **functional reactive programming (FRP)**. Functional programming is a declarative programming paradigm that avoids the changing of variable status and mutable data.

Functional programming has a few features, such as first-class functions (you can send functions as arguments, like you do in Haskell), immutable data, reducing, pipelining, recursing, currying (functions with multiple parameters), and monads. However, some authors prefer describing it as programming with functions that have no side effects, which means that a function doesn't change any data outside of it.

Let's take a look at a sample of nonfunctional programming for a better understanding. Have a look at the following code, and pay attention to how it changes an external value:

```
func changeSingletonValue() {  
    let singleton = Singleton.instance()  
    singleton.setValue("New Value", forKey: "MyKey")  
}
```

This code is a side effect, and this is what you have to try to avoid when using functional programming; try to use more deterministic functions that don't use global variables.

Using methods, such as `map` (as we saw in a previous sample), `reduce`, `sort`, `filter`, and other samples of functional programming, ensures that you send functions as arguments and don't use external stuff.

To sum up, there are frameworks that are reactive but not based on functional programming, and there are other frameworks, such as ReactiveCocoa, that are reactive and functional.

Choosing reactive programming

One difficult question you may ask yourself is, "When do I have to use reactive programming?" Basically, there is no right or wrong answer to this question; however, there are some questions that you have to ask yourself. If your application (or framework) needs to synchronize asynchronous calls, if you would like to merge different patterns (MVC, KVO, notifications, and so on) into one, or if your application needs to check a lot of rules before performing some operations, reactive programming is probably right for you.

Different industries have started using reactive programming, such as the robotics industry, the health industry, the tourism industry, the gaming industry, and even traditional mobile applications that need to use different asynchronous features. These may include GPS and other sensors.

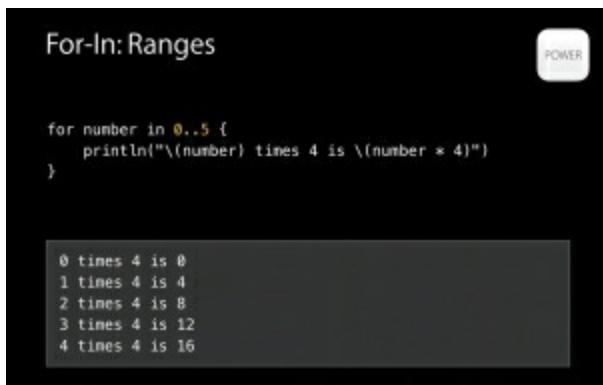
Another good news about using reactive programming, and in this case, ReactiveCocoa, is that it is ready for failures. When using imperative languages, such as C or Pascal, the developer needs to check for errors very frequently and changing the application status can be a hard task. **Object-oriented programming (OOP)** usually works with exception propagation, which is a better approach, but it can sometimes also be hard to use. ReactiveCocoa usually considers two calls when trying to perform a task: the first one deals with how to react in case of success, and the second deals with how to react in case of an error.

When should you not use reactive programming? This is hard to answer. There may not be a particular reason for not using reactive programming. Only if it conflicts with any other library that's used on your project should you not consider using it.

Swift - interactive, safe, and fast

The Swift programming language was announced by Apple in the middle of 2014. The idea was to create a new programming language that would replace Objective-C. This way, it could have a cleaner and safer language without the limitation of retaining C's language syntax. This new programming language needed to be compatible with Objective-C as they would basically share the same space for a long time; however, Swift borrows some features from other programming languages (such as Haskell), allowing you to program more like a functional paradigm.

Swift is now open source; this means that more people can improve this language by adding new features, fixing bugs, and optimizing it. One point of view is that it is good that this language is continuously evolving, but on the other hand, the syntax also changes frequently. If you check the first **Worldwide Developers Conference (WWDC)** 2014 when Swift was first announced, you will see a few features, such as the following ones, that are out of date:



A screenshot of an iPhone displaying a Swift playground. The title bar says "For-In: Ranges". The code in the playground is:

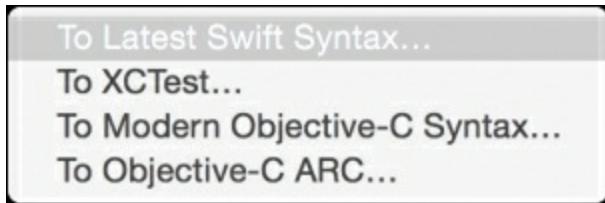
```
for number in 0..5 {
    println("\(number) times 4 is \(number * 4)")
}
```

The output of the code is displayed in a dark grey box at the bottom:

```
0 times 4 is 0
1 times 4 is 4
2 times 4 is 8
3 times 4 is 12
4 times 4 is 16
```

Some people say that Apple will stop supporting Objective-C soon or later, and other people say that there will be features that are available only for Swift. This actually already holds true for a protocol-oriented feature, for example. It doesn't matter whether these rumors are true or not. It is very clear that Apple will be focusing more on Swift than Objective-C from now on.

Keeping this fact in mind, take care when updating Xcode to ensure that the version of ReactiveCocoa is compatible with the newest version of Swift. In case of compiler errors, remember that you can always try to update your project and the ReactiveCocoa framework by clicking on the **Edit** menu, opening the **Convert** option, and then selecting the **To Latest Swift Syntax...** submenu, as shown in the following screenshot:

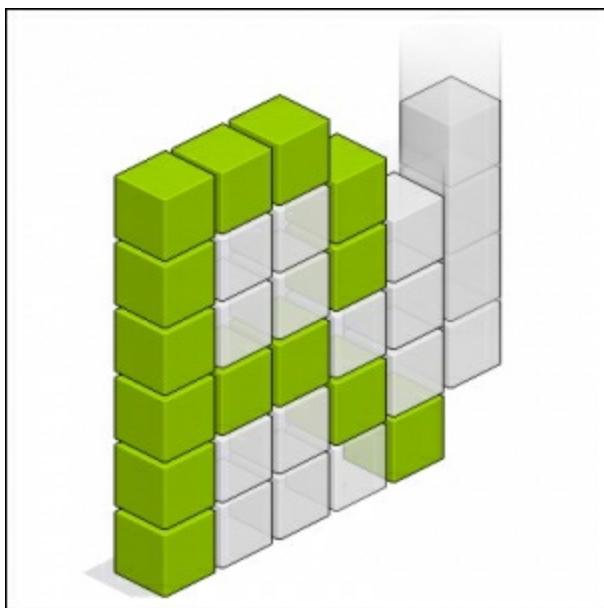


If you have problems with ReactiveCocoa and the new version of Swift, first check whether there is a new version of the framework before trying to fix it by yourself.

There are different ways to program with Swift; therefore, it is considered a multiparadigm programming language. FRP with Swift requires knowledge about closures more than using a target-action pattern, which is based on the selector of object methods.

The ReactiveCocoa project

Since the time reactive programming has increased in popularity, a variety of languages have started their own framework that allows programmers to develop using reactive programming; Swift is no exception to this. This project was created by Josh Abernathy and Justin Spahr-Summers, two GitHub employees who realized the GitHub client on Mac had many network calls and they could solve some bugs related to it in a different way. The authors of ReactiveCocoa define it as a Cocoa framework that helps compose and transform streams of values. This project, also called RAC, has the following image as its logo:



This project was started in March 2012 by Josh Abernathy. As Swift did not exist at the time, it was developed for Objective-C. After the popularity of the Swift programming language, a new branch was created and ReactiveCocoa was then ported to Swift. When the process was completed, version 3.0 of ReactiveCocoa was released with Swift as its main language.

If you still have projects that use Objective-C and you would like to use

ReactiveCocoa, don't worry: you can still do this as Swift and Objective-C can exchange calls between each other. You can read more about this bridging at

<https://github.com/ReactiveCocoa/ReactiveCocoa/blob/master/Documentation/>

An alternative to using ReactiveCocoa is RxSwift. The main difference between them is that RxSwift was a port of Microsoft Rx to the Swift programming language. ReactiveCocoa was inspired by Rx; however, it is an independent project, and the idea does not revolve around the porting of Rx.

ReactiveCocoa extensions

Is ReactiveCocoa alone? No, it is not. Today, there are many frameworks that are based on ReactiveCocoa; thus, you don't have to worry about whether the required frameworks are compatible with ReactiveCocoa. They are probably third-party extensions of your frameworks already.

In [Chapter 5, Enhance Your Application Using RAC Extensions](#), you will learn how to use **ReactiveCoreData (RCD)**, a framework that combines Core Data with ReactiveCocoa; however, this is not the only framework that's based on the idea behind ReactiveCocoa. Here are a few frameworks that are based on ReactiveCocoa.

- **ReactiveCocoaLayout (RCL)**: This describes a way of creating layouts with code in a reactive way. This project is still in the alpha phase, and some people have complained that everything you do is based on code, not on **Interface Builder (IB)**. However, for people who think that code is everything, it might be a great solution. Check out this project at <https://github.com/ReactiveCocoa/ReactiveCocoaLayout>.
- **ReactiveAnimation**: This is a framework to create animations for iOS and OS X. You will find a good example of it for Mac. For more information on this framework, visit <https://github.com/ReactiveCocoa/ReactiveAnimation>.
- **ReactiveCocoaIO**: This is a file manager that's based on reactive programming. This framework replaces `NSFileManager`, and because it works in an asynchronous way, it is supposed to be faster. For more information on this framework, take a look at <https://github.com/ReactiveCocoa/ReactiveCocoaIO>.

Migrating to ReactiveCocoa

A question you might ask yourself is, "How painful is it to migrate to ReactiveCocoa?" This, of course, depends on the size of your project, its complexity, structure, and so on; however, there are some rules that you can follow to make it easier.

The process of converting an application code into an application that's developed with ReactiveCocoa is called **RACify**. This process has no restrictive rule. What you have are some steps that you can follow to create the right type of signal or stream.

We are going to learn more about this process in [Chapter 7, Testing Your Application](#), where we are going to take a framework and convert it into one that uses ReactiveCocoa.

The future of reactive programming

Reactive programming is a relatively new paradigm; therefore, not every project uses it. However, it is now being accepted by a lot of new real-world projects.

Imagine that day by day, applications are becoming more complex for different reasons: smartphones are more powerful now, they come with a lot of sensors, and working with asynchronous calls is becoming increasingly common.

When smartphones first started gaining popularity with the first iPhone and a few Android devices, their applications were very simple, and some sensors weren't too accurate. Nowadays, however, companies have started investing in mobile applications, and you can, for example, request a taxi with your smartphone, make payments by replacing your credit card, use your phone as a boarding pass, and so on.

So, what's coming up next in terms of smartphones? You'll see a lot more features, and something that has just started gaining popularity is the **Internet of Things (IoT)**, a concept that allows you to control home devices through your phone, as represented in the following figure:



This new concept, as you can imagine, works very asynchronously. You can try

it out for yourself using the HomeKit framework. This type of development can be controlled through a simple application; however, when it progresses into becoming a complex application, you might need a better methodology in place. Reactive programming is considered the perfect framework for developing this kind of an application.

Health sensors may also be the next generation of new sensors on devices. The Apple Watch already comes with a heartbeat sensor, which submits information to your phone, and you can access it through the HealthKit. This information can also be accessed asynchronously, especially if you can compare it with data on the Internet.

A health application is a good example of an application that can be used with reactive programming. Every time, a user's health information changes, the application reacts in a different way, displaying accurate diagnostics:



What about learning? Or better yet, e-learning? Believe it or not, there will come a time when we will have to explain to our children that when we were young, we had to physically go to school. Imagine an application that teaches you how to play the piano at home with other remote students. In such a case, many asynchronous calls can be made: when you press or release a key on your piano, signals are sent to your device while it synchronizes with other students and the individual scores.

Summary

In this chapter, we learned what functional reactive programming is, where it comes from, and the advantages of using it. Now, you can appreciate that this new paradigm has had a long history, but it started out as a mature concept not too long ago.

This type of programming paradigm is trending these days as it fits the needs of the applications of today. This also makes the jobs of developers a lot easier easier.

Now that you have a better idea about reactive programming and ReactiveCocoa, be prepared because we are going to start working with the two. Don't worry if you think that some concepts look different because actually they are.

Chapter 2. Installing ReactiveCocoa and Using It with Playground

Sometimes software acts like a do-it-yourself product: you've just bought something that you are very excited about using as soon as you can, but before doing this, you have to put together some stuff first. At this point, you might want to use the ReactiveCocoa framework; however, it is first necessary to install it.

There are a few ways of installing ReactiveCocoa; in this chapter, we are going to learn about some of them. We are also going to test the ReactiveCocoa framework using Playground; this will, therefore, give us a basic idea of how it works.

In this chapter, we will cover the following topics:

- Cloning ReactiveCocoa framework
- Installing ReactiveCocoa via CocoaPods
- Installing CocoaPods with Carthage
- Using ReactiveCocoa with Playground

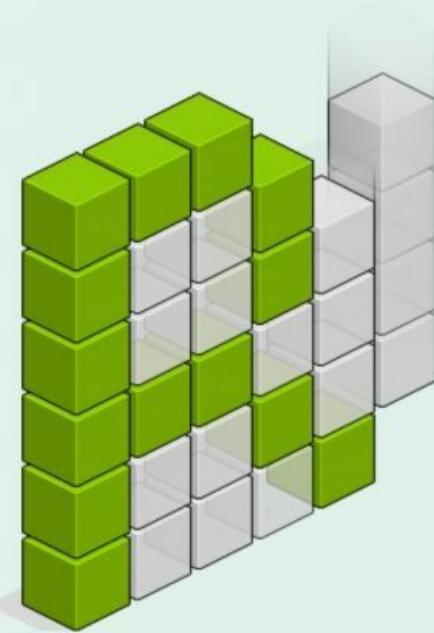
The ReactiveCocoa website

When you install a new framework, the first step is to install it and then investigate its official website. This is usually the best place to start looking for documentation or any other starter information. Many projects have websites along with their own domains, and this is the reason that you will find a few links leading to <http://reactivecocoa.io>. This is a website without very much information, as you can see in the following screenshot:



ReactiveCocoa

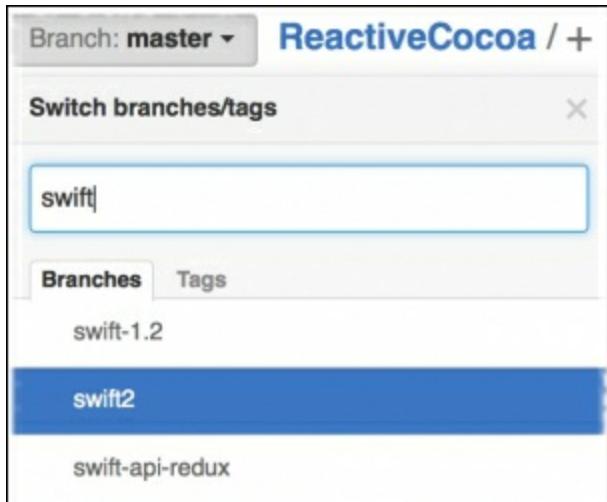
ReactiveCocoa is an implementation of functional reactive programming – for more information about FRP, see our philosophy page.



This website may not be what you expect it to be, but if you click on **philosophy**, you will be introduced to reactive programming. Once you have read the philosophy behind reactive programming, you will be willing for something to download; in this case, you will have to go to another website hosted by GitHub.

Open your web browser and type <https://github.com/ReactiveCocoa/ReactiveCocoa>. If you have already downloaded something from GitHub, you may know about the layout of the website, but if you don't, let's take a quick tour of it.

First, take a look at the **Branch** drop-down menu that is located before the files, click on it to list the available branches, and then search for a specific branch. Usually, you will have to select the **master** branch. However, sometimes, if you are working with a feature that is in progress, such as a new Swift version, you may need to change it by clicking on this drop-down menu, filtering it by the branch name and selecting the desired branch. The following screenshot shows an example of how you can switch branches:



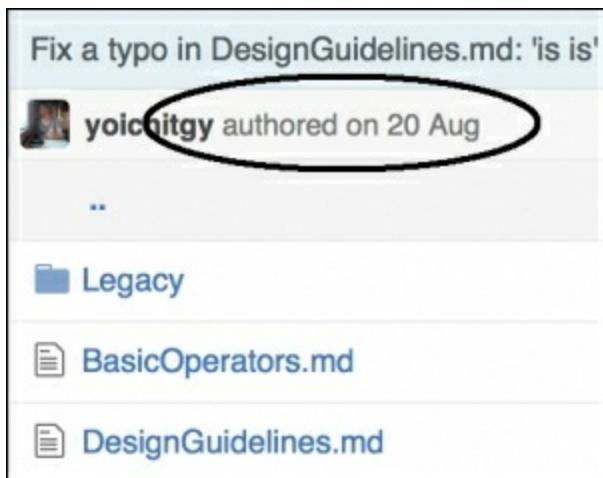
Scroll down to take a look at the website, and you will then see the initial documentation, originally from a file called `README.md`, which gives you a

basic idea of how to use this framework. It is important to read this file as it will sometimes give you crucial information of the current version; for example, if you read this file when version 3.0 of ReactiveCocoa was released, you would have noticed that this file would give you an explanation of how it was to be used with Swift instead of Objective-C.

Tip

The Swift programming language is still changing very frequently. To know more, visit <https://developer.apple.com/swift/blog/?id=14>,
<https://developer.apple.com/swift/blog/?id=22>,
<https://developer.apple.com/swift/blog/?id=29>, and
https://en.wikipedia.org/wiki/Swift_%28programming_language%29.

In the directory called `Documentation`, as its name suggests, there are files that represent the ReactiveCocoa documentation, which should be checked in case of any doubts or to learn about new concepts. It is a good practice to check whether there are any updates by taking a look at the recent developments in this folder, located above the files, as highlighted in the following screenshot:



Back to the root directory. On the right-hand side, you will see some components that allow you to download the current source code, something that

is very common when you want a project without its history. Here, you have a text field with a URL and two buttons under it, as demonstrated in the following screenshot:



The text field contains the URL used to check the source code with its history using Git. The **Clone in Desktop** button does the same thing, but you must have a desktop application installed in your computer.

Note

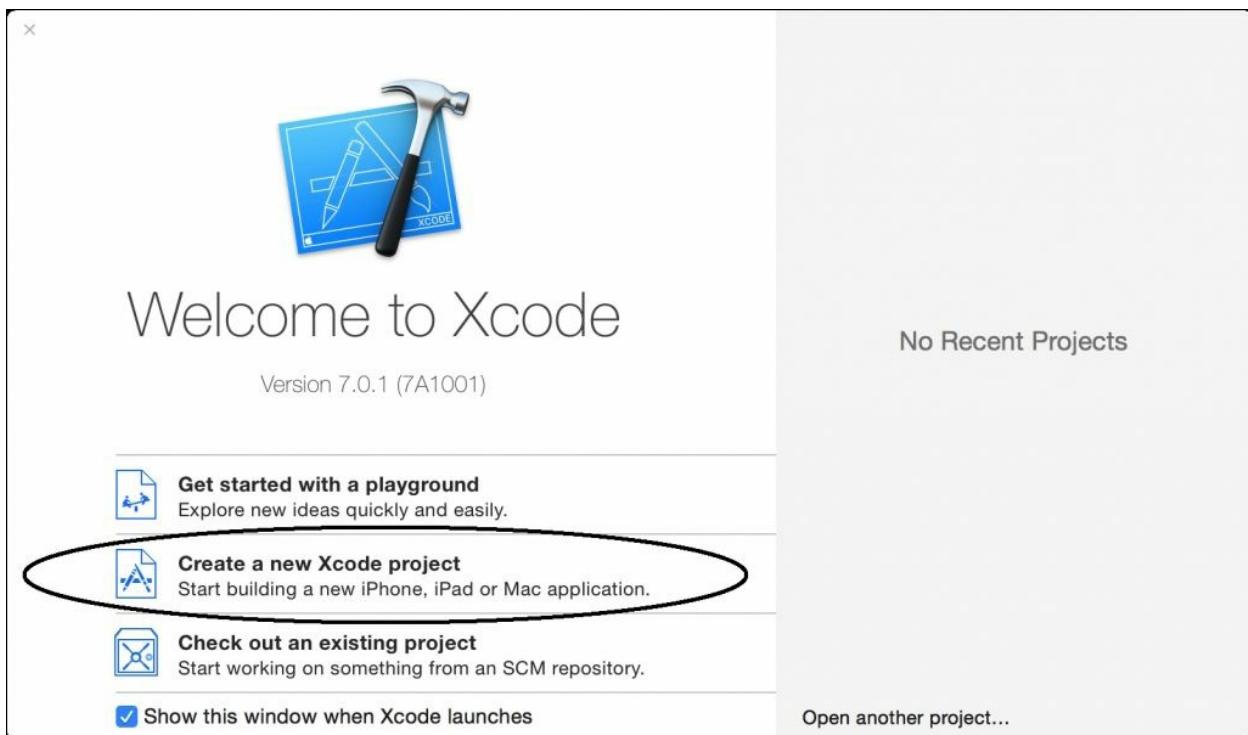
There are many desktop applications that you can use to control Git's repositories; however, if you are not contributing to the ReactiveCocoa project, you may not need to worry about these programs.

The third button, **Download ZIP**, is to download the source code without any GitHub links. This option is very common when you just want to download a project without any links to GitHub; however, the ReactiveCocoa installation needs this link. Therefore, it is not common to use this option for ReactiveCocoa.

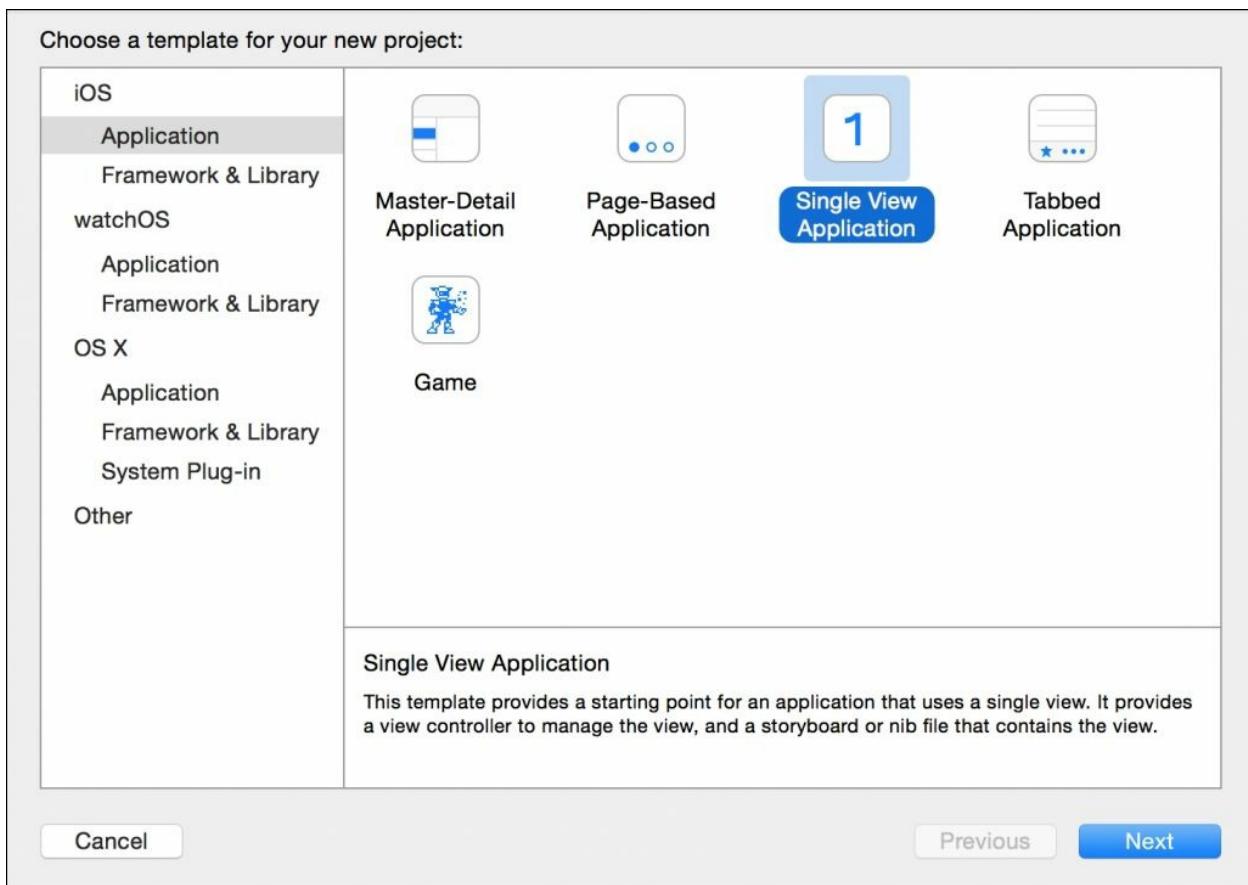
Exploring ReactiveCocoa

In this section, we will learn how to install ReactiveCocoa by checking out its GitHub repository. To do this, we will first create a simple project in Xcode and then add ReactiveCocoa to it.

Open Xcode, and select the **Create a new Xcode project** option, as shown in the following screenshot:



On the left-hand side of the window, as shown in the following screenshot, select the **Application** subsection under the **iOS** section, and choose **Single View Application** as the project template:

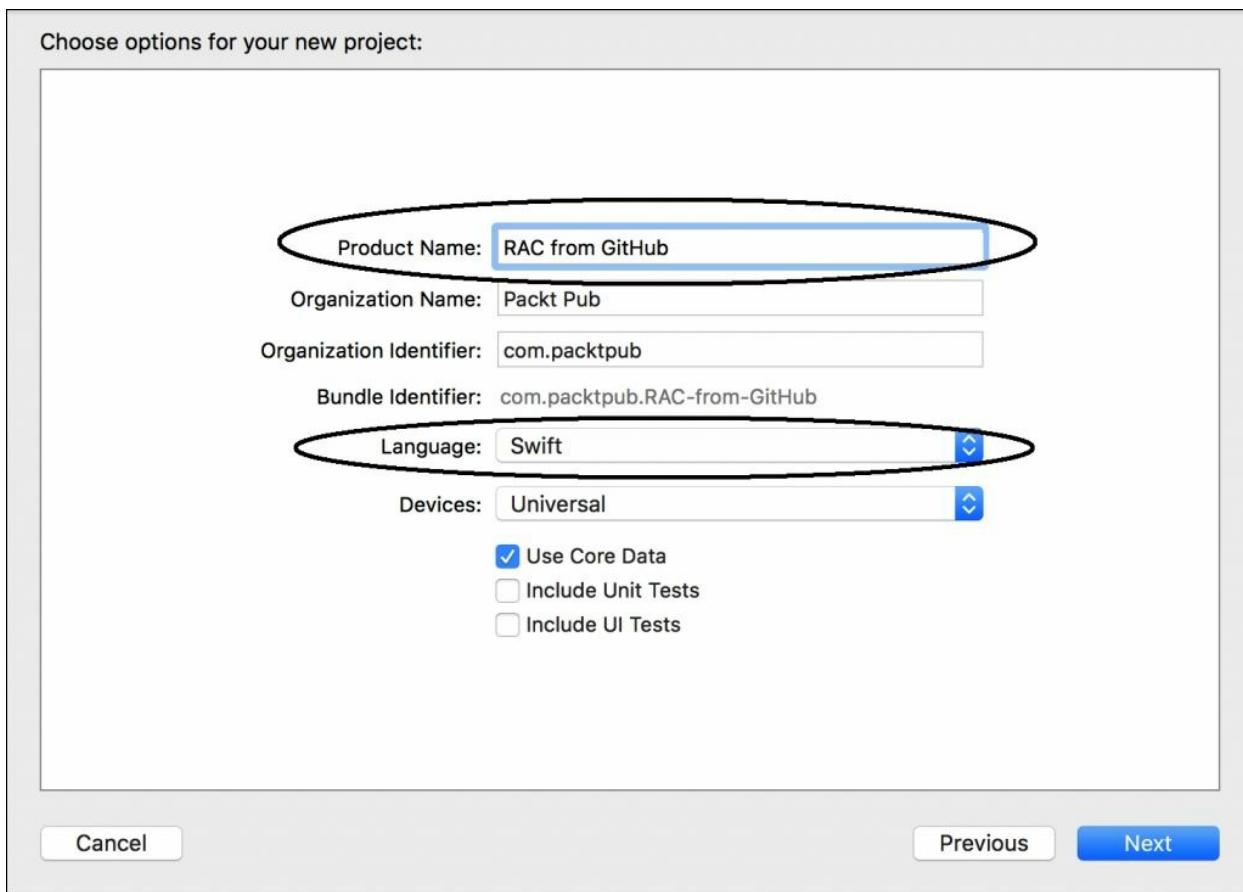


From now on, every time a new project needs to be created and nothing else is specified, we will assume that it is **Single View Application**.

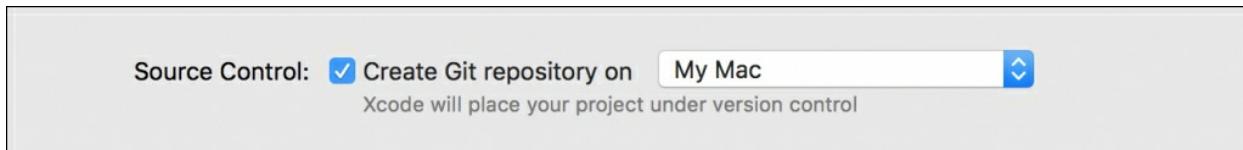
Tip

You can use ReactiveCocoa to develop with OS X; however, this book will focus only on iOS development.

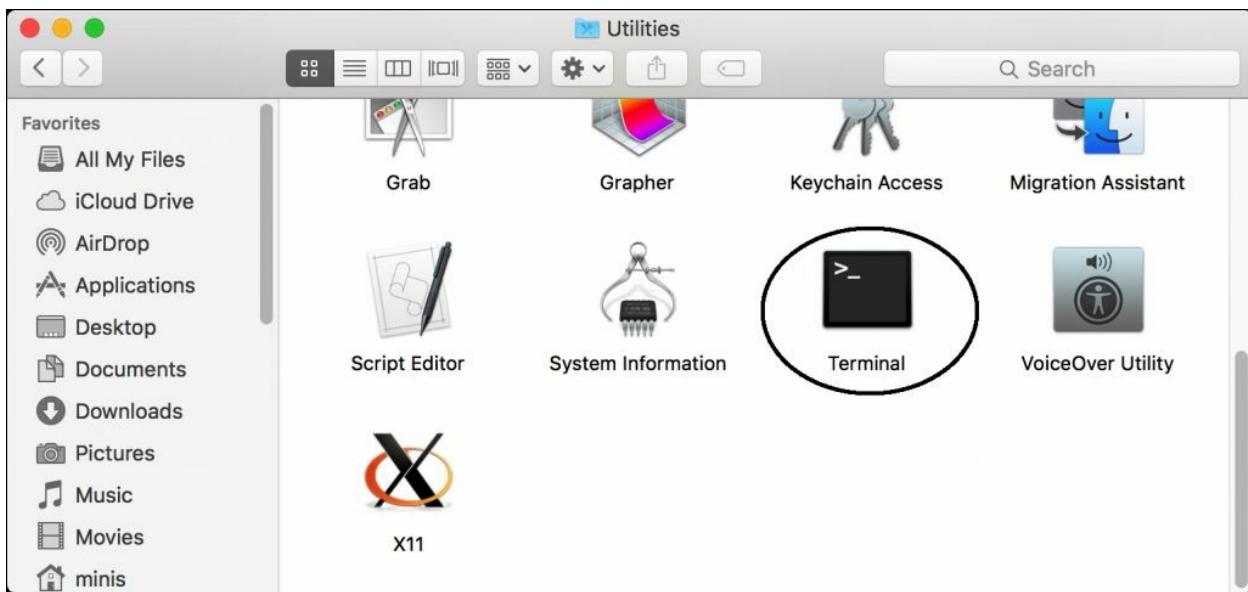
In this dialog box, set **Product Name** to `RAC` from GitHub, and make sure that the main **Language** is set as **Swift**:



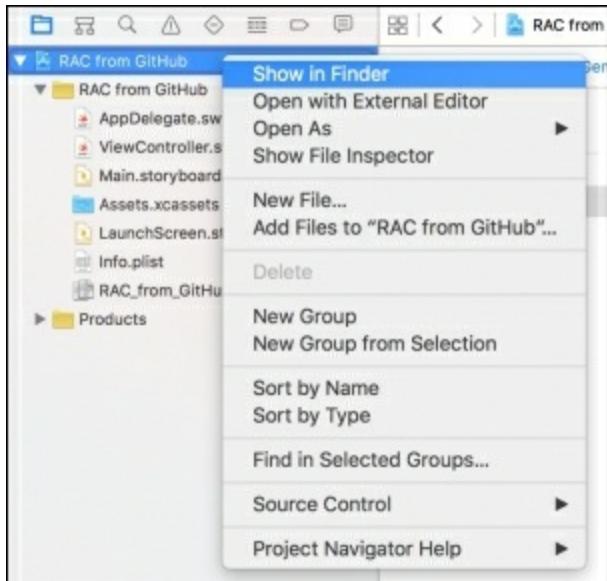
Select a folder in order to save your project, and make sure that the option to create the Git repository is checked:



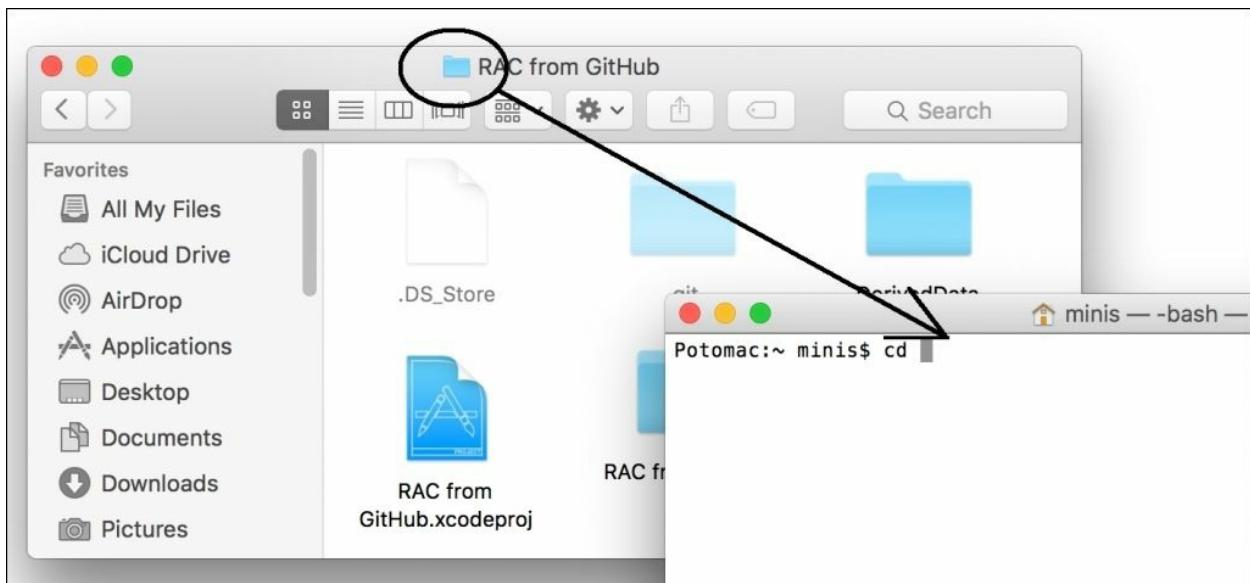
Once the project is open, we'll open an old friend called **Terminal**. Open the finder window, and use the *command + shift + U* combination key to go to the **Utilities** folder. Scroll down until you are able to see the **Terminal** icon and double-click on it:



After opening the terminal, we need to switch to our project folder. If you are already familiar with the terminal, you can change it using the `cd` command, followed by your project folder. If you are not used to the terminal or you just want to change the directory through a simple method, just type `cd` and a whitespace (do not press *enter* yet) and return to Xcode. In **Project Navigator**, right-click on the project, and when the menu appears, select **Show in Finder**, as demonstrated in the following screenshot:



Now, you have to click on the folder icon that is shown on the top bar and drag it to your terminal window. It will complete your `cd` command, and you can then press *enter*:



At this point, we are ready to take a look at the ReactiveCocoa project. As we

already have a Git repository, we don't have to clone the framework's repository, we just need to create a submodule of it. To do this, we will use the `git` command `submodule` with the `add` option for `-b` to specify the branch followed by ReactiveCocoa's URL. In this case, we will use the `master` branch, but be aware that you might need another branch; for example, when Swift 2 was released, you had to use the `swift2` branch. Our final command will be like this:

```
git submodule add -b master  
https://github.com/ReactiveCocoa/ReactiveCocoa
```

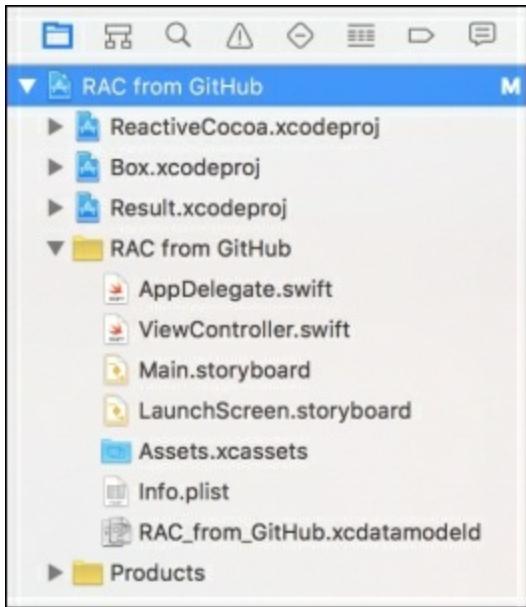
You will see a few lines displaying the progress on the screen; it may take a few seconds for this command to complete as it needs to download some files from the Internet. When the command finishes, you can check whether a new folder, called `ReactiveCocoa`, has been created. Navigate to this folder by typing `cd ReactiveCocoa`. Check whether you have changed the `script/bootstrap` directory type, which is a script that checks for download dependencies. Again, you will have to wait a few seconds.

When the prompt returns, open the current folder with the `open .` command (the dot character belongs to the command). A finder window will appear; here, you will only have to drag the `ReactiveCocoa.xcodeproj` file into **Xcode Project Navigator**.

Note

If you are working with workspaces, you should move/drag the `.xcodeproj` files to the workspace rather than your project.

Once you've done this, you will have to repeat the operation for two more files. Return to the terminal window, type `open Carthage/Checkouts/Box`, drag the `Box.xcodeproj` file to your project, type `open Carthage/Checkouts/Result/`, and drag the `Result.xcodeproj` file into your Xcode project. The final result of the project should be similar to the following screenshot:

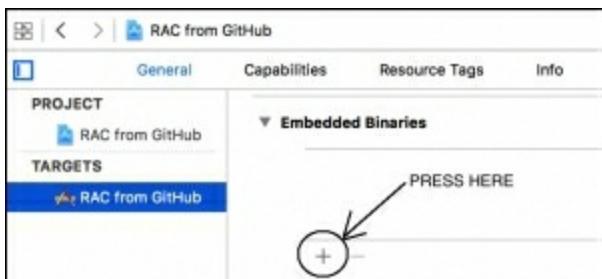


As you can see, the project file has an alphabet **M** attached to it as it was modified but no other file was marked with any change.

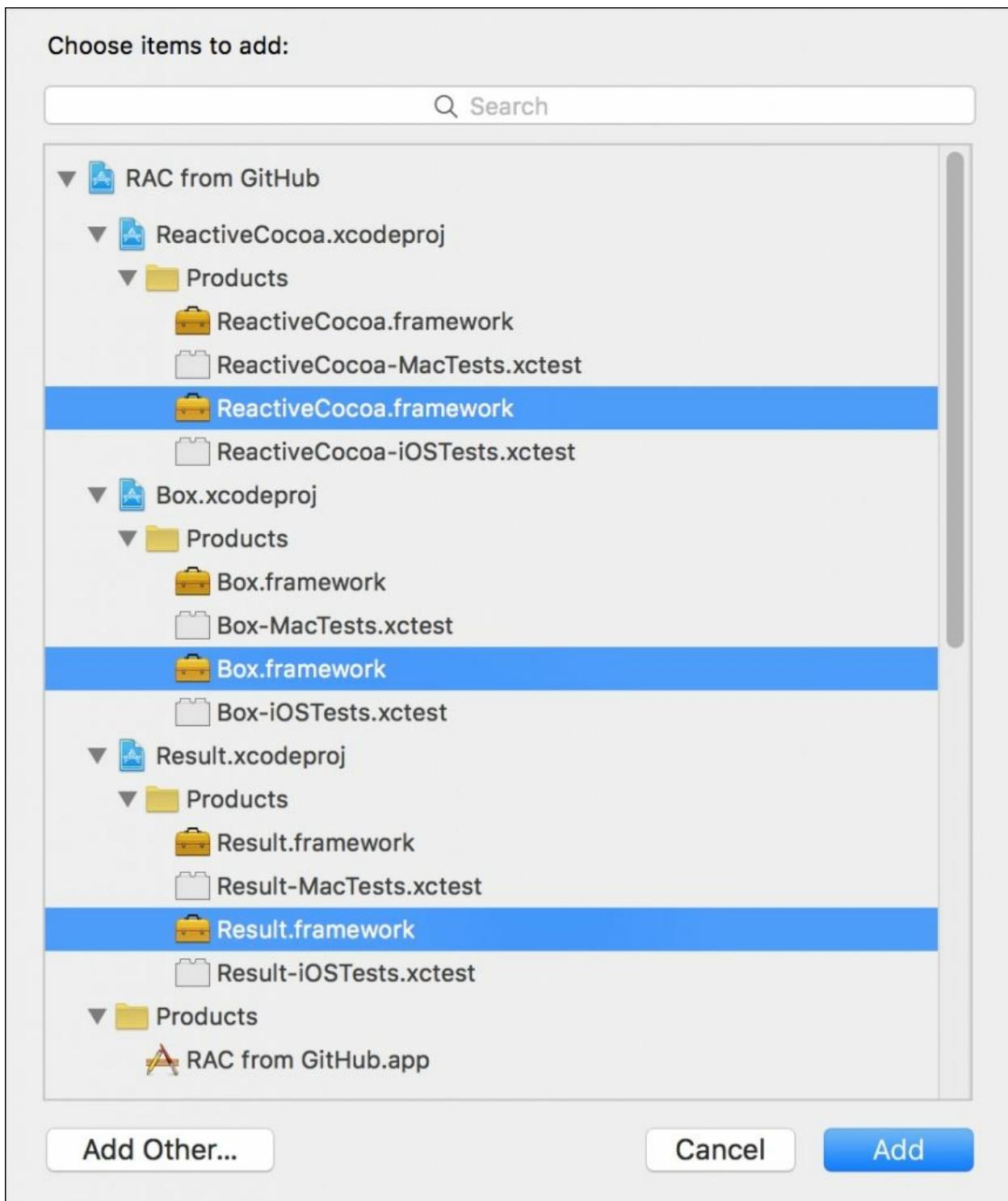
Tip

The **Box** framework was removed in the Swift 2 branch; you will probably need to ignore the steps that mention it.

Go to the your **Project Navigator**, click on the **RAC from GitHub** project, click on the **General** tab, scroll down to the **Embedded Binaries** section, and click on the plus (+) sign that is located under it:



A dialog will appear asking you to choose the frameworks; select `ReactiveCocoa.framework`, `Box.framework`, and `Result.framework`. They are duplicated because one of them is the Mac version and the other one is the iOS version. The iOS version is usually the second one, but bear in mind that it may not be the same in different versions. So, you have to choose one, and if it doesn't work, replace it with another.



Add Other...

Cancel

Add

This should be enough. There is only one more step to be performed in the case of developing only with Objective-C. This includes going to the building

settings of your project and ensuring that the **Embedded Content Contains Swift Code** option is set to **YES** (nowadays, this is the value by default).

Now, it is time to test our application! It looks like everything is working already, but we need to make sure of this. Go to your storyboard, and add the only scene that we have with `UITextField` on top of the screen and `UILabel` under it. Add the constraints that you think are necessary, and open **Assistant Editor** with *Option + command + ,* (comma). Connect the text field to a property called `textField` and the label to a property called `label`. This will be the final code:

```
@IBOutlet weak var textField: UITextField!
@IBOutlet weak var label: UILabel!
```

If you want, change the text field placeholder on its **Attribute Inspector** to `Type something here....` This will, therefore, make this simple application more user friendly.

Go to the `ViewController.swift` file, and start importing the `ReactiveCocoa` framework using the following code.

```
import ReactiveCocoa
```

Now, we only need to add a simple code in the `viewDidLoad` method to make sure that we are able to use the `ReactiveCocoa` framework. Don't worry about the meaning of this code, we will explain it in the upcoming chapters; for now, just add the highlighted code in your `viewDidLoad` method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    textField.rac_textSignal().subscribeNext { text in
        self.label.text = text as? String
    }
}
```

The code is now complete. Press play or *command + R* (build and run) if you like shortcuts, and type something in the text field. The final result shows that the text will be copied onto the label:



Great, now ReactiveCocoa is installed. However, this is not the only way that you can install it. In the next section, you will learn about a very popular way to use CocoaPods.

Installing ReactiveCocoa via CocoaPods

During the last few years, every programming language has had its own package control system, such as PHP with PEAR, Perl with CPAN, Ruby with Gem, and Lua with Rocks. The reason for this is that some packages (such as libraries, frameworks, and so on) have dependencies, these dependencies may also have other dependencies, and so on. Besides this, if you are using a certain version of a library, the dependencies must be in a version that the library is compatible with.

For these reasons, a project called CocoaPods was created. This program allows you to easily download a framework with its dependencies, and this is an alternative way of installing ReactiveCocoa.

Note

Although you can install ReactiveCocoa with CocoaPods, it is not officially supported by the ReactiveCocoa team.

If you have never installed CocoaPods on your machine, you have to start by installing it. Firstly, you need administration permission to do this; if you are using a standard user, you have to switch to an administrator account. There is another way to install it without administrator permission, which will be explained afterwards.

Open a finder window, use the *command + shift + U* combination to open the Utilities folder, and then open the terminal as we did earlier. Now, just type the following command to install CocoaPods:

```
sudo gem install cocoapods
```

This command will prompt your password, as `sudo` is a Unix command. When you type your password, no character will echo for security reasons.

Installing CocoaPods without administrator permission

If for any reason, you can't use an administrator's account, you can specify a directory for its installation; but remember, the CocoaPods program will only be installed for the current user, other users will need to repeat the operation.

Note

When the first version of OS X 10.11 (El Capitan) was released, there was an issue when installing it with the `sudo` command; to solve this problem, you had to install it by specifying a user path.

Open a finder window, go to the `Utilities` folder, and open the terminal. Create a folder, called `Gems`, in your `HOME` directory using the following command:

```
mkdir -p $HOME/Gems
```

After this, create a bash variable to let the `gem` program know the location to store the software it has downloaded. This variable must be called `GEM_HOME`, and you have to set its value using the following command:

```
export GEM_HOME=$HOME/Gems
```

Now, you can use the `gem` command without using `sudo`:

```
gem install cocoapods
```

You now have CocoaPods installed but only for the current user. The command line has no idea that the `Gems` directory or any of its subdirectories has any program installed. We have to let the command line know that programs inside the `Gems/bin` directory must be executed if called. To do this, let's add a line to the `.profile` file as this file is executed every time we open the terminal by typing the following commands in it:

```
echo 'export PATH=$PATH:$HOME/Gems/bin' >> $HOME/.profile
chmod 700 $HOME/.profile
```

Next, exit the terminal and open it again.

Tip

Another way of installing a standard user is using the `--user-install` option. Here, you don't have to specify a destination directory, and it will also update the `.profile` file for you. This method is also acceptable; however, it will install into a folder called `.gem`, which is supposed to be hidden.

At this point, it doesn't matter if you've installed CocoaPods as an administrator or a standard user. You just have to type `pod` in the terminal, and you'll see an output similar to what is shown in the following screenshot:

```
[Potomac:~ minis$ pod
Usage:

$ pod COMMAND

CocoaPods, the Cocoa library package manager.

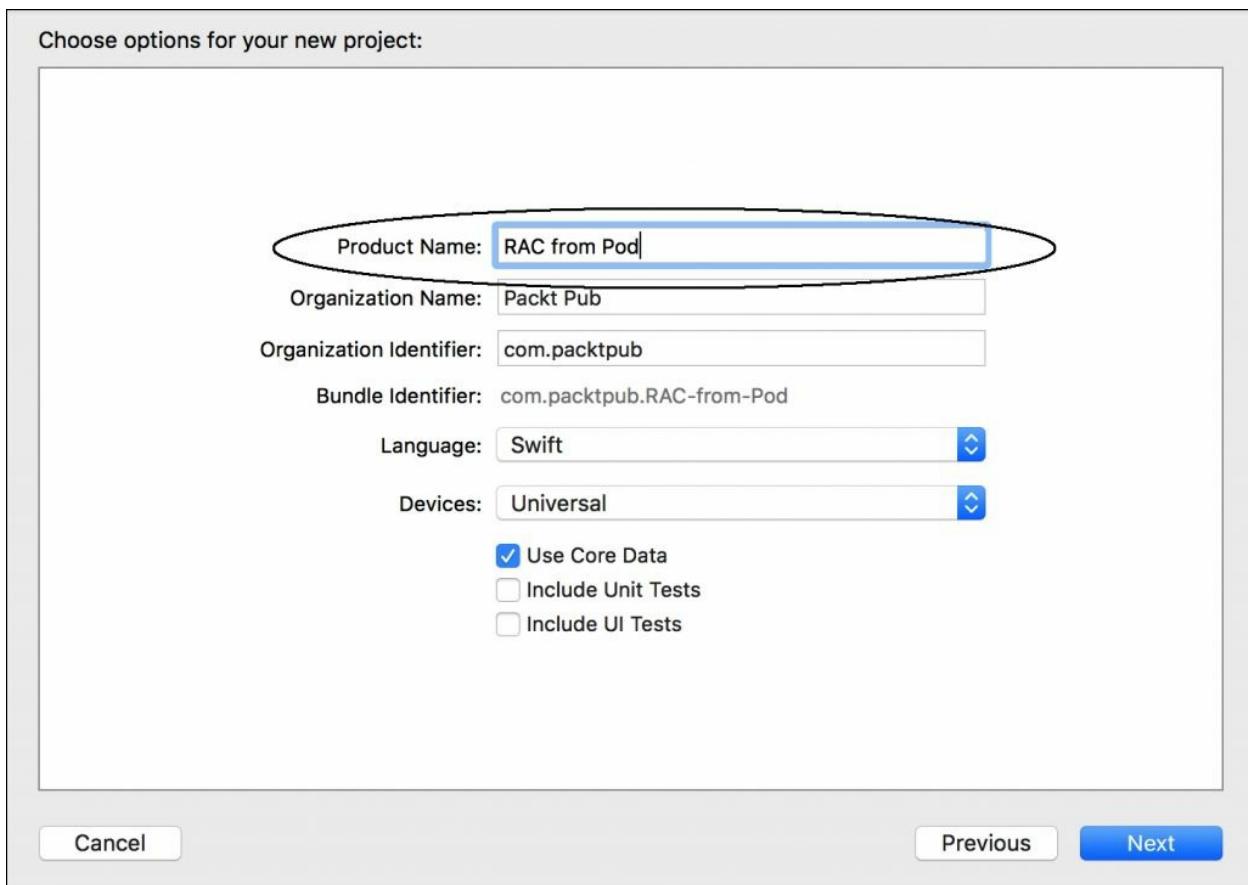
Commands:

+ init      Generate a Podfile for the current directory.
+ install   Install project dependencies to Podfile.lock versions
+ ipc       Inter-process communication
+ lib        Develop pods
+ list      List pods
+ outdated  Show outdated project dependencies
+ plugins   Show available CocoaPods plugins
+ repo      Manage spec-repositories
+ search    Searches for pods
+ setup     Setup the CocoaPods environment
+ spec      Manage pod specs
+ trunk    Interact with the CocoaPods API (e.g. publishing new specs)
+ try      Try a Pod!
+ update   Update outdated project dependencies and create new Podfile.lock

Options:

--silent    Show nothing
--version   Show the version of the tool
--verbose   Show more debugging information
--no-ansi   Show output without ANSI codes
--help      Show help banner of specified command
Potomac:~ minis$ ]
```

Now, open Xcode, create a new **Single View Application**, and call it `RAC` from `Pod`. The only difference between the previous process of creating projects and this one is **Project Name**, as you can see in the following sample dialog:



Return to the terminal, and switch over to your project directory as discussed earlier. We have to create a file called `Podfile`, and use the **TextEdit** editor to edit it using the following commands:

```
touch Podfile
open -e Podfile
```

Once TextEdit is open, you have to add this content to it:

```
platform :ios, '8.0'
use_frameworks!

target 'RAC from Pod' do
  pod 'ReactiveCocoa', '4.0.4-alpha-4'
```

end

Save the file using *command + S* and close TextEdit with *command + Q*. Now, use the `pod install` command to install it. You will see a message at the end such as `From now on use `RAC from Pod.xcworkspace``. Once you've received the message, close your Xcode project and open the new workspace. Here, you can repeat the same test we performed in our first project.

Installing CocoaPods with Carthage

Carthage is another package system that is developed for OS X and iOS. Some people prefer Carthage over CocoaPods as it is simpler and less intrusive, while others may prefer CocoaPods due to its completeness and flexibility. This debate is out of the scope of this book. The important part that you have to consider is that ReactiveCocoa doesn't support CocoaPods officially. Rather, it supports Carthage, and this reason will be explained in this section.

Carthage can be installed by downloading the latest version of it from <https://github.com/Carthage/Carthage/releases/>. When the file with the `.pkg` extension has been downloaded, you just have to double-click on it and follow the instructions. You will need administrator permission to do this.

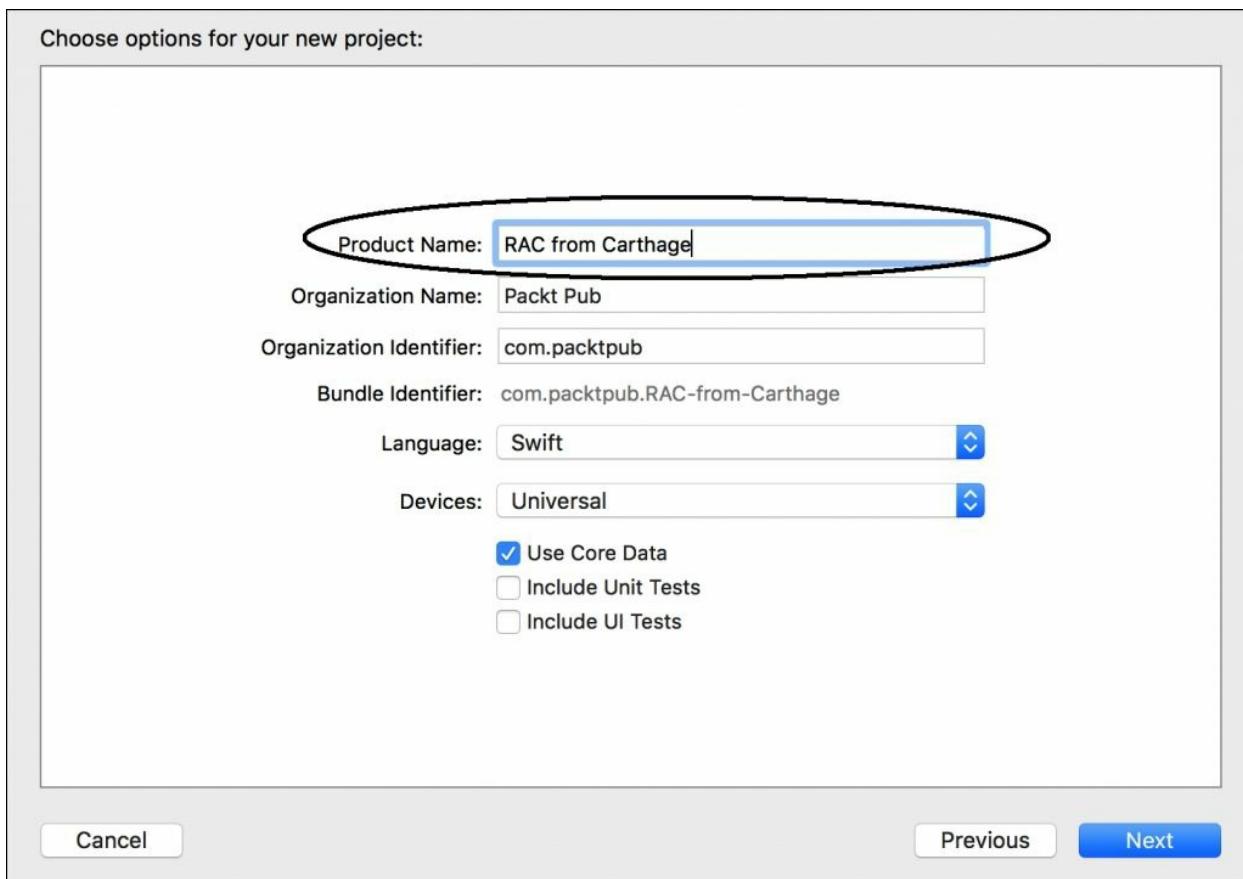
Another way of doing this is using **Homebrew**, another package manager for OS X. If you don't have `brew` installed in your computer, you can install it opening the terminal and typing the following command:

```
ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You will now get a `Password` prompt. To install `brew` as the administrator, just feed the password into this prompt. Wait for a few seconds until the installation is done. Now, just type the following command and `brew` will install Carthage for you:

```
brew install carthage
```

Great! Carthage is installed! We now have to use it in a project. Open Xcode, and create a new **Single View Application** called `RAC` from Carthage:



Return to the terminal, and use the `cd` command, as we learned earlier in this chapter, to switch to your project directory.

Here, we have to create a file called `Cartfile`, and specify the framework that we want to download and compile. You can do this using the following commands:

```
touch Cartfile
open -e Cartfile
```

TextEdit will open with an empty file. The syntax for this file is bit easy. First, you have to tell the repository about the source of your framework. As GitHub is currently the only framework supported by Carthage, you only have to write `github`. After this, you have to specify the path of the framework or library

that you want to download; in this case, we have to specify "ReactiveCocoa/ReactiveCocoa". Finally, you can optionally specify the version that you want to download; for example, if you want the version for Swift 2, you just have to write `swift2`. This will be the final line of code:

```
github "ReactiveCocoa/ReactiveCocoa" "swift2"
```

Save the file using *command* + *S*, and close TextEdit using *command* + *Q*. In the terminal, type the following command to download the ReactiveCocoa framework:

```
carthage update
```

This will take a few seconds to complete. When it finishes, you will see a folder called `Carthage`, with a subfolder called `Checkouts` inside it. Here, you can open the `ReactiveCocoa` folder with the `open Carthage/Checkouts/ReactiveCocoa` command, and drag the `ReactiveCocoa.xcodeproj` file into your project. Do the same for the Result framework.

Go to Xcode, click on your project, select the application target, scroll down to `Embedded Binaries`, click on the plus (+) sign, and select the `ReactiveCocoa` and `Result` frameworks.

Now, repeat the same process that we performed in the first example by adding a text field and label to the scene and placing the same code on the View Controller. Press play and test the application and ... voilà! You will see that `ReactiveCocoa` has been installed again.

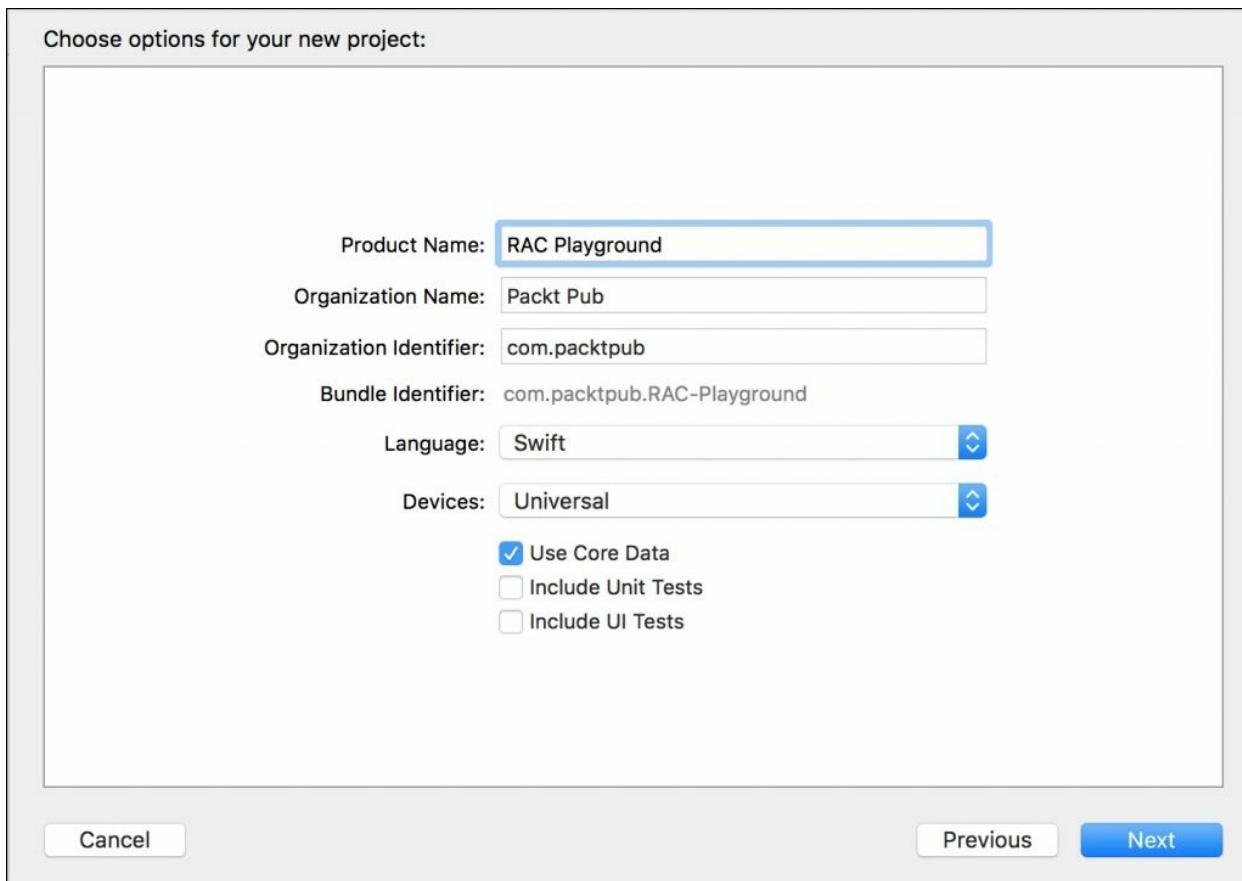
Using Playground

Since Swift was released, Xcode comes with a feature called Playground. This is used to test Swift code in real time in a way that allows you to see the results without the need to execute the whole application.

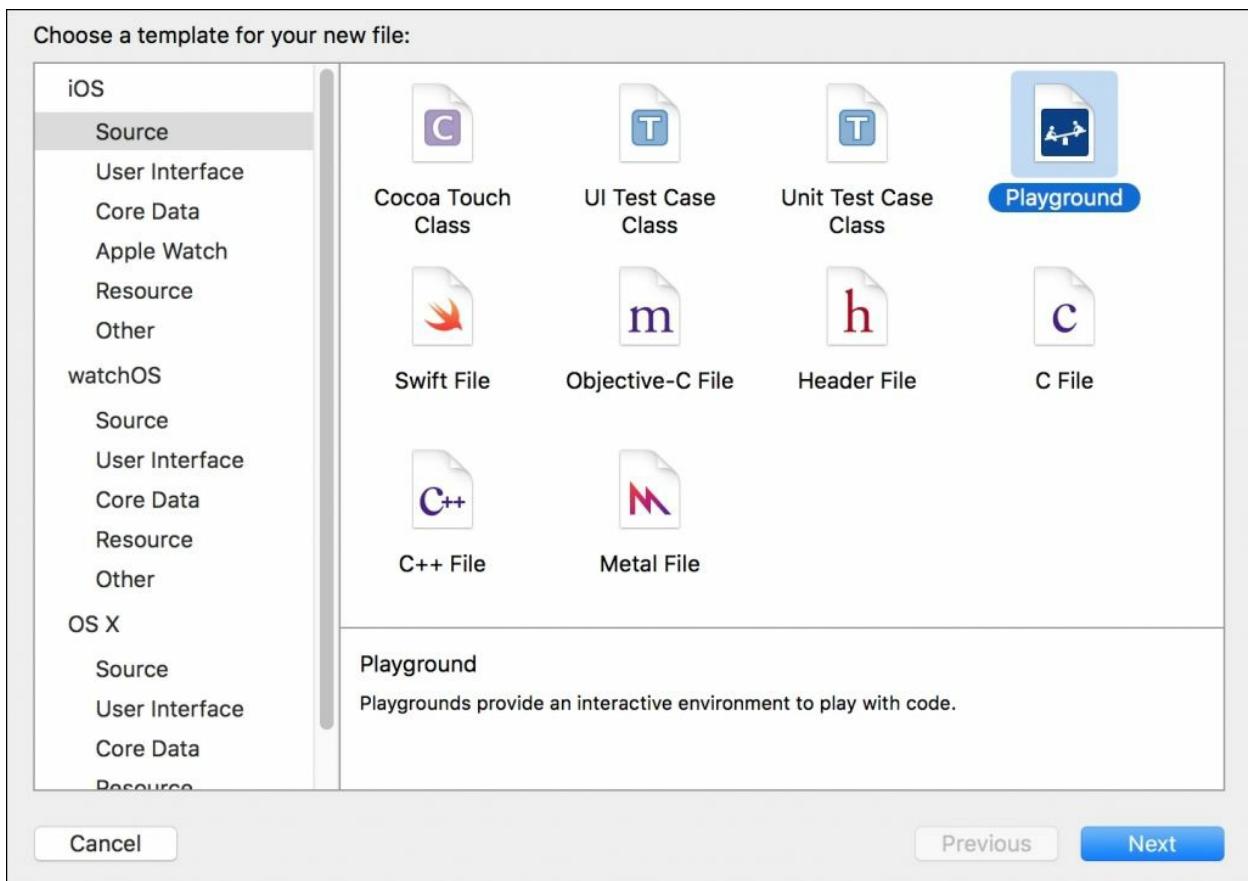
The advantage of using Playground is that it is much faster to test than if you create new applications or unit tests when you have small tasks, such as testing the result of a signal or mutable property.

In this section, we are going to use Playground to test the ReactiveCocoa framework.

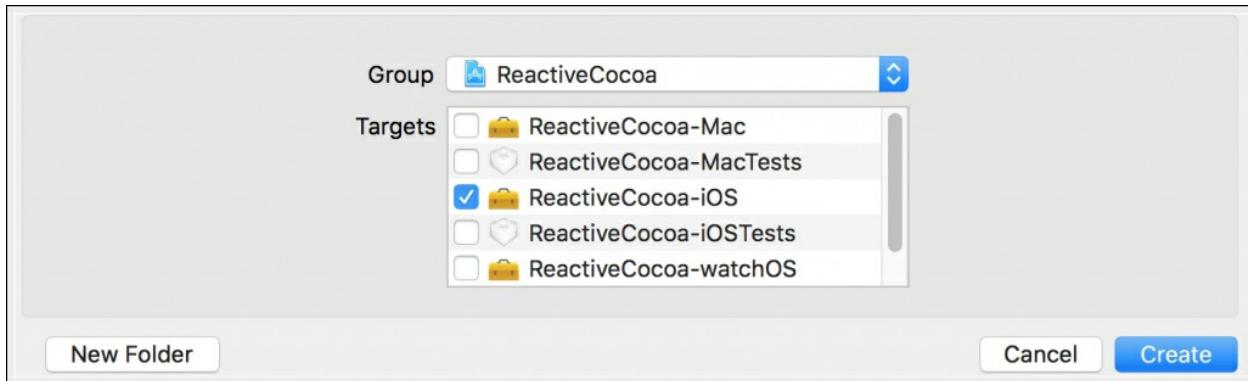
Start by creating a new **Single View Application** called `RAC Playground`. As usual, check whether **Swift** is the main programming language for this project:



Choose your favorite way of installing ReactiveCocoa and then install it (time to practice!). Once everything is done, select the ReactiveCocoa project from **Project Navigator** (not your application), and add a playground to it by using the *command + N* shortcut in order to add a new file. When the dialog requesting you to choose a template appears, select **Playground**, as demonstrated in the following screenshot:

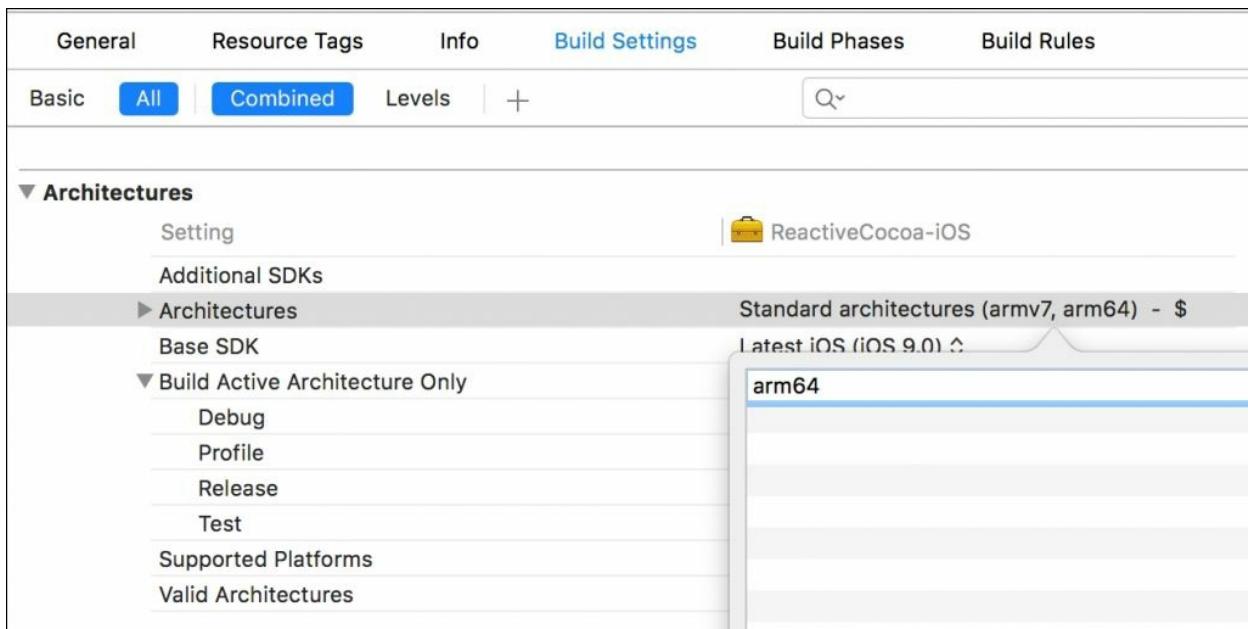


When the name is asked, call it `Reactive Playground`, and then save it. It is very important to check the iOS framework in the **Save** dialog.



According to your Xcode version, if you start importing ReactiveCocoa into your playground, you will see that Playground doesn't recognize it as a framework.

Click on the ReactiveCocoa project in **Project Navigator**, select the **ReactiveCocoa-iOS** target, click on the **Build Settings** tab, and then change the **Architectures** record to **arm64**, as shown in the following screenshot:

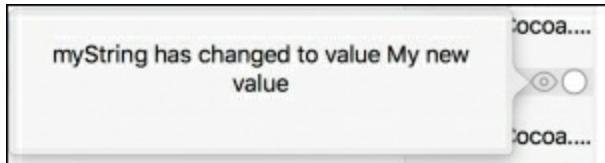


Now, recompile your project, and click on **Playground**. Replace the sample code with the following one:

```
import ReactiveCocoa

let myString = MutableProperty<String>("")
myString.producer.startWithNext({ (text) -> () in
    print("Result")
    print("myString has changed to value \(text)")
})
myString.value = "My new value"
```

In the second line of the `print` statement move your mouse pointer to the right until it reaches the eye icon, which is also called **Quick Look**. Click on it, and you will see that the text printed contains your new value:



Congratulations, now you can test ReactiveCocoa with Playground!

Summary

In this chapter, we learned about the different ways of installing ReactiveCocoa in a project. You took a look at a project as a submodule of your Git repository. By doing this, you can apply the latest changes made by the ReactiveCocoa team.

You also learned to use CocoaPods. This is a very common method for installing different frameworks and controlling the version used by your project. If your project uses frameworks that are already being installed via CocoaPods, it might be a good idea to install ReactiveCocoa this way.

There is also another way of installing ReactiveCocoa: by using Carthage. This is very similar to CocoaPod, and it is officially supported by the ReactiveCocoa team.

In each case that we looked at, we had to use the terminal. However, once it is installed, you need not use the terminal again for ReactiveCocoa's purposes; however, it is recommended that you know how to use the terminal since other frameworks require it.

Finally, we took a look at a small example in order to test the ReactiveCocoa framework using Playground. This way, you could perform small tests before coding the application.

In the next chapter, we are going to work with a sample application and learn the basics of ReactiveCocoa.

Chapter 3. Performing UI Events with ReactiveCocoa

Programming is something that can be done in different ways. Someone who programs only with an assembly language (also called a dinosaur) may be very impressed when they start learning a procedural language such as C. They are most likely to devise a new way of performing their tasks. A person who only programs with a procedural language may also have the same reaction when they start learning object-oriented programming. In this chapter, we are going to experience this as well by switching to the reactive paradigm.

In this chapter, you will learn how to create a simple application, but instead of using it in the traditional way, we will use the reactive programming paradigm using signals rather than selectors. You will experience a different way of programming tasks, and you will realize how simple this can be.

In this chapter, we will cover the following topics:

- Using signals
- Combining signals
- Extending RACSignal
- Using channels

An overview of the project

The idea behind this project is to take a look at today's horoscope of a user, and to do this, we will need some user information. We can't check the user's horoscope until every mandatory field is filled in correctly.

This application is only made for iPhones in portrait mode in order to make it easier for us to design it with the help of AutoLayout. If you want to use this application on an iPad or in landscape mode, feel free to. For this application, we are not going to focus on doing this as we don't want to waste time with stuff that is not relevant.

Note

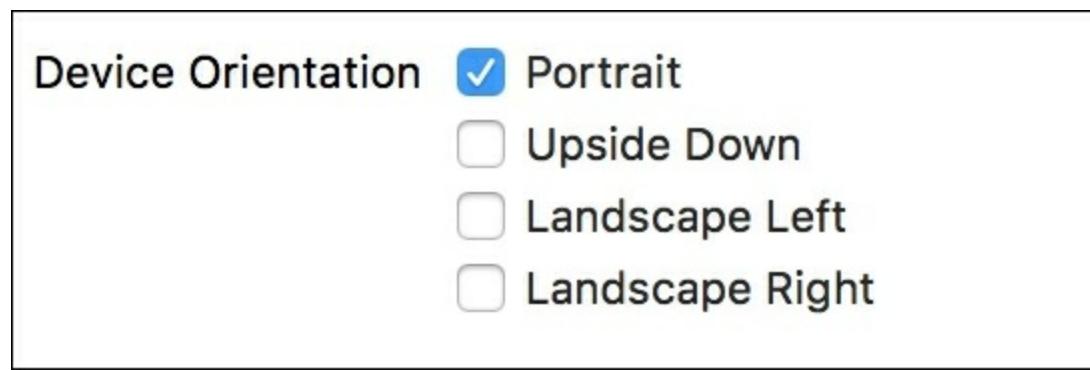
Disclaimer

This is not a real horoscope application; the algorithm used here has been invented. The idea behind this application is to create a sample of an application using ReactiveCocoa.

Setting up the project

Create a new **Single View Application** project called `Chapter 3 Horoscope`. Ensure that **Swift** is the main programming language and you use Git as your version control system. Install ReactiveCocoa using your favorite method, as shown in the previous chapter.

Click on your project in **Project Navigator**, select the **General** tab, and leave only the **Portrait** orientation checked, as shown in the following screenshot:



Before we start coding/adding UI components to the storyboard, we have to add some resources to our project. Here, we will want to add two pictures that will represent a checked and an unchecked box, which will be used in our form. Download the resources file from this book's website, and drag the `checked.png` and `unchecked.png` files into the `Assests.xcassets` component, which is located in **Project Navigator**.

Check whether the ReactiveCocoa installation is working by going to `ViewController.swift`, importing the framework with `import ReactiveCocoa`, and recompiling the project with *command + B*.

Now, we can start designing the layout of the application. Go to the storyboard, change the size class configuration to portrait mode for all iPhone devices by clicking on the configuration location in the center of the bottom of the storyboard where **wAny hAny** is mentioned, as demonstrated in this

screenshot:

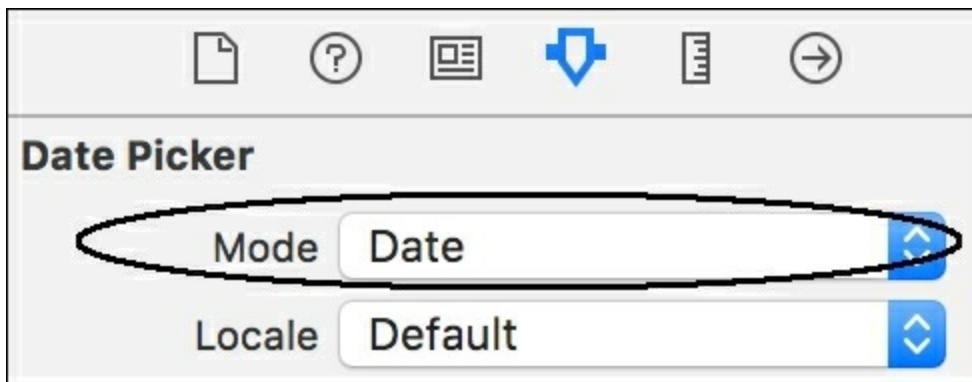


To start this application, we need to request some information on the user, otherwise, the user won't be able to check their horoscope. The information that's required is the username, e-mail, city of birth, gender, and the date of birth. To fill in this information, let's add three labels to our scene, three text

fields, three buttons, and a date picker.

Set the **Text** property of these labels to **Full name**, **Email**, and **City of birth**, respectively. Place each text field under each label. Put two buttons under the last text field, change their image to unchecked, and set **Title** to **Woman** and **Man**.

Under the buttons, place a `UIDatePicker` component. Go to its **Attribute Inspector** and change **Mode** to **Date** instead of the date and time, which appear by default:



In the bottom-right corner of the screen, place the last button and change its **Title** to **Check your horoscope**. The final result should be a view that's similar to what is shown in the following screenshot, but feel free to modify it if you are a creative person:

iPhone 5 - iPhone 5 / iOS 9.1 (13B134)

Carrier 7:10 PM

Full name

Email

City of birth

Woman Man

June	8	2012
July	9	2013
August	10	2014
September	11	2015
October	12	2016
November	13	2017
December	14	2018
January	15	2019

Check your horoscope

Once the layout is complete, we have to connect the UI components to their corresponding properties. You can use **Assistant Editor** if you like using the *command + Option + enter* shortcut. These are the properties that you have to connect the UI components to:

```
@IBOutlet weak var nameTextField: UITextField!
@IBOutlet weak var emailTextField: UITextField!
@IBOutlet weak var cityTextField: UITextField!
@IBOutlet weak var womanButton: UIButton!
@IBOutlet weak var manButton: UIButton!
@IBOutlet weak var datePicker: UIDatePicker!
@IBOutlet weak var checkButton: UIButton!
```

Creating a validator class

As mentioned earlier, the user can't continue to the next step if their username is not written or the e-mail ID is wrong. So, we need to create a class that can validate some fields. We will call this class `DataValidator` (very creative, indeed).

Add a new file to your project with *command + N*, select a Swift file when the dialog appears, and set its name to `DataValidator.swift`. Here, we have to start creating a class with two methods of validating: one for the username (and also for the name of the city) and the other one to check whether the e-mail ID is valid.

Starting with the name validator, we are going to consider that a name is valid if it contains more than two characters (I know, there is a city in France called Y; this is just a sample, so feel free to change it if you like), and it contains only words with letters or words that end with a period (such as J.R.R. Tolkien). To do this, we are going to use regular expressions. Type the following code in a new file in order to create the first validator:

```
class DataValidator {
    class func validName(name:String) -> Bool {
        if let regex =
            try? NSRegularExpression(pattern: "^\\w+(\\w+\\.?)*$",
                                     options: .CaseInsensitive) {
            return
                name.lengthOfBytesUsingEncoding(NSUTF8StringEncoding) > 2 &&
                    regex!.matchesInString(name, options:
                        NSMatchingOptions.ReportProgress, range: NSMakeRange(0,
                            name.lengthOfBytesUsingEncoding(NSUTF8StringEncoding))).count
                > 0
        }
        return false
    }
}
```

Once you've understood the idea behind this validator, the second one should be very straightforward as we just need to use another regular expression:

```
class func validEmail(email:String) -> Bool{
```

```
        if let regex =
            try? NSRegularExpression(pattern:
"^\S+@\S+\.\S+$", options: .CaseInsensitive) {
                return regex!.matchesInString(email, options:
NSMatchingOptions.ReportProgress, range: NSMakeRange(0,
email.lengthOfBytesUsingEncoding(NSUTF8StringEncoding))).count
> 0
            }
        return false
    }
} // end DataValidator
```

Tip

These regular expressions are just reduced expressions; use expressions that are more complete when you're working with a real application.

Validating text fields

Until now, everything is quite traditional with regard to developing with Xcode, but now we are going to start using the ReactiveCocoa framework. Let's start with only the username text field. If it is valid, its border must change to green. This way, the user will know that they have introduced a valid value.

Click on your `ViewController.swift` file and take a look at the `viewDidLoad` method. Here, we are going to check whether the ReactiveCocoa signals are working and try to understand the concept behind them first. Type the following highlighted code in `viewDidLoad` and run your code using *command + R*:

```
override func viewDidLoad() {
    super.viewDidLoad()
    nameTextField.rac_textSignal().subscribeNext { (input)
-> Void in
    print(input)
}

}
```

Once the application starts running, tap on the username text field and start typing your name. Have a look at the log console and ensure that each letter typed in the console prints the content of the text field. You might get a result similar to the following screenshot:

```
C
Ce
Cec
Ceci
Cecil
Cecil
Cecil C
Cecil Co
Cecil Cos
Cecil Cost
Cecil Costa
```

All Output ◊   

When you call the `rac_textSignal` function, it returns an object of the `RACSignal` type. A signal is any sequence of event that can be observed. In this case, we are using the text signal to observe when the text changes in the user text fields. After receiving this object, we add a subscriber to it that prints the current content on the screen by calling the `subscribeNext` method.

Make sure that everything is done on one line by chaining the function's calls; this is where ReactiveCocoa uses the fluent pattern. Another detail that you must keep in mind is that the input argument is of the `AnyObject` type, not `String`, and other methods also return `AnyObject`. Be careful about this as you will usually need to cast some arguments.

The next step is to check whether the name is valid or not. Right now, we can continue by just adding code to the `subscribeNext` method; however, this doesn't showcase the magic of ReactiveCocoa. We need a similar signal, but instead of receiving text as input, we need a Boolean value that represents the validity of the text field. To do this, we can convert one signal to another using the `map` function.

The usage of the `map` function is very simple: we just need to take the input, in this case, the text, and return a new value that represents something that we need for the signal. In our case, we have to return a Boolean value, which represents whether the text is valid. The result is a new signal that can have its own subscribers. Replace the code we had earlier with the following one:

```
let nameSignal:RACSignal =
nameTextField.rac_textSignal().map { (text) -> AnyObject! in
    return DataValidator.validName(text as! String)
}
```

Now, we have a new signal called `nameSignal`, which works the same way as the first code we had but sends the validity of the text to the log console. Again, let's perform a test by printing some output; continue our code by adding a subscriber using the following code:

```
nameSignal.subscribeNext { (valid) -> Void in
    if valid as! Bool {
        print("Valid")
    } else {
        print("Not valid")
    }
}
```

Click on play and type your name in the text field; take a look at the log console while you type your name. You will now receive a log similar to what is shown in the following screenshot:

```
Not valid
Not valid
Not valid
Valid
Valid
Valid
Not valid
Valid
Valid
Valid
Valid
Valid
```

All Output ◊

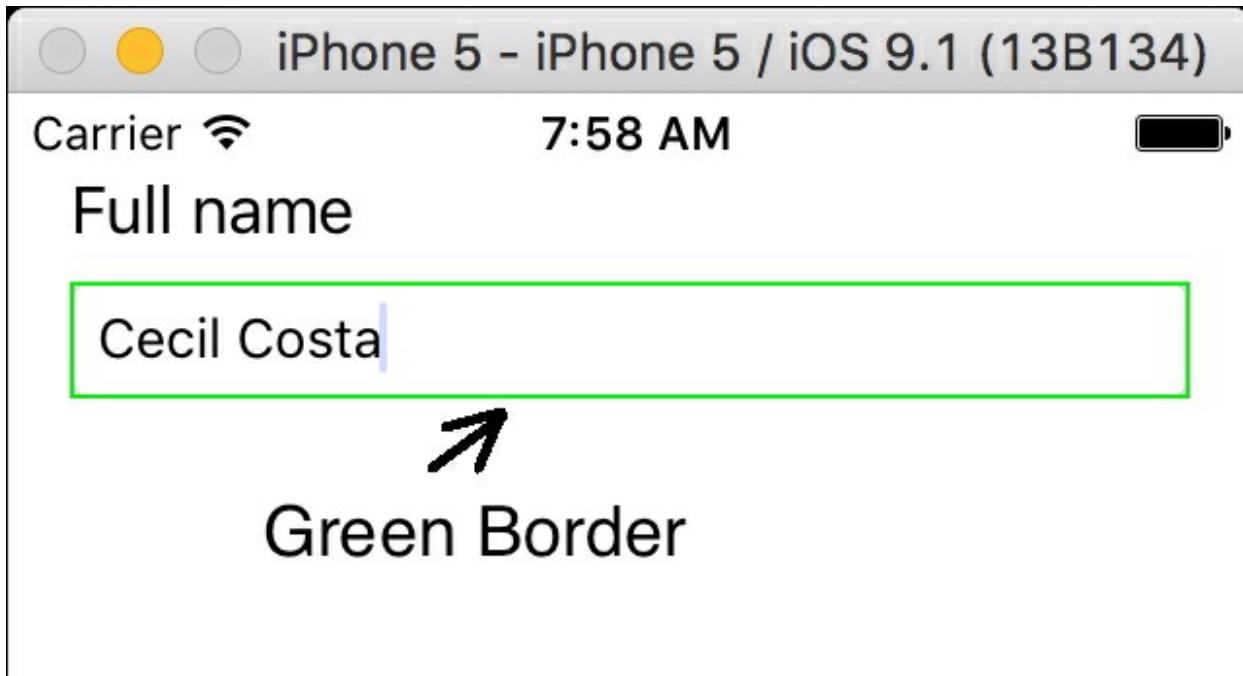


Now that we have explored the philosophy, we can do something more real. The user is not going to check the log console when using the application; thus, we have to change something on the phone screen. We will add a green border to the text field when the text is valid, and remove this border when the result is not. Again, we could do this inside the `subscribeNext` method, but that's not part of ReactiveCocoa's style. Map the signal to a border color, and then call `subscribeNext`. Replace the previous code with the following one:

```
nameSignal.map { (valid) -> AnyObject! in
    if valid as! Bool {
        return UIColor.greenColor()
    } else {
        return UIColor.clearColor()
    }
}.subscribeNext { (color) -> Void in
    self.nameTextField.layer.borderWidth = 1
    self.nameTextField.layer.borderColor = (color as!
UIColor).CGColor
}
```

Here, we've just taken `nameSignal`, created another signal that converts a

Boolean value into `UIColor`, and then the subscriber receives the border color and applies it. Now, test the application again, and make sure that when you type your name, the color green is applied to the text field or remove it when the text is not valid, for example, when you press the spacebar key but you haven't started yet with the next name/surname:



With this simple example, you can see some differences between using reactive programming and the traditional MVC pattern. Ensure that everything is set in the `viewDidLoad` method in a way that can be easily be read with just a few lines of code. It is also divided into different steps rather than a single function that contains everything.

Let's optimize our code a little bit: as the border width is always 1, we can move it outside the subscriber and place it at the beginning of the `viewDidLoad` method. By default, the color of the border is black, but if you start the application, you will see that it starts with a clear color. Why? The reason for this is very simple: when this signal starts, it triggers a sequence of events as an empty string is not considered a valid value in our

sample; it is mapped to a clear color when the application starts.

Great! We have other two text fields that follow the same rule, so let's follow the same steps we've followed for the name text field. To begin with, they will need a border width. This means that we just need to set their borders to 1 after the username border by adding the following highlighted code:

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

    nameTextField.layer.borderWidth = 1
emailTextField.layer.borderWidth = 1
cityTextField.layer.borderWidth = 1
```

Now, we can retrieve their signals in the same way as the username. As we already know that the value returned by the `map` function is `RACSignal`, this time we are not going to specify it (this is how you will usually find samples). Just place the highlighted code and compare it with the earlier one:

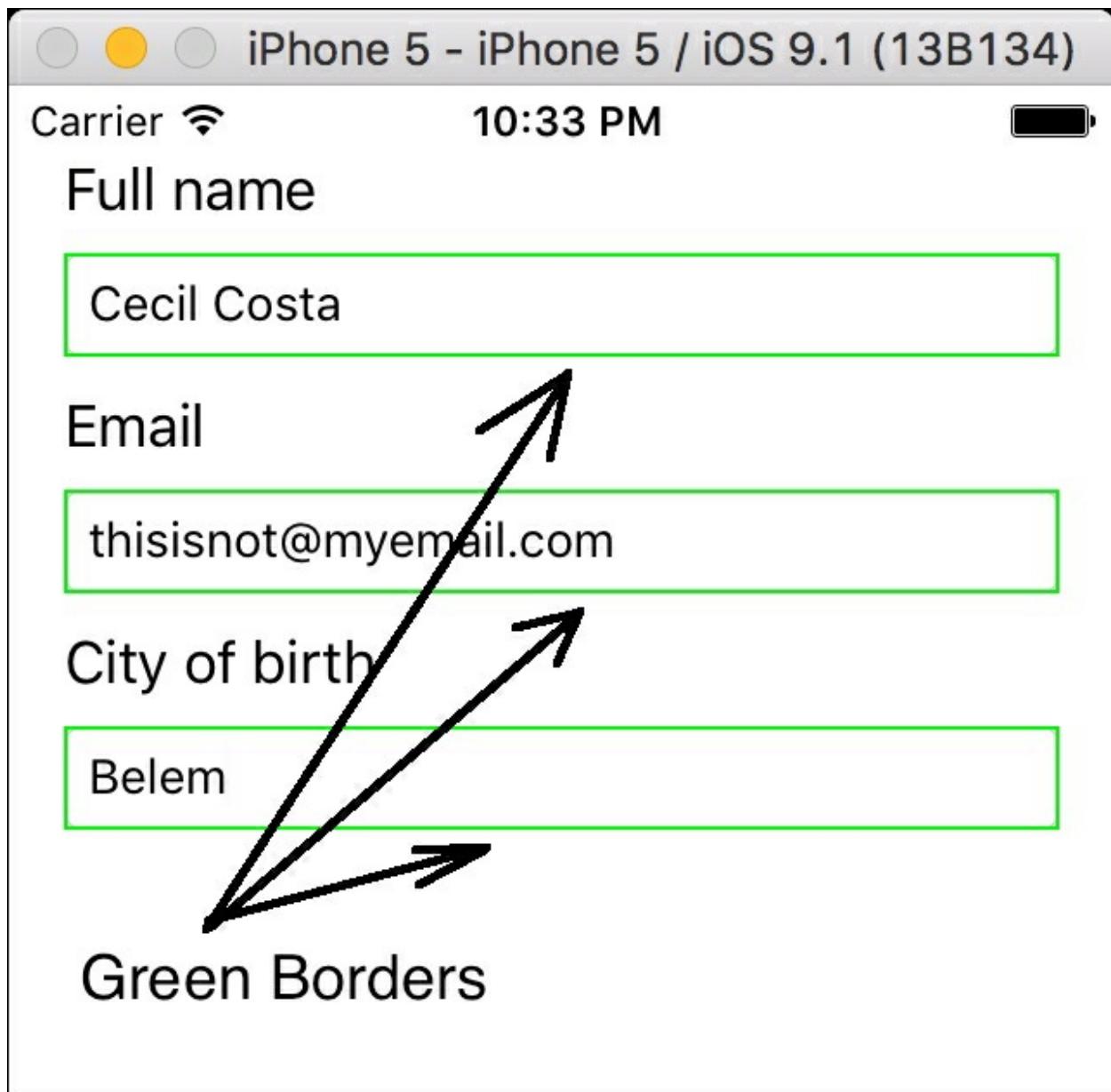
```
let nameSignal:RACSignal =
nameTextField.rac_textSignal().map { (text) -> AnyObject! in
    return DataValidator.validName(text as! String)
}
let emailSignal = emailTextField.rac_textSignal().map
{ (text) -> AnyObject! in
    return DataValidator.validEmail(text as! String)
}
let citySignal = cityTextField.rac_textSignal().map {
(text) -> AnyObject! in
    return DataValidator.validName(text as! String)
}
```

To finish off, we are going to add an equivalent code to add the green border. The code for these text fields is very similar to the username code; however, this time, we will use the `?:` operator; this way, we can have shorter and easier-to-read code:

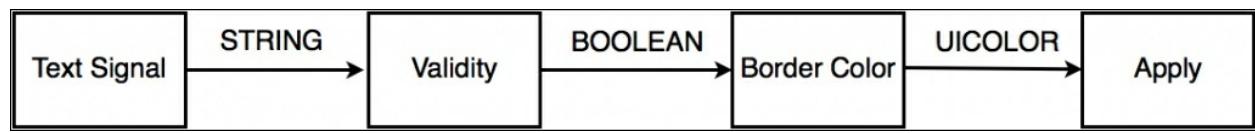
```
emailSignal.map { (valid) -> AnyObject! in
    return valid as! Bool ? UIColor.greenColor() :
UIColor.clearColor()
```

```
    }.subscribeNext { (color) -> Void in
        self.emailTextField.layer.borderColor = (color as!
UIColor).CGColor
    }
    citySignal.map { (valid) -> AnyObject! in
        return valid as! Bool ? UIColor.greenColor() :
UIColor.clearColor()
    }.subscribeNext { (color) -> Void in
        self.cityTextField.layer.borderColor = (color
as! UIColor).CGColor
    }
}
```

Let's test the application again. As you might have expected, each text field has its own green border when it has a valid value:



Basically, the flow of these text fields is very simple: every time the user types something, we map the text to a Boolean value that represents its validity. Then, we map this validity into a color, and finally, we apply the color to the text field border, as shown in the following diagram:



Enabling and disabling the button

The next part is enabling and disabling the button that allows the user to check their horoscope. The idea is to leave this button disabled if there anything that is missing or wrong in the form and enabling it only when we have a valid state to check the horoscope. But when should this button be enabled or disabled? At this point of the application, the button must be enabled when every text field is valid.

Right now, each text field works individually; they don't have any relationships. This is when reactive programming starts getting interesting. We need another signal that is basically the combination of the other three signals. In such cases, the `RACSignal` class has a class method called `combineLatest`.

Using `combineLatest` is very simple; we just need to send an array of signals as an argument (actually, it requests an `NSFastEnumeration` object). In this case, we have to send `nameSignal`, `emailSignal`, and `citySignal`. What next? In the first stage, we are going to add a subscriber that logs the word `Typing...`; this way, we can be sure that the signal is caught independently of the text field that is in use. Place the following code before closing the `viewDidLoad` method:

```
RACSignal.combineLatest([nameSignal, emailSignal,  
citySignal]).subscribeNext { (input) -> Void in  
    print("Typing...")  
}
```

Now, recompile your application and run it again. Start typing in any text field, and you will start receiving some feedback in the log console like this:

```
Typing...
Typing...
Typing...
Typing...
Typing...
Typing...
Typing...
Typing...
Typing...
```

All Output ⇲

Great! Now you have a signal that can be generated by any of the three initial signals; however, what are we receiving in the `input` argument? As you know, this argument is defined as `AnyObject`, but if you inspect it properly, you will see that it is an object of the `RACTuple` type.

`RACTuple`, as the name suggests, is a representation of a tuple of elements. In this case, it is a tuple where the first element is a Boolean value that comes from `nameSignal`, the second element is a Boolean value that comes from `emailSignal`, and the third element is a Boolean value that comes

from `citySignal`. To access these values, this object has properties called `first`, `second`, `third`, and so on.

Once we have understood this concept, we will retrieve these values and enable or disable `checkButton`. To do this, replace the previous code with the following one:

```
RACSignal.combineLatest([nameSignal, emailSignal,  
citySignal]).subscribeNext { (input) -> Void in  
    let tuple = input as! RACTuple  
    let validName = tuple.first as! Bool  
    let validEmail = tuple.second as! Bool  
    let validCity = tuple.third as! Bool  
    self.checkButton.enabled = validName && validEmail  
&& validCity  
}
```

Recompile your application and run it again. Ensure that the button at the bottom is enabled when the three text fields are green and disabled otherwise:



This code works perfectly well; however, the subscriber code style is imperative. In such cases, you usually have to map the signal, giving only the required information to the subscriber. Thus, update your code like this:

```

RACSignal.combineLatest([nameSignal, emailSignal,
citySignal])
    .map { (input) -> AnyObject! in
        let tuple = input as! RACTuple
        let validName = tuple.first as! Bool
        let validEmail = tuple.second as! Bool
        let validCity = tuple.third as! Bool
        return validName && validEmail && validCity
    }.subscribeNext { (valid) -> Void in
        self.checkButton.enabled = valid as! Bool
    }
}

```

Now, our code is more functional. Can we improve it? In cases where every tuple member is of the same type, you can use a method called `allObjects`. This method returns an array of `AnyObject`, but we can easily cast it as an array of `Bool`. Once we have this array of Boolean values, how we should evaluate them? The easiest way to do this is with a `for` loop, but in functional programming, it is preferable to use the `reduce` function that is built into the array type.

Note

If you want to know the power of the `map` and `reduce` functions, there are some NoSQL databases whose querying systems are based on these functions.

The `reduce` function is performed in order to converting the elements of an array into new values. It receives an initial value and one function that combines the previous value with the current one. Now, our code can be even simpler when you replace it with the following:

```

RACSignal.combineLatest([nameSignal, emailSignal,
citySignal])
    .map { (input) -> AnyObject! in
        let tuple = input as! RACTuple
        let validityValues = tuple.allObjects() as! [Bool]
        return validityValues.reduce(true, combine: {
(previousValue, currentValue) -> Bool in
            return previousValue && currentValue
        })
    }.subscribeNext { (valid) -> Void in
        self.checkButton.enabled = valid as! Bool
}

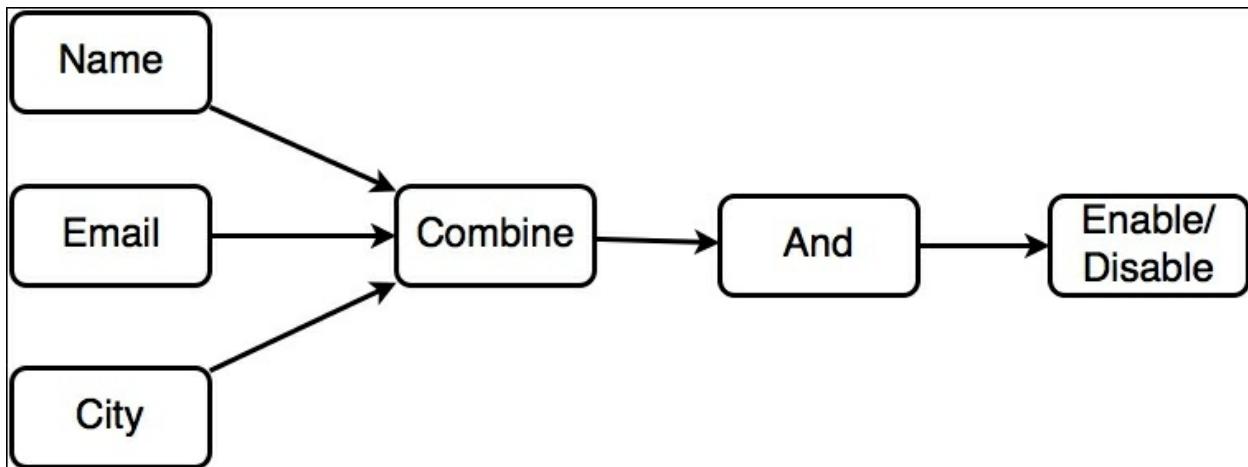
```

}

This type of mapping is very common as it is normal to use `combineLatest` with signals that have mapped the values to a Boolean array and then send the `and` operation between every element to the subscriber. In this case, we validate a form that can be used, for example, for a multiplayer game. This is used in instances when we try to figure out whether every player has reached the goal or when the application checks whether the phone status is fine for continuous work (for example, whether there's an Internet connection, the application is in the foreground, and so on). In such cases, we can reduce our code even more using a method called `and` rather than `map`. The final result is a code that's as simple as this:

```
RACSignal.combineLatest([nameSignal, emailSignal,  
citySignal])  
    .and().subscribeNext { (valid) -> Void in  
        self.checkButton.enabled = valid as! Bool  
    }
```

Rebuild your application, and check whether everything is working as it should. A button must be enabled when every text field is green and disabled when one of them isn't. To summarize this process, the following diagram gives you an idea of the current flow:



Using UIDatePicker

Until now, we have used text fields, receiving their signals through the `rac_textSignal` method, but text fields are not the only UI components that the user can interact with. In this case, we have `UIDatePicker`, which allows the user to choose their date of birth.

It's obvious that `UIDatePicker` doesn't have a text signal; therefore, we need to use a different one. Here, we have to use an equivalent for the `ValueChanged` event. ReactiveCocoa provides us with a function for every object that inherits from `UIControl`, called `rac_signalForControlEvents`.

We need to map this signal to a Boolean value by checking whether the user has entered a valid date every time they change a row. The `map` function, when used after `rac_signalForControlEvents`, receives the UI component itself as an argument. For the date of birth, we will assume that any date until today is valid, and only future dates are invalid. Let's take a look by adding the following code just before the `combineLatest` call:

```
let dateSignal =
datePicker.rac_signalForControlEvents(UIControlEvents.ValueChanged).map { (input) -> AnyObject! in
    let datePicker = input as! UIDatePicker
    return
    datePicker.date.timeIntervalSinceDate(NSDate(timeIntervalSinceNow: 0)) < 0
}
```

Once we've got this signal, it is necessary to include it in the list of signals in the `combineLatest` call. Have a look at the highlighted code:

```
RACSignal.combineLatest([nameSignal, emailSignal,
citySignal, dateSignal])
    .and().subscribeNext { (valid) -> Void in
        self.checkButton.enabled = valid as! Bool
    }
```

In this case, the order of the signals doesn't matter as they are processed with the `and` function; however, you use `RACTuple` with the `first`, `second`...

properties, so the order does matter.

Let's test this part and check its current behavior. Build and run your application, and when the scene appears, check whether you can find anything weird. Right, the button is enabled even if you type an invalid e-mail; for example, the button that shows the horoscope is still enabled. After changing the date for the first time, the button starts responding according to our rules. Why?

The reason is very obvious once you know it. We map the date after changing its value; when the application starts, this event hasn't even been performed yet; therefore, `combineLatest` can't process the latest date signal as it hasn't happened yet. So, the `and` function is not even called.

How can we solve this problem? We just need to start the signal with an initial value; in this case, we can just send the date picker as the initial value. To set this initial value, a function called `startWith` receives the initial value. Complete the code by adding the highlighted call:

```
let dateSignal =  
datePicker.rac_signalForControlEvents(UIControlEvents.ValueChanged).map { (input) -> AnyObject! in  
    let datePicker = input as! UIDatePicker  
    return  
datePicker.date.timeIntervalSinceDate(NSDate(timeIntervalSinceNow: 0)) < 0  
} .startWith(datePicker)
```

Build and run the application. Now, you will see that we have the right behavior. The button starts when it's disabled, and when every text field and the date picker has valid values, the button gets enabled.

Should we add a green border to the date picker? This is the typical question that you have to ask a **User Experience (UX)** guy; he is the one who needs to decide the look and feel of our application. Imagine that he says that we need to apply the green border to the date picker. How can we do this? As you know, we just need to take the signal, map it to a color, and add a subscriber. Place the following code just before the `combineLatest` call:

```

dateSignal.map { (valid) -> AnyObject! in
    return valid as! Bool ? UIColor.greenColor() :
    UIColor.clearColor()
        }.subscribeNext { (color) -> Void in
            self.datePicker.layer.borderWidth = 1
            self.datePicker.layer.borderColor = (color as!
    UIColor).CGColor
}

```

Rebuild the application, and then run and appreciate a good crash. Why? When we assigned the value of `dateSignal`, it received the value from the last call in the chain, in this case from the `startWith` call, which returns a signal based on its input, a date picker. How can we fix this problem?

Here, we have to switch calls: `startWith` must come first, and after this, we can map it to a Boolean value. Thus, fix the code by replacing the current `dateSignal` assignation in this way:

```

let dateSignal =
datePicker.rac_signalForControlEvents(UIControlEvents.ValueChanged).startWith(datePicker).map { (input) -> AnyObject! in
    let datePicker = input as! UIDatePicker
    return
datePicker.date.timeIntervalSinceDate(NSDate(timeIntervalSinceNow: 0)) < 0
}

```

Now, you can build the application and test it again. It works perfectly, and moments like these are when you realize that you are very proud of your code. You then show it to your boss and he asks you, "Why does the date picker start with a green border?"

Green border when the app starts



July	13	2012
August	14	2013
September	15	2014
October	16	2015
November	17	2016
December	18	2017
January	19	2018
February	20	2019

Check your horoscope

The answer for this is logical. Since the date picker starts with the current

date, which is a valid date, it therefore starts with a green border. This explanation is okay, but imagine when we receive the typical answer: no green border when the application starts, and do not change the initial date. This case basically means that the first time, the subscriber mustn't act. For cases like this, we have a method called `skip`.

The `skip` method takes an integer as an argument; it represents the number of times that the subscriber should ignore the signal. Once we have understood this, we only need to insert the highlighted code:

```
dateSignal.map { (valid) -> AnyObject! in
    return valid as! Bool ? UIColor.greenColor() :
    UIColor.clearColor()
}.skip(1).subscribeNext { (color) -> Void in
    self.datePicker.layer.borderWidth = 1
    self.datePicker.layer.borderColor = (color as!
UIColor).CGColor
}
```

Great! Now, test the application, and check whether it is working as it did earlier along with the initial green border.

Selecting the gender of the user

So far, the application has control for the text fields and the date picker, but it still needs to ask the user their gender. To fix this, we have two buttons with an image of an empty circle on them. These buttons work as radio buttons; when one of them is selected, the other one should be deselected. Why don't we use a switch instead? A switch has two states, on and off, and here, we would like a third state that is not selected, which means that there is no gender selected by default.

Once the user has tapped on the gender button, it doesn't matter which one, the corresponding signal must be valid. How can we do this? It's very easy: first, we have to take the `TouchUpInside` signal of each button, and map it to the `true` value of a Boolean. Let's visualize it by placing the following code before the `combineLatest` call:

```
let womanSignal =  
womanButton.rac_signalForControlEvents(.TouchUpInside).map {  
(signal) -> AnyObject in  
    return true  
}
```

When is this signal going to be false? Right now, the answer is never. If we add this signal to the collection of validations, we will have the same problem, wherein it will block the evaluation. Again, we can solve this problem using the `startWith` method. Complete the previous code we had with the following code; here, we have the code for both buttons:

```
let womanSignal =  
womanButton.rac_signalForControlEvents(.TouchUpInside).map {  
(signal) -> AnyObject in  
    return true  
}.startWith(false)  
  
let manSignal =  
manButton.rac_signalForControlEvents(.TouchUpInside).map {  
(signal) -> AnyObject in  
    return true  
}.startWith(false)
```

Now, we have to add these signals to the array that is sent to the `combineLatest` function; however, if we add both buttons, the behavior of the application will be wrong as we are using the `and` operator, which would require both buttons to be true in order to return a valid state. This problem can be easily fixed by combining both signals with the `or` operator in the same way we learned earlier in this chapter. Place the following code to create a combination of both of these signals:

```
let genderSignal = RACSignal.combineLatest([womanSignal,  
manSignal]).or()
```

Great, now add this signal to the main `combineLatest` call. Rerun your code to check whether the button that allows us to check the horoscope is enabled only when the user has filled in every text field correctly, chosen a valid date of birth, and selected their gender:

```
RACSignal.combineLatest([nameSignal, emailSignal,  
citySignal, dateSignal, genderSignal])  
.and().subscribeNext { (valid) -> Void in
```

After testing, you will notice that the application is working perfectly fine; however, there are a couple of details that need to be solved. First, we are not changing the icon of the tapped button. Second, whenever you need to, you can take the text fields' values and the date on the date picker, but the buttons will not store this information.

For the first detail, we could add the corresponding code inside the subscriber of each button, but this wouldn't be too maintainable. Imagine a similar case where you have an application with five or six buttons on the screen; would we have to repeat the same operation for every button?

For the second detail, we can create an optional property of the Boolean type. Again, the solution is valid, but it will not work in reactive programming. Changing a Boolean value doesn't make propagation easier, which means that changing its value doesn't change the button's status. Changing the propagation will change the variable value and notify other observers about such a change.

The solution for both issues is the same: we have to create a property of the

`MutableProperty` type. This generic class works like a traditional value but with the advantage of propagating its changes through a subscriber.

Let's declare a new property before starting the `viewDidLoad` method with the following code:

```
let gender = MutableProperty<Bool?>(nil)
```

As you can see, `gender` can have three values: `nil`, `true`, and `false`. It starts with `nil`, and represents that no gender is selected; after this, if the user clicks on the **Woman** button, it shall take the `false` value, and if the user presses the **Man** button, it shall take the `true` value. Once we know it, we have to update the signal of both buttons by setting the `value` property of the `gender` property. Update your code by adding the following highlighted code to it:

```
let womanSignal =
womanButton.rac_signalForControlEvents(.TouchUpInside).map {
(signal) -> AnyObject in
    self.gender.value = false

    return true
}.startWith(false)

let manSignal =
manButton.rac_signalForControlEvents(.TouchUpInside).map {
(signal) -> AnyObject in
    self.gender.value = true

    return true
}.startWith(false)
```

A `MutableProperty` has two properties: `value`, which was just explained and contains the current value, and `producer`, which is a signal producer and something like a signal but with some added details. We are going to learn more about the signal producer in the next chapter.

How can we set a subscriber to a signal producer? We just need to call the `start` method with the `next` argument. This argument is a closure that receives the new value and returns nothing. As this property is of the `Bool?` type, the new value will be of the same type. Here, we can use a switch to check out the

current scenario and act accordingly by setting the right image for each button. Add the following code after the declaration of the `genderSignal` constant:

```
self.gender.producer.startWithNext { newValue in
    switch newValue {
        case nil:
            self.womanButton.setImage(UIImage(named:
"unchecked"), forState: .Normal)
            self.manButton.setImage(UIImage(named:
"unchecked"), forState: .Normal)
        case .Some(true): // Man
            self.womanButton.setImage(UIImage(named:
"unchecked"), forState: .Normal)
            self.manButton.setImage(UIImage(named:
"checked"), forState: .Normal)
        case .Some(false): // Woman
            self.womanButton.setImage(UIImage(named:
"checked"), forState: .Normal)
            self.manButton.setImage(UIImage(named:
"unchecked"), forState: .Normal)
    }
}
```

Once again, test the application, and check whether it is working as expected. What do we have to do now? We just need to display something when the horoscope button is tapped. How can we do this? Just add a subscriber. Add the following code before closing the `viewDidLoad` method:

```
checkButton.rac_signalForControlEvents(.TouchUpInside).subscribeNext { (button) -> Void in
    let alertController = UIAlertController(title:
"Horoscope", message: "You will have a wonderful day!", preferredStyle: .Alert)
    self.presentViewController(alertController, animated: true, completion: nil)
}
```

Run this final version of our application, and check that when the horoscope button is enabled, you can tap it so that you receive this message:

iPhone 5 - iPhone 5 / iOS 9.1 (13B134)

Carrier 12:48 AM

Full name

Cecil E. C. Costa

Email

not@this.com

City of birth

Belem

Woman Man

Horoscope

You will have a wonderful day!

August	10	1977
September	11	1978
October	12	1979
November	13	1980
December	14	1981

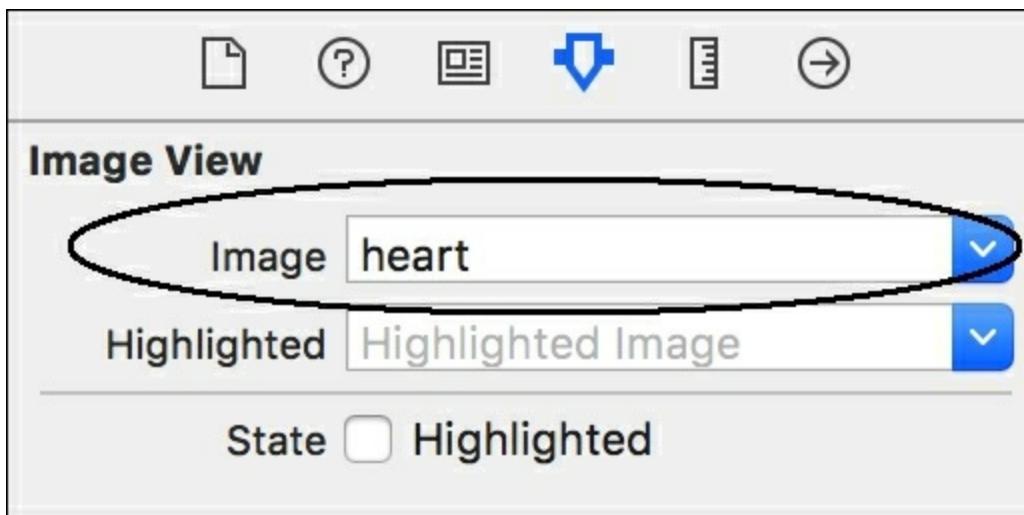
Check your horoscope

Adding more information

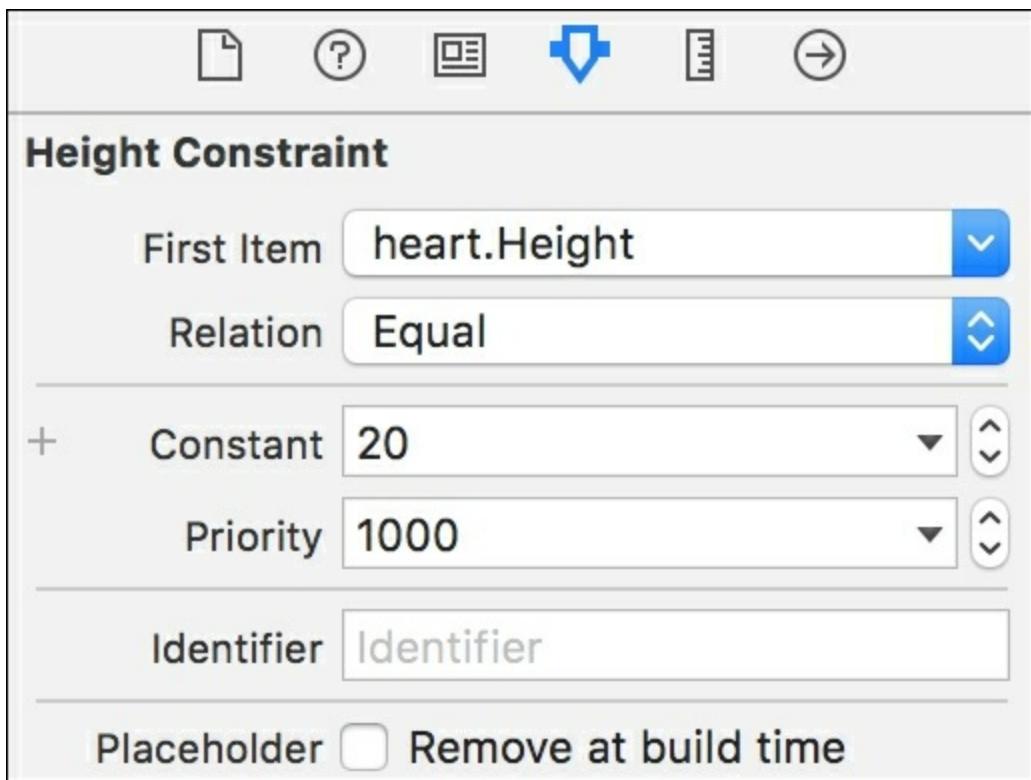
Let's imagine that the current information is not enough to calculate the horoscope, and we need to request a little more data from the user, for example, their current emotional status.

In order to do this, we will need to add a new image to our project, which is the `heart.png` file and is located in this book's resources. Click on your `Assets.xcassets` project, and add your file to this folder.

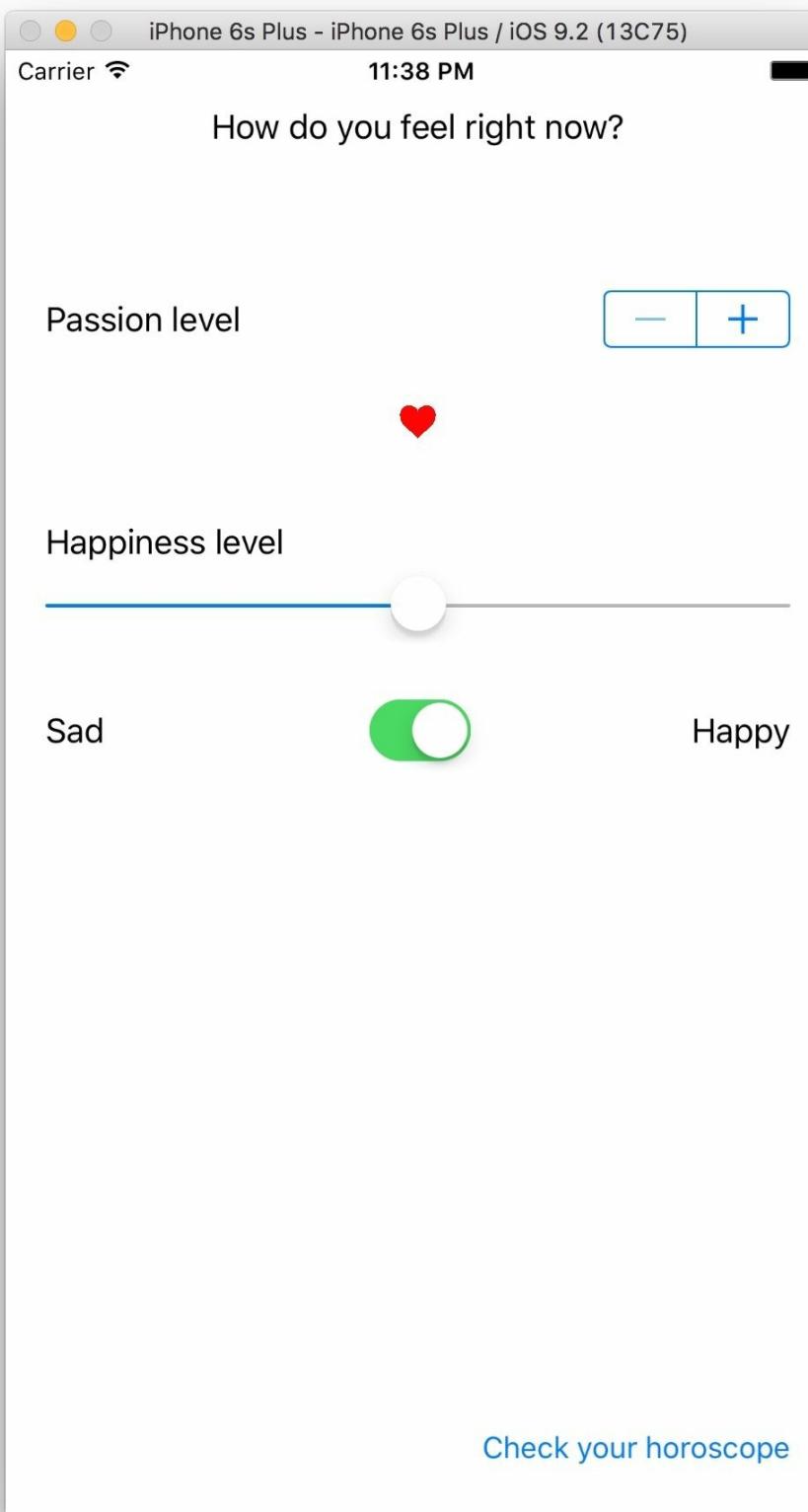
Return to the storyboard, and add a new scene to it. Here, we have to add one label to the top where you see **How do you feel right now?** Then, you will see a section with a stepper and an **Image View**, which indicate the passion level. Select **Image View**, go to its **Attribute Inspector**, and set its image to heart, as shown here:



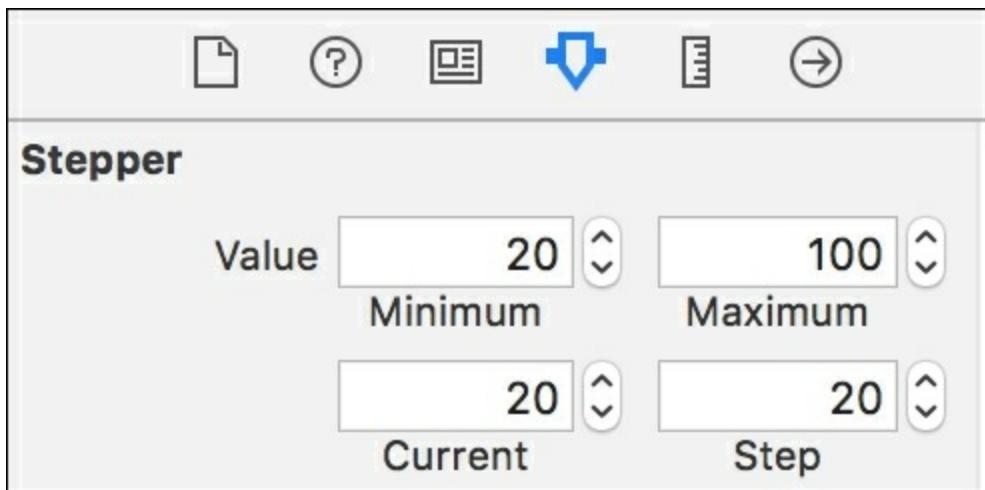
After setting its image, add a **Height Constraint** and set its value to 10. You can refer to this sample:



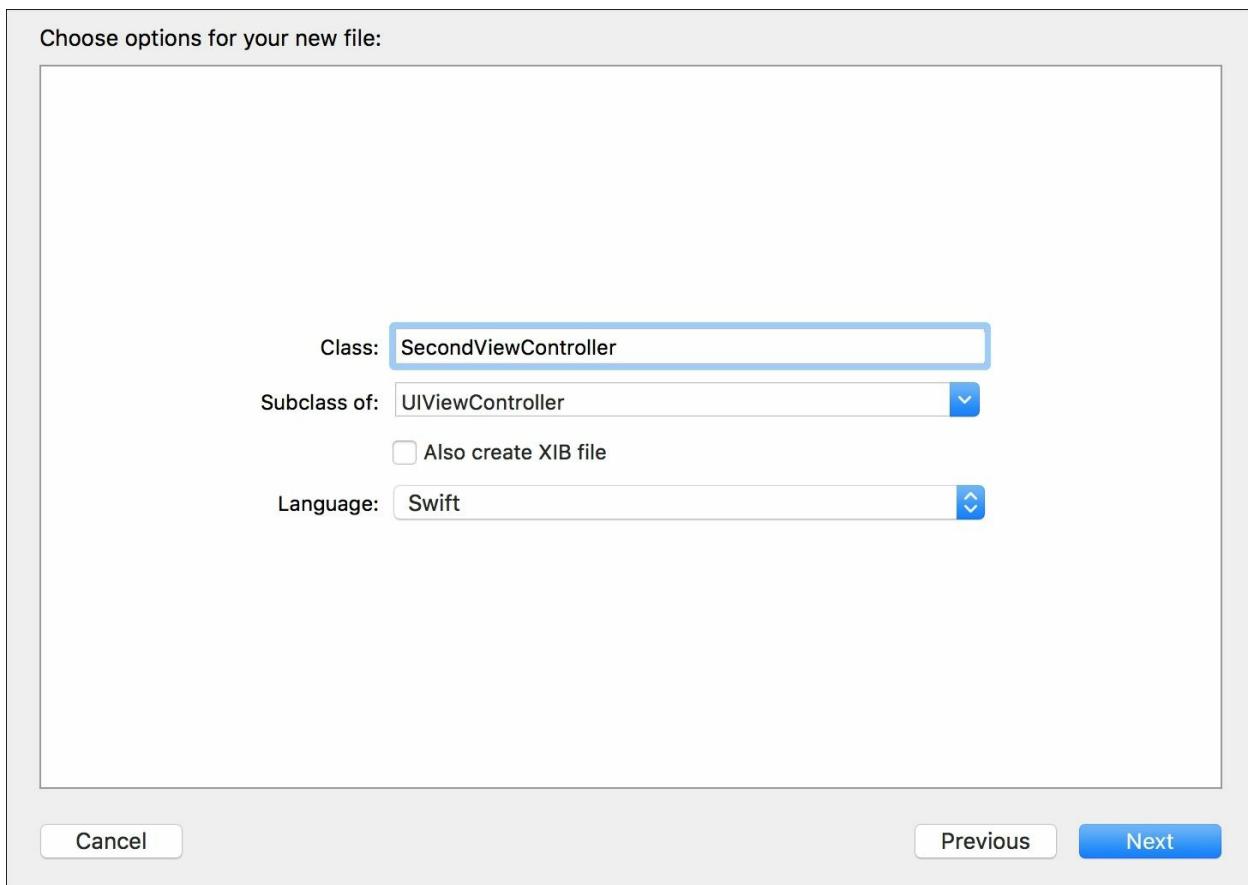
Under the passion level, we will add `UISlider` and `UISwitcher`, which will represent your happiness level. Add a few labels that you think that are necessary for the user. In the bottom-left corner of the scene, add another button saying, **Check your horoscope**, as we did on the first scene. The final layout for this scene should look similar to what is shown here:



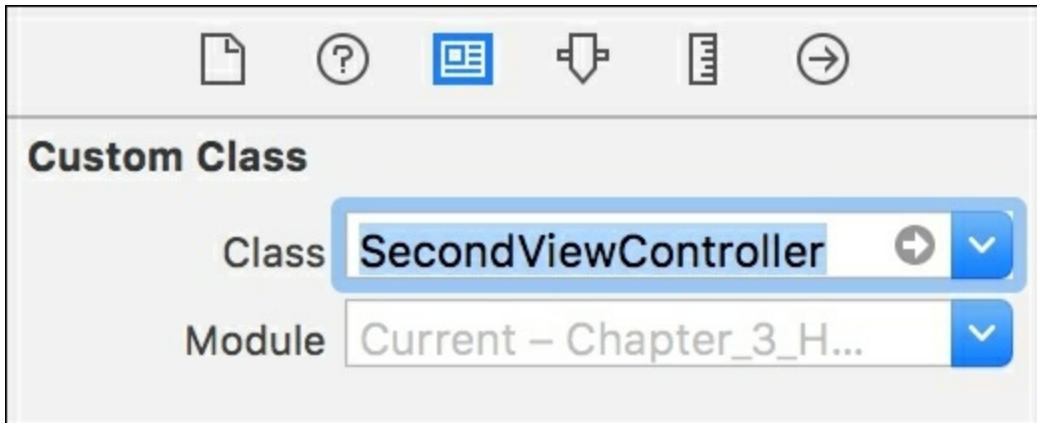
Before continue, we will configure the stepper. The stepper will increase the heart size; therefore, we can use values that are more likely to be present in this feature. Suppose the minimum value is 20, the maximum value is 100, and every step will be increased by 20. This means that we will have the values 20, 40, 60, 80, and 100. Go to the **Attribute Inspector** of our stepper and set its configuration according to this:



Once the scene is done, we need to create a class that inherits from `UIViewController`. Add a new file to your project with the combination *command + N*, and select the **Cocoa Touch Class** option in the first dialog. In the second dialog, set the class name to `SecondViewController`, and ensure that it is a subclass of `UIViewController`. Here, you can see the correct configuration:



Right now, this class has no relationship with the last scene we created, so we have to return to the storyboard, select the **Identity Inspector** of the second scene by selecting the scene, and using the *command + option + 3* combination, set its class to `SecondViewController`, as shown here:



Now we can open the **Assistant Editor** and connect the stepper, slider, switch, button, and the image height constraint (yes, the constraint that represents the height, not the image view) with their corresponding attributes:

```
@IBOutlet weak var passionStepper: UIStepper!
@IBOutlet weak var happinessSlider: UISlider!
@IBOutlet weak var happySwitch: UISwitch!
@IBOutlet weak var checkButton: UIButton!
@IBOutlet weak var imageHeight: NSLayoutConstraint!
```

Then, we can start developing the signal subscriptions. Firstly, we will ask the user for their passion level, which will be displayed according to the heart size that is displayed on the screen; therefore, we can start developing the `viewDidLoad` method with a signal that detects a value that's changed from the stepper with the following code:

```
passionStepper.rac_signalForControlEvents(.ValueChanged)
    .subscribeNext { (input: AnyObject!) in
        let stepper = input as! UIStepper
        self.imageHeight.constant = CGFloat(stepper.value)
    }
```

Run your app, and check whether the heart size increases when you tap the plus sign or decreases when you tap the minus sign.

Getting the right input type

Until now, you could appreciate that every time we used methods such as `subscribeNext`, `map`, or `filter`, we received an argument of the `AnyObject` type, and the first task we had to undertake was to convert this argument into its corresponding value.

The reason for this is that `RACSignal` was written in Objective-C, a language that doesn't have generics as a feature; thus, the only way of creating functions in which the input type can vary is using a type called `ID`, which is similar to `AnyObject` in Swift.

If Swift has the generics feature, why not using it? Of course, this is something that we can do. Colin Eberhardt has a proposal that includes extending the `RACSignal` class with some new methods such as `subscribeNextAs`, `filterAs`, and `mapAs`. This extension must be written in Swift as it takes advantage of generics.

How can we add this extension to our project? It is not complicated: just add a new Swift File to your project, call it `RACSignalExtension.swift`, and paste the following code made by Mr. Eberhardt:

```
import ReactiveCocoa

extension RACSignal {
    func subscribeNextAs(nextClosure: (T) -> ()) -> () {
        self.subscribeNext {
            (next: AnyObject!) -> () in
            let nextAsT = next as! T
            nextClosure(nextAsT)
        }
    }

    func filterAs(nextClosure: (T!) -> Bool) -> (RACSignal) {
        return self.filter {
            (next: AnyObject!) -> Bool in
            if(next == nil) {
                return nextClosure(nil)
            } else{

```

```

        let nextAsT = next as! T
        return nextClosure(nextAsT)
    }
}
}

func mapAs(nextClosure: (T!) -> AnyObject!) -> (RACSignal)
{
    return self.map {
        (next: AnyObject!) -> AnyObject! in
        if(next == nil) {
            return nextClosure(nil)
        }else{
            let nextAsT = next as! T
            return nextClosure(nextAsT)
        }
    }
}
}

```

Have a look at this code: what it does is simply wrap the corresponding method (`mapAs` wraps up a `map`, for example) but through the use of closures that receive a specific type; this way, we can have cleaner code. Once we have understood this idea, we can replace the previous signal call with this one:

```

passionStepper.rac_signalForControlEvents(.ValueChanged)
    .subscribeNextAs { (stepper: UIStepper) in
        self.imageHeight.constant = CGFloat(stepper.value)
    }

```

Great, run your project again and check whether it works like it did earlier; however, now we don't need to cast the input as it already comes with its own type. In this book, we will eventually learn about the new features of ReactiveCocoa 3 and 4; this will also remove the usage of `AnyObject` as an argument.

What about the first View Controller? Shall we change all the calls to this new format? It is up to you! If you like, do it as an exercise or homework.

Using bidirectional channels

In this part of the app, the user can tell whether they feel happy or not. There are two ways of doing this: the first one is through a slider, which allows the user to convey their happiness level with some kind of accuracy. Moving the slider to the far left means that the user is really sad, and moving it to the far right means that the user is extremely happy.

However, not everybody would want to express their happiness with that kind of accuracy. Some people can only say that they are happy or sad; for this reason, we have an alternative switch where the user can just express whether they are happy or not.

The main detail we have to consider is that when the slider is on the half-left, the switch must be off, and if the slider is on the half-right, the switch must automatically turn on. What if the user prefers pressing the switch button? In this case the slider needs to move to a position where it indicates whether the user is happy or sad. As the switch doesn't have very much accuracy, we can set the slider to 0.25 when the user is sad or 0.75 when the user is happy.

After going through this theory, have you noticed that something redundant? Notice that one UI component is changing more than the other. Doesn't this sound like recursion? For cases like this, Reactive Cocoa has a feature called RACChannel.

RACChannel allows us to use bidirectional data binding; this means that we can use a channel for communication, switching from one component to another. These channels allow the UI components to send or receive a value. To use it, let's start by asking the slider to create a channel with a method called `rac(newValueChannelWithNilValue)`, and start by continuing with `viewDidLoad`:

```
let happinessSliderChannel =
happinessSlider.rac(newValueChannelWithNilValue(0))
```

On the other hand, the switch needs to create another channel. `UISwitch` can

can create a channel through a method called `rac_newOnChannel`. Continue implementing the `viewDidLoad` with the following code:

```
let happySwitchChannel =  
happySwitch.rac_newOnChannel()
```

When the switch changes, we have to map the switch value. In this case, when the value is `false`, we will map it to `0.25`, and when it is `true`, we will map it to `0.75`. We can use `map` as we use to do with `RACSignal`, and thus continue with the following code:

```
let happyChannelTerminal = happySwitchChannel.map {  
    (value: AnyObject!) -> AnyObject! in  
    if let active = value.boolValue where active {  
        return 0.75  
    }  
    return 0.25  
}
```

Now that we have the channels, we can tell this mapped channel (called `happyChannelTerminal`) that the slider channel (called `happinessSliderChannel`) will be its subscriber using the `subscribe` method. What does it mean? It means that when the switcher changes its values, it will be mapped and sent directly as a new value of the slide. Place the following code to make this possible:

```
happyChannelTerminal.subscribe(happinessSliderChannel)
```

At this point, we can rebuild our app and test it again. Right now, you can move the slider freely; however, when you change the switcher from sad to happy, you will see that it's set to `0.75`, and when the switcher is set to sad (`false, off`), the slider is automatically set to `0.25`.

Now, we need it to work the other way around: whenever the slider moves to a position where its value is greater than `0.5`, the switcher must be turned on (happy); otherwise, it must be `off` (sad).

The logic is the same as the previous one. We have to start mapping the slider threshold into a Boolean value. Do this using the following code:

```
let happinessChannelTerminal =  
happinessSliderChannel.map {  
    (value:AnyObject!) -> AnyObject! in  
    return value != nil && value.doubleValue >= 0.5  
}
```

Finally, we can make the switch channel a subscriber of the slider channel using the `subscribe` function:

```
happinessChannelTerminal.subscribe(happySwitchChannel)
```

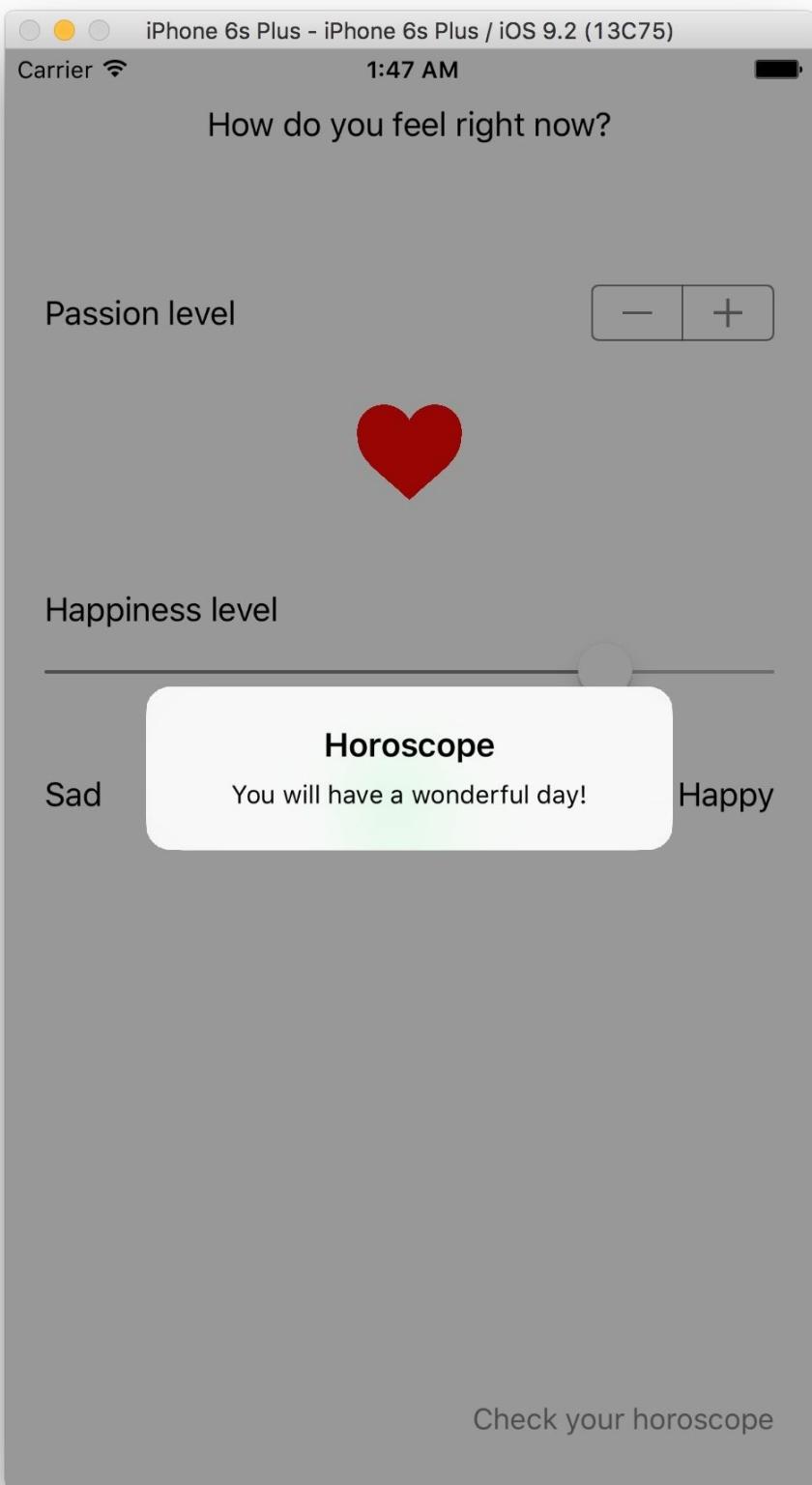
The code is done, so now it is time to test our app again. Fill the first and second form; once you are happy with it, you can move to the last step, which basically displays the horoscope result.

Displaying your horoscope

After testing the application and checking whether it is working as expected, what do we do now? We just need to display something when the horoscope button is tapped. How can we do this? Just add a subscriber. Add the following code before closing the `viewDidLoad` method:

```
checkButton.rac_signalForControlEvents(.TouchUpInside).subscribeNext { (button) -> Void in
    let alertController = UIAlertController(title: "Horoscope", message: "You will have a wonderful day!", preferredStyle: .Alert)
        self.presentViewController(alertController, animated: true, completion: nil)
}
```

Run this final version of our application, and verify that when the horoscope button is enabled, you can tap it so that you receive this message:



Summary

In this chapter, you learned the basics of reactive programming. You can now appreciate how it works with a fluent pattern by chaining calls.

You took a look at how everything is based on signals. These signals could be mapped by converting a current value into another value, after which the final result is a new signal.

You also learned how to combine these signals and avoid some pitfalls.

Finally, you took a look at the propagation of changes, which, in this case, was done with the `gender` property. Every time it was updated, it updated a button's images as well.

If you were a good observer, you would have noticed that we had almost no variables (only those that were required by **Interface Builder (IB)**); everything else was in the form of constants. Constants can be optimized to a greater extent by a compiler. Furthermore, we had almost no `if/switch` statements or loop controls, and everything could be done inside the `viewDidLoad` method, avoiding the creation of new methods in order to remember where they are connected.

In the second scene, we learned how to extend `RACSignal`, with the advantage of using Swift's features, such as generics, and avoiding the casting of every input. At the end of this chapter, we also learned how to use `RACChannel`, which is a way of mapping the value of a component directly as the value of some other component. In our example, we mapped the slider values to a switch value and vice versa.

You can develop the same application without using reactive programming and compare both sets of code. Simply check which one has fewer lines of code and is easier to maintain.

In the next chapter, you are going to learn how to work with asynchronous calls, mainly by performing network requests. We are also going to use table

views with ReactiveCocoa. You will see that performing some tasks, such as updating table view cell pictures, can be easy and safe using this framework.

Chapter 4. Network and Change Propagation

In the previous chapter, you learned about the basics of ReactiveCocoa and how to use it with form validation. As mentioned in [Chapter 1](#), *Introduction to Reactive Programming*, reactive programming is the perfect fit when you use it for asynchronous calls.

What's a better example of asynchronous calls than a network? That's right, nowadays, it is very common to exchange messages with a server in order to receive information from it. Now, you are going to learn how to make some HTTP requests with the help of JSON messages (also called AJAX in a web environment) and ReactiveCocoa. In this application, a user will be able to search for their favorite movie and receive some information about it.

In this chapter, we will cover these topics:

- Using ReactiveCocoa with network streaming
- Using ReactiveCocoa as a fail-safe alternative
- Creating signals
- Observing a property
- Lifting selectors
- More RACSignal options

Overviewing the project

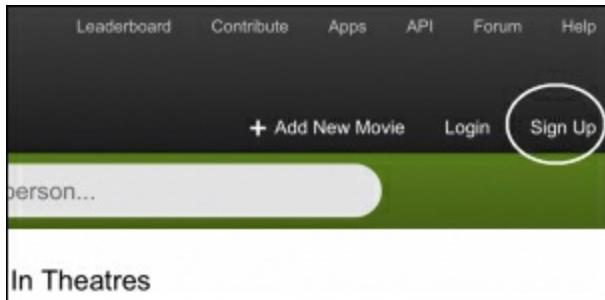
The next project is about searching for information about a movie. A user will be able to type the name of a movie in the search box and receive a result with the movies found according to their enquiry.

Using a public API, we can simplify our work as we don't need to register any movie information on a database or create a server with its own database and HTTP server.

Setting up the project

Before we start creating the application, we have to register on a website that has an API connected to a movie database; once there, we will be able to query for a movie and retrieve some information about it. The website that offers us this public API is called **The Movie Database (TMDb)**.

Open your favorite web browser and type <https://www.themoviedb.org> in the address bar. Once the website has been loaded, click on the **Sign Up** button, as shown in the following screenshot:



Fill in the form that comes up next, and click on the **Sign Up** button. After this, you need to verify your e-mail address by opening the e-mail that TMDb sends you, and click on the link that verifies your e-mail address.

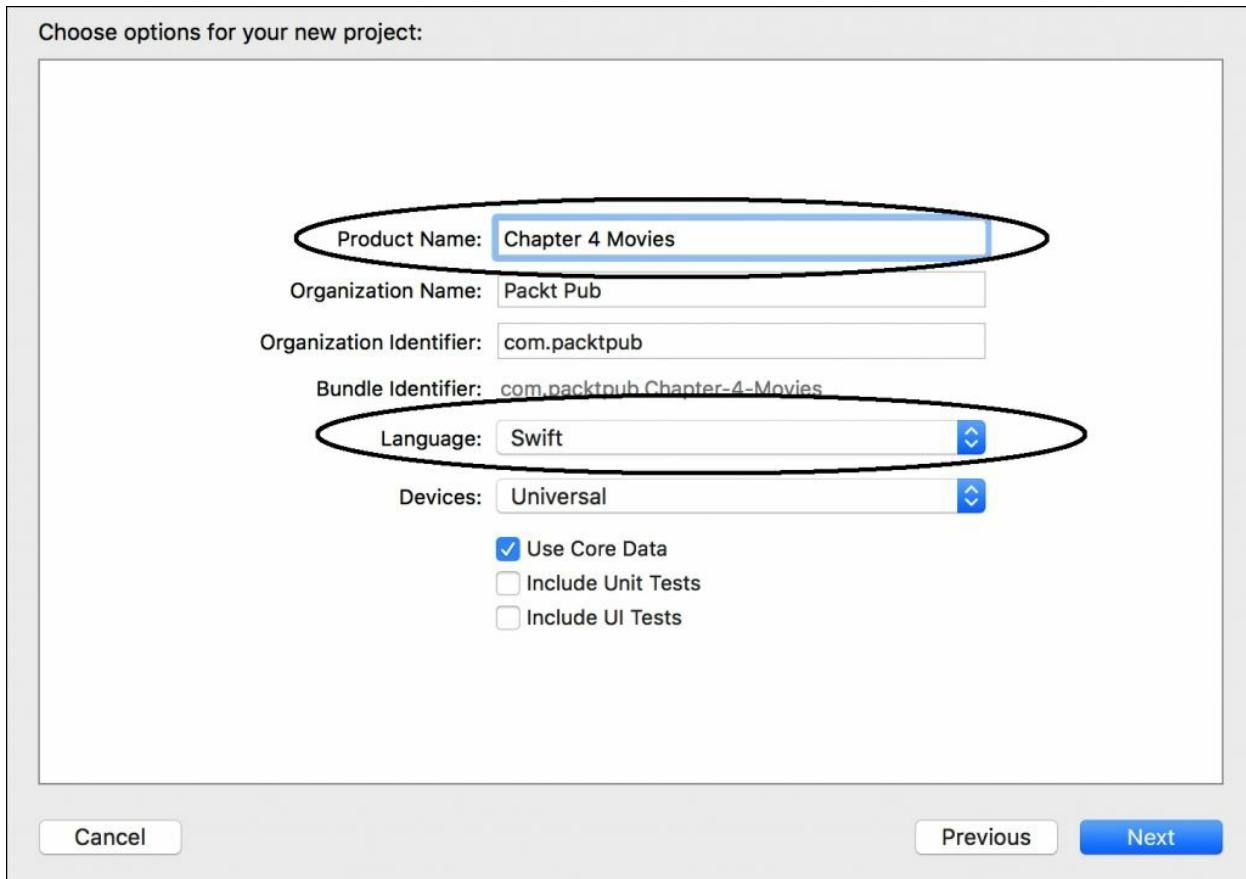
Once you have verified your e-mail address, you can sign into your account, and click on the **API** option that is on the left-hand side of your panel. Click on detail, and register for a new API key. Again, you have to fill in a form explaining your application and so on, and wait until it is approved.

Note

If you want to distribute an application that uses the TMDb API, you can do it for free; however, the number of requests are limited to 30 requests every 10 seconds, and you also have to show that it used TMDb somewhere in your application.

Once your key request is approved, you have to copy the keys as these will be needed in our application.

Now, you can open your Xcode, and create a new **Single View Application**. Call it `Chapter 4 Movies`, and ensure that **Swift** is the main language, as demonstrated in the following screenshot:

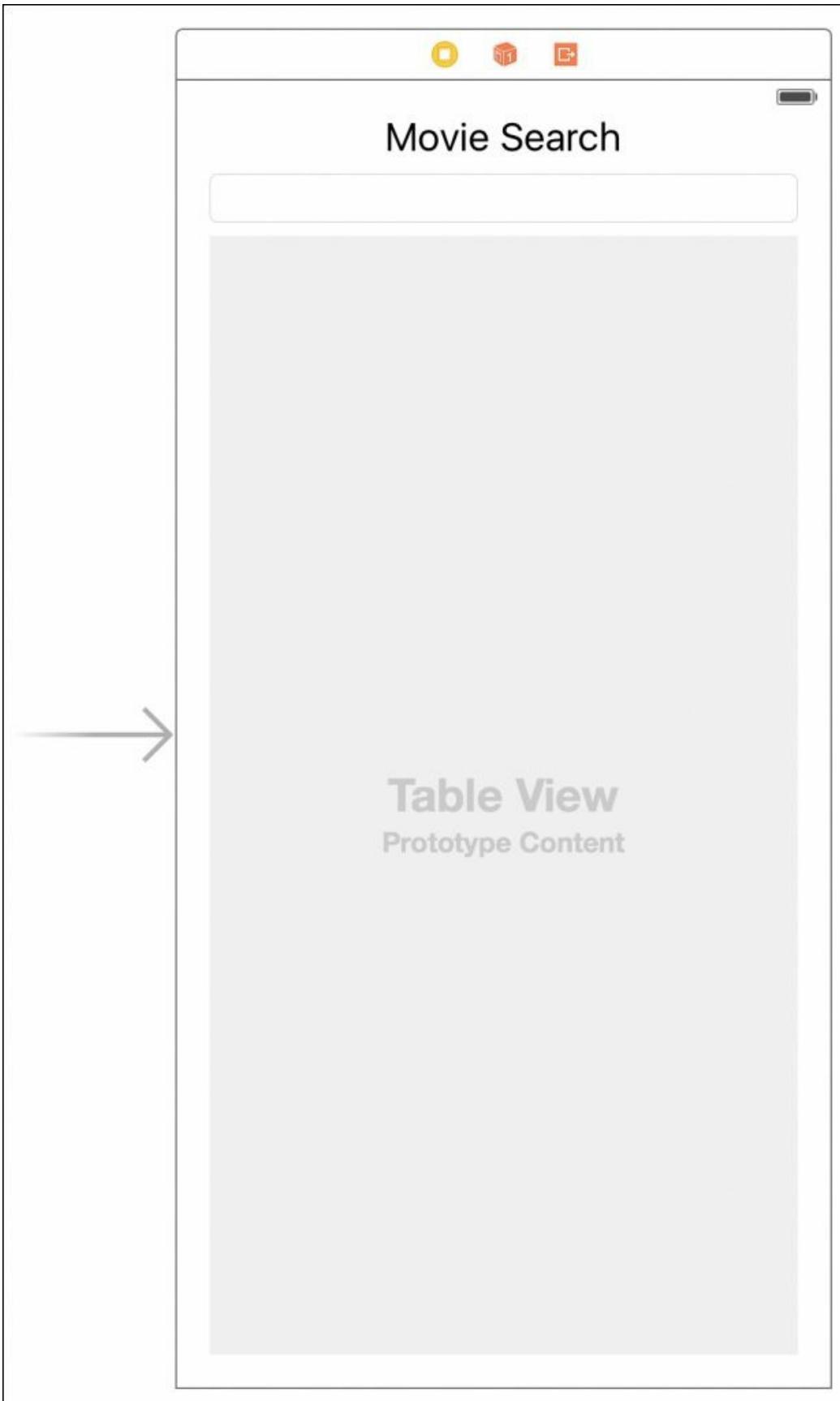


Select a destination folder and save your project. Install ReactiveCocoa in your favorite way, as shown in [Chapter 1, Introduction to Reactive Programming](#).

Once your project has the ReactiveCocoa framework installed, set your project orientation to **Portrait** only, and go to the storyboard and change the size class to portrait, as we did in the previous chapter. The final result should be similar

to what is shown in the following screenshot.

Add add a label, text field, and a table view to your scene. Set the label **Title** property to `Movie Search`, and organize these UI components as shown in the following screenshot. Don't forget to add the AutoLayout constraints.



Connect the text field and the table view to their respective attributes, as shown in this code:

```
@IBOutlet weak var textField: UITextField!
@IBOutlet weak var tableView: UITableView!
```

Great! Everything is prepared to start coding.

Searching for a movie

Now, it is time to start coding. We are going to start with one basic code where we will just subscribe to the text field. Eventually, we will improve this code according to our needs. Go to the `viewDidLoad` method, and add one subscriber to the text that logs what the user is typing. To do this, use the following code:

```
override func viewDidLoad() {
    super.viewDidLoad()
    textField.rac_textSignal().subscribeNext({ (input) ->
Void in
    print(input)
})
}
```

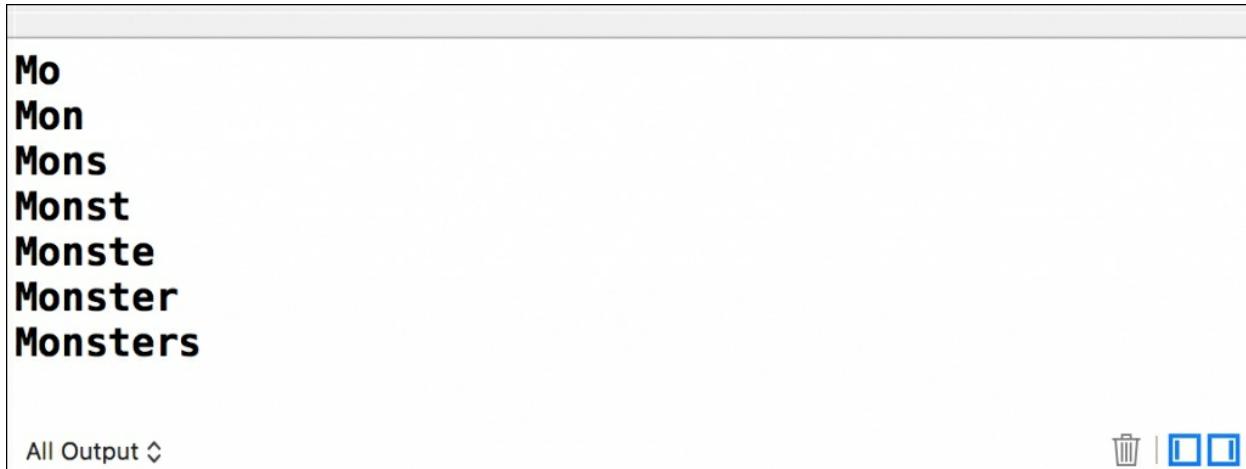
That was easy, but now we need to think about the requirements. Every time the user types a letter, a request should be made. Imagine that the user types just the letter "a"; this is a very small letter that's taken to perform any query. Remember that a request implies consuming broadband, processing its response, and displaying the result. It's not worth doing it for just one letter. (I'm sorry for films like Z and Q.)

What we have to do is filter the signal so that the subscriber acts only when the user types a word that is two or more letters long. We will do this with a method called `filter`. This method receives a closure with the signal input (in this case, the user text) and returns a Boolean value that indicates whether the current value is accepted or not. Let's update our code by adding the following highlighted code:

```
textField.rac_textSignal()
    .filter({ (input) -> Bool in
        let text = input as! String
        return text.characters.count >= 2
    })
    .subscribeNext({ (input) -> Void in
        print(input)
    })
}
```

Rebuild and run your application, type a movie name, and take a look at the log console; you will see that a few lines will be printed out, except one-letter lines. The following screenshot shows the result when searching for

Monsters:



The screenshot shows a log console window with a light gray background. On the left, there is a vertical scroll bar. The main area contains the following text, which is a list of partial movie titles starting with 'Mo':

```
Mo
Mon
Mons
Monst
Monste
Monster
Monsters
```

At the bottom left of the window, there is a small button labeled "All Output" with a downward arrow. At the bottom right, there are three icons: a trash can, a square with a diagonal line, and a square with a double-lined border.

If you think that each line is a new request, we still have too many. Increasing the number of letters in the `filter` function won't help us very much as there are movie titles that are two letters long, such as *Up* or *It*. A better solution for this is to request only when the user stops typing, which means that the application must wait a little bit after the last received signal, and then it must be processed.

That's when `throttle` comes into the picture. This method takes a number of the `NSTimeInterval` type as an argument, also known as `Double`, that represents the number of seconds that need to be passed before the last signal in order to call the next part. In this case, we are going to set this time to `0.6` and see what happens. Feel free to change this value according to your preference. Let's complete our code by adding the following highlighted code:

```
textField.rac_textSignal()
    .filter({ (input) -> Bool in
        let text = input as! String
        return text.characters.count >= 2
    })
```

```
.throttle(0.6)

    .subscribeNext({ input) -> Void in
        print(input)
    })
}
```

Now, rebuild and run the application. Once it is open, type the name of any movie that you would like to search for again, and take a look at the log console and to check whether we can reduce a few lines; now, your result will be similar to what is shown in the following screenshot:



The screenshot shows a terminal window with the following content:

```
mo
mon
monsters
```

At the bottom left, there is a "All Output" button with a dropdown arrow. At the bottom right, there are three icons: a trash bin, a square, and a double square.

Finally, we can start concentrating on performing our request. We are going to do this in two different ways.

Creating signals

The first method to create a signal is by creating it ourselves; this way, we can see how to deal with different occasions when there is no specific signal for a task. The second one is by using the existing method.

First, let's see how to create our own signal. In this case, we will create a signal for the request in a different method. To do this, start by adding a new method to the current View Controller. We will call this method `requestSignal`; it will receive a string that contains the user query, and it will return a signal. Let's start by adding the following code to our View Controller:

```
private func signalForQuery(query:String) -> RACSignal {  
}
```

As we need to return a signal, we have to instantiate one. To do this, you just need to call the `createSignal` method, which belongs to the `RACSignal` class. This method receives as argument on closure that has `RACSubscriber` as its argument and returns `RACDisposable`. Once we have this information, we can start with the instantiation by adding the following highlighted code to our method:

```
private func signalForQuery(query:String) -> RACSignal {  
    return RACSignal.createSignal(  
        { (subscriber:RACSubscriber!) -> RACDisposable! in  
    })  
}
```

`RACSubscriber`, as you already know, is an observer that acts according to the signal response. It can receive a *next* message, which means that the signal has sent some information, an *error* message, which means that something has gone wrong, or a *complete* message, which means that the signal is complete.

An `RACDisposable` object contains information on what to do when the signal is destroyed. For example, if you have a KVO subscription in a View Controller, you have to remove the observer when the View Controller is

dismissed. In this case, as we are executing an HTTP request, we'll have to cancel such a task.

After understanding these concepts, we can create the URL following the instructions of the TMDb API. The URL for this search is

`https://api.themoviedb.org/3/search/movie`, the method is `GET`, and it has two parameters: `api_key` and the user `query`. We will leave the API key as a constant; just copy it from the Web, and replace the following one with the one that TMDb has given you:

```
private func signalForQuery(query:String) -> RACSignal {  
    let apiKey = "a22160e11a5de46dc792f6d2fa8b434a"
```

Although we use the user query as an argument, we have to consider some URL requirements. For example, URLs don't accept whitespace; instead, you have to convert every whitespace to its corresponding percentage code, which is `%20`.

We can perform this conversion manually by iterating through every character in the query string; however, there is an easier and faster way of doing this. Strings have a method called

`stringByAddingPercentEncodingWithAllowedCharacters`, which already performs this task for us. Thus, we can create another constant, which is the encoded user query:

```
let encodedQuery =  
query.stringByAddingPercentEncodingWithAllowedCharacters(.URLH  
ostAllowedCharacterSet())!
```

Finally, we can create a constant that represents the full URL based on the previous constants:

```
let url = NSURL(string:  
"https://api.themoviedb.org/3/search/movie?api_key=\n(apiKey)&query=\\"(encodedQuery)\\"")!
```

Great! Right now, we have the required constants and the call needed to create a signal; therefore, we can retrieve the `URL session` object and its task:

```

        return RACSignal.createSignal(
            { (subscriber:RACSubscriber!) -> RACDisposable! in
                let session = NSURLSession.sharedSession()
                let task = session.dataTaskWithURL(url,
completionHandler: { (data, urlResponse, error) -> Void in
                })
            }
        )
    )
}

```

What should we do inside the completion handler? Firstly, we have to detect whether there were any errors; if so, we can send this error to the subscriber using the `sendError` method. In this case, we are going to send the object that we received as an argument; however, there are cases where you have to create a custom `NSError`. Add the following code to the completion handler to send the error to the subscriber:

```

        if let error = error {
            subscriber.sendError(error)
        }
    }
}

```

If there were no errors, we have to convert the result, which is assumed to be a JSON message, into a native object. Again, we can have another error as the JSON conversion might fail. If it doesn't fail, we just need to send the converted object to the subscriber with the `sendNext` method. Place the following code after the last `if` statement we had:

```

        else {
            do {
                let json = try
NSJSONSerialization.JSONObjectWithData(data!, options:
NSJSONReadingOptions(rawValue: 0))
                subscriber.sendNext(json)

            }catch let raisedError as NSError {
                subscriber.sendError(raisedError)
            }
        }
}

```

It doesn't matter whether there was an error or not; we must report to the subscriber that the signal has been completed and it shouldn't expect any other message. To do this, we have to call the `sendCompleted` method after the

closed brackets of the `else` statement:

```
subscriber.sendCompleted()
```

The task handler is complete, but as you know, you have to call the `resume` method to make it work. Finally, we have to return `RACDisposable`; in this case, you may wonder what should be done when this signal is destroyed and the task is still running. The answer to this is very easy: we have to cancel the task. Add the following code outside the completion handler but inside the signal creation:

```
task.resume()
return RACDisposable(block: { () -> Void in
    task.cancel()
})
```

Can we test this code? Not yet, we still need to call this function inside the signal chain we had. But if we already had a signal, how could we switch the signal with this new one? Signals have a method called `flatMap`, which receives the current input, but it must return a signal that takes control from now on.

Return to the `viewDidLoad` method. Here, we have to complete the signal chain by calling the `flatMap` method after `throttle`. Its handler is very simple; just call the method that we've created and return its signal. Update your code by adding the following highlighted lines of code:

```
.throttle(0.6)
.flatMap({ (input) -> RACStream! in
    let text = input as! String
    return self.signalForQuery(text)
})

.subscribeNext({ (input) -> Void in
    print(input)
})
```

Rebuild and run your application. Now, if you are running under iOS 9, you will receive a weird message complaining that the HTTP connection has failed:

```
NSURLSession/NSURLConnection HTTP load failed  
(kCFStreamErrorDomainSSL, -9802)
```

The reason for this is that Apple has decided to accept connections that are very secure in terms of signed certificates. As we can't change the certificates from TMDb, we have to solve this problem in a different way.

Handling errors

As we are receiving an error, we will take advantage of it to trap it and display it to the user. The `subscribeNext` method can be called with a second argument that contains the action that the application has to perform when something goes wrong; in this case, we are going to show `UIAlertController` with an error description. Return to `viewDidLoad`, and add the highlighted argument to the `subscribeNext` method:

```
.subscribeNext({ (input) -> Void in
    print(input)
}, error: { (error) -> Void in
    let alertController =
UIAlertController(title: "Error",
                    message: error.localizedDescription,
                    preferredStyle: .Alert)
    let alertAction = UIAlertAction(title:
"Dismiss",
                                style: .Cancel, handler: nil)
    alertController.addAction(alertAction)

self presentViewController(alertController,
                            animated: true, completion: nil)
})
```

Is there something missing in this code? Let's see: HTTP requests are usually not executed on the main thread, but the alert controller, as part of the UI components, must be executed on the main thread. For this kind of case, we have to ask to deliver the signal on a different thread using the `deliveryOn` method, otherwise, you might get a crash. Place the following highlighted code before the `subscribeNext` method:

```
)  
.deliverOn(RACScheduler.mainThreadScheduler())  
  
.subscribeNext({ (input) -> Void in
```

Once we test the errors that can be trapped, we have to fix the known issue as we need to receive information from the server. Go to a file called `Info.plist` in **Project Navigator** and click on the plus (+) sign located on

the right-hand side of the **Information Property List** record, as shown in the following screenshot:

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
Localization native development re...	String	en
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)

When the new record appears, change its key to `NSAppTransportSecurity` and press *enter*. The **Key** record will be automatically replaced with its full name, **App Transport Security Settings**. Expand this new record by clicking on the triangle located to its left-hand side. Click on the plus (+) sign of this new record and a *subrecord* will appear. Select the **Allow Arbitrary Loads** option and set its value to **YES**. This step is done only once as it is saved in the `Info.plist` file. The final result will look similar to what is shown here:

Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
▼ App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Localization native developm...	String	en
Executable file	String	\$(EXECUTABLE_NAME)

Finally, now is the time we have been waiting for: the execution of our application. This time, you will appreciate that the server answers JSON messages. The way we used to execute a request was the traditional way, using `NSURLSession`, but the same logic can be applied if you use a framework such as **Alamofire**.

This section is done; in the next one, we are going to fill in the table view with the result provided by the server. Thus, we will be able to watch our

requested movies on the screen.

Filling in the table view

There are some UI components that are very commonly used in our applications, such as `UIButton`, `UIDatePicker` and `UITableView`, which is perfect to display different options to the user. This component works with two different protocols: `UITableViewDataSource`, which provides cells to the table view, and `UITableViewDelegate`, which controls some actions and cell settings.

As some table view functions were not written to work in a reactive way, we can't do too much with them.

Note

The data source in this application will be executed in the traditional way and delegate in the reactive way.

Scroll up to your `ViewController.swift` file, and add the following protocols to the `viewDidLoad` class:

```
class ViewController: UIViewController, UITableViewDataSource, UITableViewDelegate {
```

After this, we need to add a new property where we are going to store the list of movies that are retrieved. This property is an array of dictionaries, and it will be filled in when we receive a response from the server:

```
lazy var movieResult = [[String:AnyObject]]()
```

Now, inside `viewDidLoad`, we have to set the delegate as the current object. Add this highlighted code to set it:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.tableView.dataSource = self
```

Then, we can update our subscriber by setting the `movieResult` property and reloading its data. Remove the previous `print` command as we don't need it

anymore:

```
.subscribeNext({ (input) -> Void in
    let result = input as! [String:AnyObject]
    self.movieResult = result["results"] as!
    [[String:AnyObject]]
    self.tableView.reloadData()
},
```

Finally, we have to implement the required method from the `UITableViewDataSource` protocol. Place these methods at the end of the `ViewController` class:

```
func tableView(tableView: UITableView,
numberOfRowsInSection section: Int) -> Int
{
    return movieResult.count;
}

func tableView(tableView: UITableView,
cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell {
    var cell =
tableView.dequeueReusableCellWithIdentifier("cell")
    if cell == nil {
        cell = UITableViewCell(style: .Default,
reuseIdentifier: "cell")
    }
    cell?.textLabel?.text = movieResult[indexPath.row]
    ["original_title"] as? String
    return cell!
}
```

This part is done; confirm that everything is working as expected by rebuilding and running the application. If not, return to the section where it seems to be failing, and check your code again. Then, write a word in the text field, and check whether some movie titles are displayed in the table view like this:

iPhone 5 - iPhone 5 / iOS 9.1 (13B137)

Carrier 6:15 PM

Movie Search

Kung Fu

Kung Fu Panda

Kung Fu Panda 2

Kung-Fu

Kung Fu Panda 3

Leng Waan Tung Ji

You ling shen

Kung Fu and Titties

Kung Fu Killer 2

Kung Fu Killer

Kung Fu Shadow

Once the table view displays data, we can detect whether the user has selected any movies. The strategy when dealing with this kind of a signal is detecting whether one selector (a certain method) was called. The `rac_signalForSelector` method creates a signal based on a selector call.

The method we want to trap is one that is called when the user taps on the table view that (`didSelectRowAtIndexPath`) belongs to the `UITableViewDelegate` protocol and where the current View Controller is our table view delegate; the method and this `rac_signalForSelector` method will be called from `self` (not `self.tableView`).

The `signalForSelector` method has two ways of being called: the first one is by passing a selector as an argument, and the second involves specifying the selector and the protocol where the method comes from. In this case, it comes from a specific protocol (`UITableViewDelegate`) that we shall use to specify both arguments. Once we've received the signal, we will just print the user's action to check whether it is working. Place the following code at the end of the `viewDidLoad` method:

```
self.rac_signalForSelector(Selector("tableView:didSelectRowAtIndexPath:"), fromProtocol: UITableViewDelegate.self)
).subscribeNext { (input) -> Void in
    print("Tap")
}
```

Is this all? The answer is no. We still need to set the current View Controller as the table view delegate, and this assignment *must* be done after the signal creation. Why? This is not ReactiveCocoa's fault; the problem is that internally when the table view receives a delegate, it optimizes it assuming that no modification will be made afterwards. That's the reason you must add the following assignment after the `rac_signalForSelector` method:

```
self.tableView.delegate = self
```

In case you're using a class that inherits from another that already assigns the

table view delegate, such as `UITableViewController`, you have to set the delegate to `nil` and then you have to set it to `self` again.

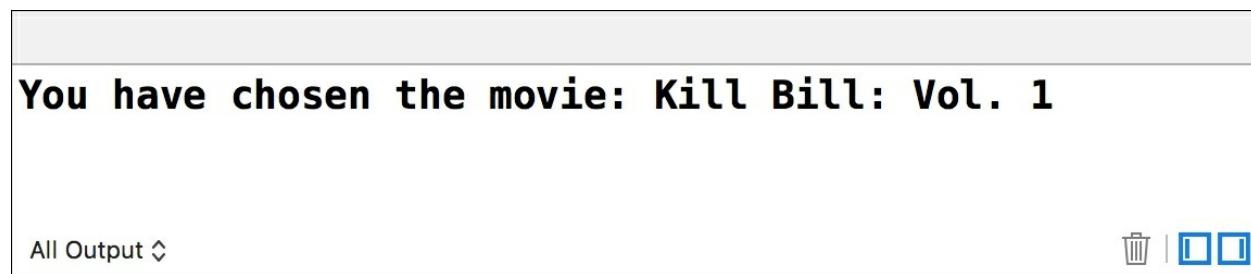
Build and run your application, and check whether when you tap on any movie the log console prints `Tap`. Once you've checked that it is working, you have to think about how you can receive the movie title. The `subscribeNext` method receives an input that is specified as `AnyObject`; however, if you go into this in detail, you will see that it is an `RACTuple`, where each tuple's item represents an argument that's sent to the selector.

The argument we need is the second one, which is of the `NSIndexPath` type. As we learned in the previous chapter, we can access it through the `second` property and cast it to the right object type. Update your implementation, replacing the previous log with the following code:

```
self.rac_signalForSelector(Selector("tableView:didSelectRowAtIndexPath:"), fromProtocol: UITableViewDelegate.self
).subscribeNext { (input) -> Void in
    let arguments = input as! RACTuple
    let indexPath = arguments.second as! NSIndexPath
    let title = self.movieResult[indexPath.row]
    ["original_title"] as! String
    print("You have chosen the movie: \(title)")

}
```

Again, test your application by running it, searching for a movie, and selecting one. Now, the log console will display the movie you have chosen, as shown in the following screenshot:



The screenshot shows a terminal window with a light gray background. At the top, there is a header bar with a light gray background. Below the header, the text "You have chosen the movie: Kill Bill: Vol. 1" is displayed in a bold black font. At the bottom left of the window, there is a small text "All Output" followed by a dropdown arrow. At the bottom right, there are three icons: a trash can, a square, and a double square.

Model-View-ViewModel bindings

So far, every time we receive a new movie list, we have to reload the table view with `reloadData`. This is okay, but imagine how we could also receive some results using different channels, such as loading from a file. We need to remember that the table view needs to reload the data again, and there will be many `reloadData` throughout our code. This is when the **Model-View-ViewModel (MVVM)** pattern comes into the picture.

MVVM is a pattern that connects directly to a property with a view. It was created with the idea that when a value is changed, its graphical representation should automatically change as well, and this is what we are going to do with the `movieResult` property.

The first step is to add the `dynamic` modifier to the `movieResult` property; it will make this property observable. What are observable properties? These are properties that can be observed when we request them with the `addObserver:forKeyPath:options:context` method. Then, every time such a property is modified, the `observeValueForKeyPath` method will be called. ReactiveCocoa wraps it up in an easier way, which we're going to take a look at. Add this modifier to the following sample:

```
lazy dynamic var movieResult = [[String:AnyObject]]()
```

Now, the `movieResult` property is able to be observed with the help of the KVO pattern. ReactiveCocoa allows us to create a signal that internally uses KVO by calling a method called `rac_valuesForKeyPath`. Using this method is very simple: you just need to send a string with a property name and an observer (such as KVO). Let's create our signal by appending the following line of code before the text field signal:

```
let movieResultSignal =
self.rac_valuesForKeyPath("movieResult", observer: self)
```

Now that we have the signal, we just need to add a subscriber; in this case, the subscriber will ask the table view to reload its data. As it was already doing this earlier, here, we are going to add another log to make sure that this signal

is being called correctly. Place the following code after the assignation of movieResultSignal:

```
movieResultSignal.subscribeNext { (input) -> Void in
    print("Reloading data...")
    self.tableView.reloadData()
}
```

Now, we have to remove the `reloadData` call from its previous location by deleting the corresponding line:

```
.subscribeNext({ (input) -> Void in
    let result = input as! [String:AnyObject]
    self.movieResult = result["results"] as!
    [[String:AnyObject]]
    self.tableView.reloadData() // DELETE THIS LINE
}, error: { (error) -> Void in
```

Finally, we have to test our code, rebuild it, and then run the application. Check whether the table view continues with the same behavior, but now you will receive a log message saying `Reloading data....`

Displaying movie posters

TMDb provides us with movie posters, which is something that the user might like. Who doesn't remember Casablanca's poster? Here, we are going to take a look at a better example of what we have learned and also of the idea of change propagation.

To retrieve any posters, firstly, we have to call another endpoint that gives us the URL and sizes accepted by the server. This endpoint is `/configuration`, which replies to a JSON message. The `images` key in this JSON message contains a dictionary with the information needed to download images from the TMDb website. Inside the `images`' dictionary, you can find `secure_base_url`, which contains the base path to download a picture, and `poster_sizes`, which contains arrays of different sizes; we will use the first element of this array.

As the key is the same, move the `apiKey` constant from the outside of `signalForQuery`, which is now a property of the `ViewController` class. Then, create a new method called `getPosterSignal`, which returns `RACSignal`, as shown here:

```
private func getPosterSignal() -> RACSignal {  
}
```

The implementation of this method could be very similar to `signalForQuery`; however, this time we are going to do this in a different way. `NSURLSession` has an extension that contains a method called `rac_dataWithRequest`, which returns `SignalProducer`. A signal producer is very similar to a signal except that it doesn't do anything if there is no subscriber. In this case, we are going to convert it into `RACSignal` with the following code:

```
let url = NSURL(string:  
"https://api.themoviedb.org/3/configuration?api_key=\n(apiKey)"!)  
let request = NSURLRequest(URL: url)  
let producer =  
NSURLSession.sharedSession().rac_dataWithRequest(request)  
return RACSignal.createSignal { subscriber in  
    let selfDisposable = producer.start { event in
```

```

        switch event {
            case let .Next(value, _):
                subscriber.sendNext(value)
            case let .Failed(error):
                subscriber.sendError(error)
            case .Completed:
                subscriber.sendCompleted()
            case .Interrupted:
                break
        }
    }

    return RACDisposable {
        selfDisposable.dispose()
    }
}

```

Have a look at this code, and think about the kind of problems we can encounter. Sometimes, network requests fail. This may be because the device switches between networks, the server is overloaded, or any other reason. So, in case of receiving an error instead of handling it and telling the user that there was an error, we might have another try. The `retry` method is executed for cases like this, and we can use it now that we know that we may come across such problems. If we get an error, we could ask to retry it up to five times; this is the reason we are going to call the `retry` method. Place the following code after the `subscribeOn` call:

```
.retry(5)
```

So far, the signal has only `NSData`, which we know contains a JSON message; now, we can map this `NSData` into a Swift object. Again, we have to call our friend, the `map` function:

```

.map({ (input) -> AnyObject! in
    let data = input as! NSData
    let json = try!
    NSJSONSerialization.JSONObjectWithData(data, options:
    NSJSONReadingOptions(rawValue: 0))
    return json
})

```

The dictionary returned, but the `map` function is okay and we can work with it;

however, we are more interested in a string with the base URL already composed for us. Thus, we have to map the dictionary into a string again:

```
.map { (input) -> AnyObject! in
    let dictionary = input as! [String:AnyObject]
    let images = dictionary["images"] as!
    [String:AnyObject]
        let baseUrl = images["secure_base_url"] as!
    String
        let posterSizes = images["poster_sizes"] as!
    [String]
        return "\\"(baseUrl)\\"(posterSizes[0])"
}
```

This method is done. The next part involves using this signal to store its result in a property. Create a new property called `posterUrl` of the optional `String` type. This property must be dynamic as once it is changed, it might start requesting for the posters. Create this property like this:

```
dynamic var posterUrl:String?
```

Now, we can subscribe to the poster signal, and when we get a reply, we can just set the `posterUrl` property. Inside the `viewDidLoad` method, after setting the data source, we can set the subscription with the following code:

```
self.getPosterSignal()
    .subscribeNext { (input) -> Void in
        let url = input as! String
        self.posterUrl = url
    }
```

If we have the poster URL, it means that we are able to download them. Who knows which posters need to be downloaded? The posters have to be downloaded according to the cells that are being displayed on the screen; therefore, every time we create a new cell for the table view, it must subscribe to the `posterUrl` property. When the property has a value different from `nil`, we must request for the posters, and when the request is made, we can update the current cell. Easy, isn't it?

Scroll down to the `tableView:cellForRowAtIndexPath:` method; here, we have to indicate that we want to observe the `posterUrl` property, and for

every change, we would like to try to download the poster. First, let's observe this property by calling `rac_valuesForKeyPath` after assigning the movie title:

```
self.rac_valuesForKeyPath("posterUrl", observer: self)
```

Then, we have to map the poster's URL with the final image. Calling the `map` function might resolve it very easily as we can create `NSData` with a URL and `UIImage` with `NSData`:

```
.map { (input) -> AnyObject! in
    if let baseUrl = input as? String, poster =
self.movieResult[indexPath.row]["poster_path"] as? String, url =
NSURL(string: baseUrl + poster),
        data = NSData(contentsOfURL: url) {
            return UIImage(data: data)
        }
    return UIImage()
}
```

What do we do next? We have to apply the received image into the current cell. Remember that once it is set, we have to `setNeedsLayout` to apply the new image:

```
.subscribeNext { (input) -> Void in
    let image = input as! UIImage
    let cell =
tableView.cellForRowAtIndex(indexPath)
    cell?.imageView?.image = image
    cell?.setNeedsLayout()
}
```

Is anything wrong here? Not necessarily wrong, but something can be improved. Have a look at the `map` code, and check whether it how much time is taken to download each poster. If this is done in the main thread, the UI might get clunky. How can we solve this problem? If we tell the signal that it has to be created and executed in a low-priority queue, everything will be solved. To do this, we have another method called `subscribeOn`, which receives the queue (this is called `RACScheduler` here) where the signal will be executed. Place the highlighted code between `rac_valuesForKeyPath` and the `map` call:

```
    self.rac_valuesForKeyPath("posterUrl", observer: self)
        .subscribeOn(RACScheduler(priority:
RACSchedulerPriorityLow))

        .map { (input) -> AnyObject! In
```

We've solved one problem and created another one: now the subscription is executed in a background thread, but changing the UI is something that must be done in the main thread. The solution for this is to call the `deliverOn` method, which can switch queues. Type the highlighted code between the `map` and `subscribeNext` calls:

```
}
```

```
.deliverOn(RACScheduler.mainThreadScheduler())
```

```
.subscribeNext { (input) -> Void in
```

Does this work? Of course, it does! Press play and search for a movie; try to search for your favorite one, and you should get some posters in the table view. The following screenshot demonstrates this:

iPhone 5 - iPhone 5 / iOS 9.1 (13B137)

Carrier 12:59 AM

Movie Search

Star Wars



Star Wars



Star Wars: Episode VII - T...



Star Wars: Episode V - Th...



Star Wars: The Clone Wars



Star Wars: Episode III - Re...



Star Wars: Episode II - Att...



Star Wars: Episode VI - Re...



Star Wars: Episode I - The...



Star Wars Marathon Event



Star Wars: Episode IX

Someone might think that it is working because the poster URL is received before we receive any results. This is not true, and there is a simple way that we can test it. Before subscribing to the poster signal, let's wait for 20 seconds using the `delay` method. This gives us enough time to allow us to search for movies and receive some results; then, all you have to do is wait a little bit and voilà! The posters appear. Place the highlighted code to perform this test:

```
self.getPosterSignal()
    .delay(20)

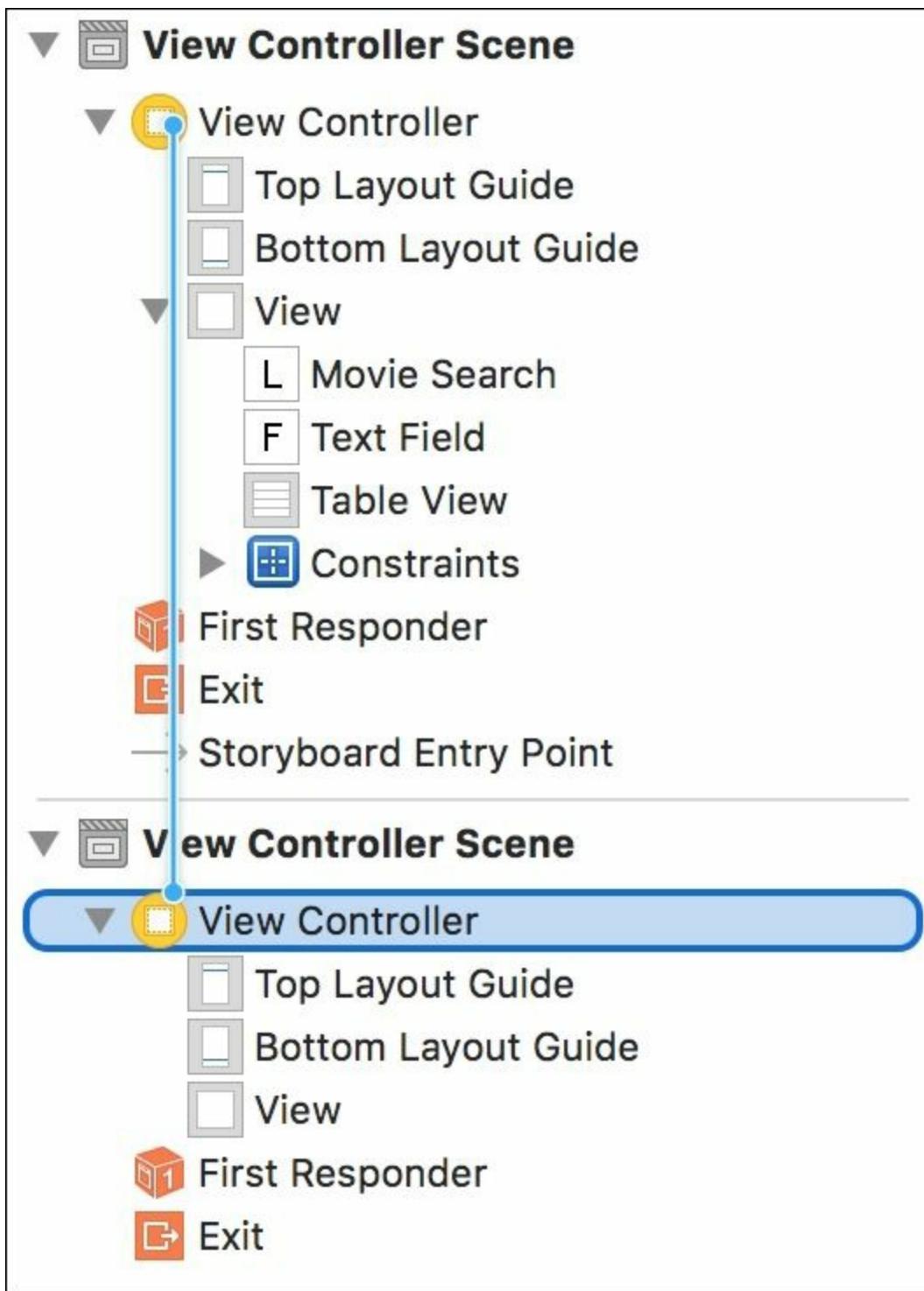
    .subscribeNext { (input) -> Void in
```

Something magical has just happened; here, it doesn't matter what comes first, they can be synchronized afterwards.

Improving your code for a second scene

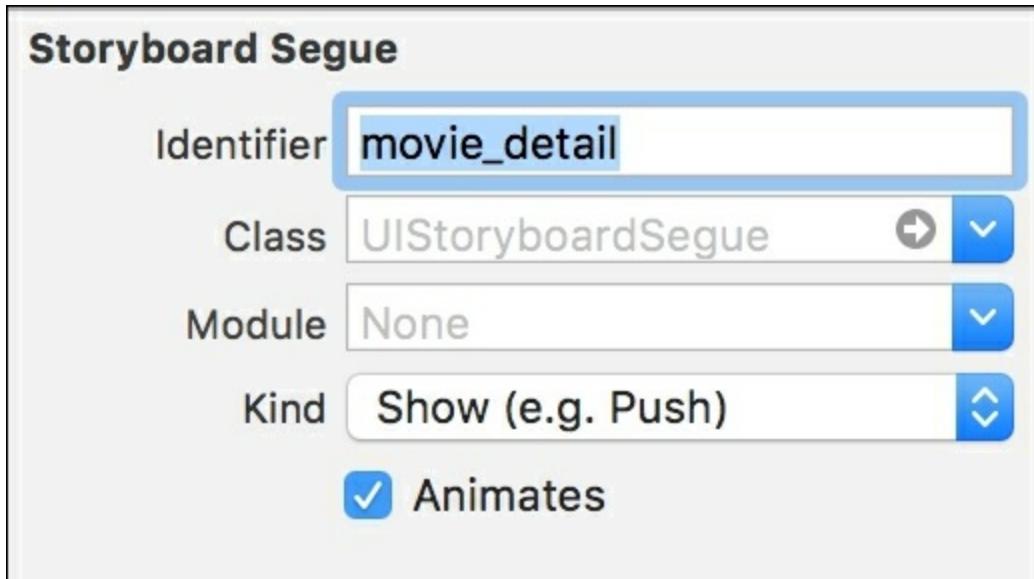
The code works fine but, as you know, a real app always require new features, fixing bugs, new UI...blah blah blah. Imagine that in this case our boss/client/stakeholder/wife tells us that displaying the movie details is crucial.

How can we do this? Firstly, let's add a new empty scene to the storyboard. Click on the storyboard file and add a new View Controller to it. Then, open the document outline and right-click from the first View Controller to the View Controller of the new scene, as is shown in the following screenshot:



When the pop-up menu appears, select the **Show** option. You will notice that a new segue has been created from the first scene to the new one. Select the

created segue (the arrow that connects both view controllers), go to its **Attribute Inspector** with *command + option + 4*, and set its identifier to `movie_detail`, as shown in the following screenshot:



Now, go back to `ViewController.swift`. Here, we are going to split the table view selection into two parts; the first one is the signal itself and the second one is creating of the subscriber. The idea is basically mapping this signal into a signal that sends the selected movie dictionary.

Once we have this in mind, we can start replacing the previous code (the one that recognizes the selector `tableView:didSelectRowAtIndexPath:`) with this one:

```
let tableViewSignal =
self.rac_signalForSelector(Selector("tableView:didSelectRowAtIndexPath:"), fromProtocol: UITableViewDelegate.self).map({
(input:AnyObject!) -> AnyObject! in
    let arguments = input as! RACTuple
    let indexPath = arguments.second as! NSIndexPath
    return self.movieResult[indexPath.row]
})

tableViewSignal.subscribeNext { (input) -> Void in
```

```

        let movie = input as! [String : AnyObject]
        let title = movie["original_title"] as! String
        print("You have chosen the movie: \(title)")
    }

```

Someone might ask, "What's is the difference?" As you can see, first we just create a signal for the `tableView:didSelectRowAtIndexPath:` selector and map its result to a dictionary that represents the selected movie. In a separated instruction, we get this signal and use it for printing the selected movie.

In the `subscribeNext` function, we could call a method named `performSegueWithIdentifier`. Our code would end with something similar to the following (don't make this change; it is just an example):

```

tableViewSignal.subscribeNext { (input) -> Void in
    let movie = input as! [String : AnyObject]
    let title = movie["original_title"] as! String
    print("You have chosen the movie: \(title)")
    self.performSegueWithIdentifier("movie_detail",
sender: self)
}

```

This code is fine and it works; however, the new question is this: is there a more Reactive way of doing it? The answer is: yes, there is. Remove this whole `subscribeNext` code, and we will replace it with something called `rac_liftSelector`.

What does `rac_liftSelector` do, and how does it work? Lift selector is the equivalent way of calling a method in ReactiveCocoa; the difference is that rather than calling the method immediately, it calls it every time the arguments, which are signals, send the next value.

Once we have understood its concept, we can call it just in the same place where we were calling the `subscribeNext` method by adding the following code:

```
self.rac_liftSelector(Selector("performSegueWithIdentifier:sender:"), withSignalsFromArray: [])
```

This code isn't complete yet—the array signal is empty. How should we fill it? Basically, we have to add signals whose values are equivalent to the selector argument. In this case, the first argument should be a signal that returns the segue identifier, which is always `movie_detail`.

How can we create a signal that always returns a constant? The answer is very simple: `RACSignal` has a method called `return`, which basically returns a value repeated times. Now that we know about this, we can start completing our code by adding the following highlighted code:

```
self.rac_liftSelector(Selector("performSegueWithIdentifier:sender:"), withSignalsFromArray:  
[RACSignal.return("movie_detail")])
```

Try to compile, and oops, there is an error! The reason for this error is that the function `return` was created in Objective-C and here in Swift, the compiler interprets it as a reserved word. We can solve this problem by surrounding the method name with back ticks, like the following code:

```
self.rac_liftSelector(Selector("performSegueWithIdentifier:sender:"), withSignalsFromArray:  
[RACSignal.`return`("movie_detail")])
```

The array is not completely filled yet; we still need to add the second argument, which is the sender. Here, we can use a small trick: the sender won't be used for anything; therefore, we can send the movie detail as a sender. Therefore, the final code is this:

```
self.rac_liftSelector(Selector("performSegueWithIdentifier:sender:"), withSignalsFromArray:  
[RACSignal.`return`("movie_detail"), tableViewSignal])
```

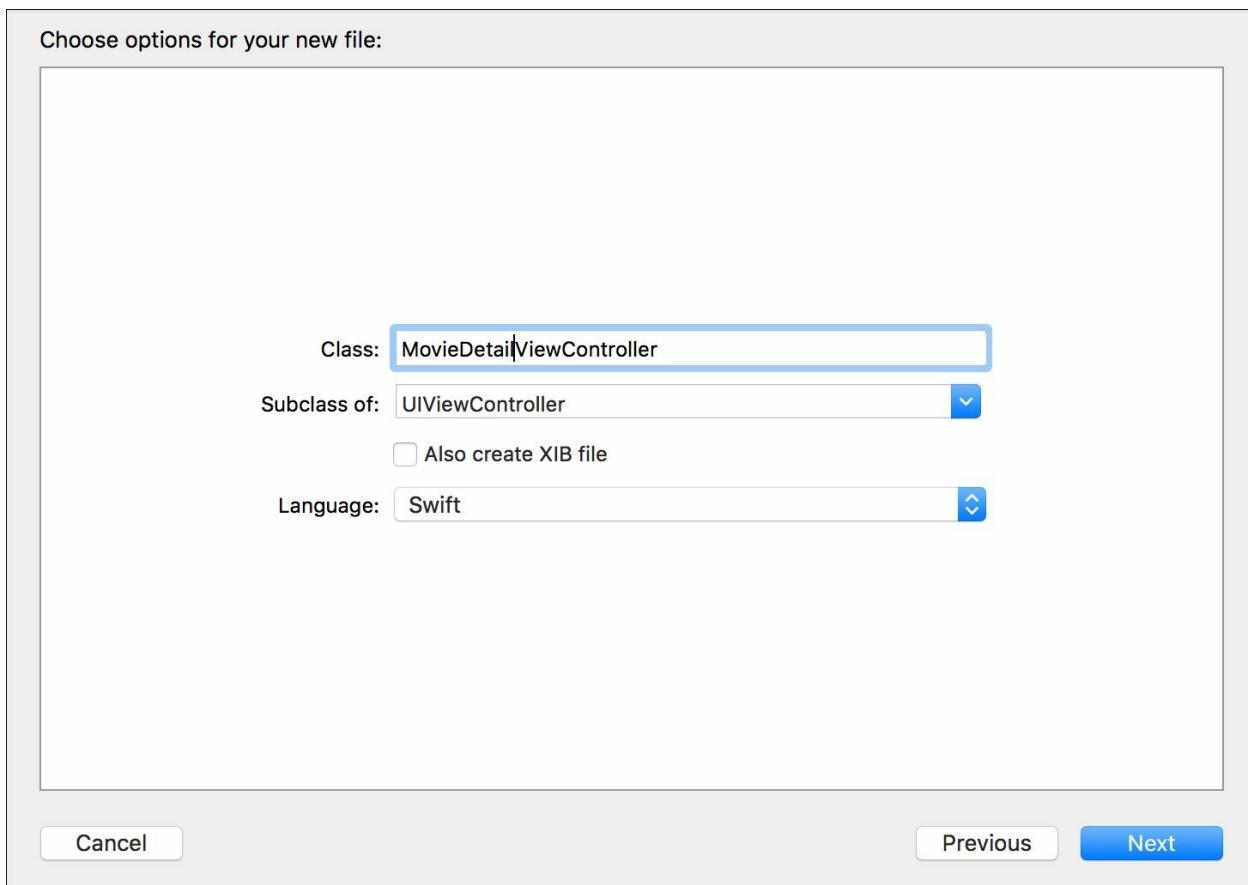
If you don't like this trick of using the movie information as a sender, you can solve it by creating an intermediate method, but this task will be left as homework.

Press play and check whether the second scene is called whenever a movie is selected. Of course, at this stage of the app, it doesn't display any movie information; thus, that's our next task.

Filling in the movie form

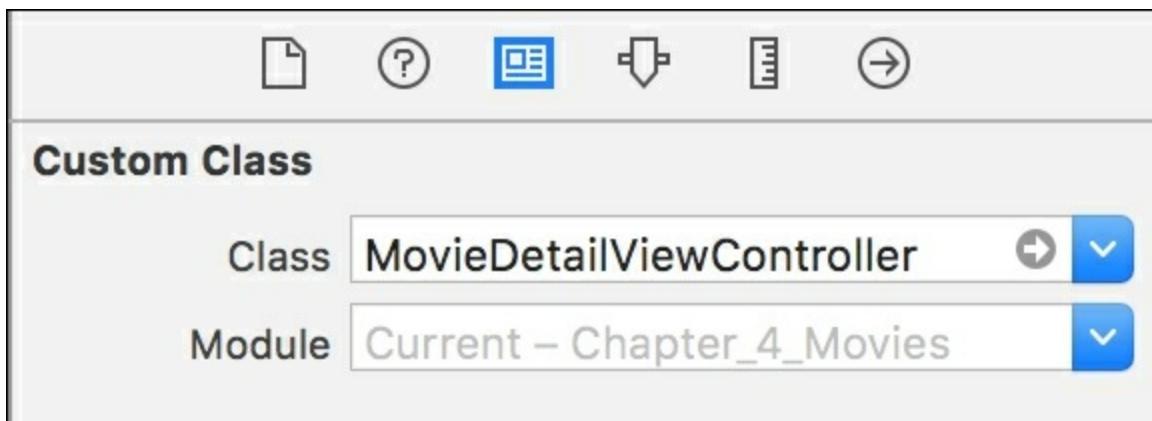
As you might imagine, we have to display the movie information in the new scene; this way, the user will see more detailed information about the movie he has chosen. Start by adding a new file, then select a new **Cocoa Touch Class**, and then, on the next dialog, set the class name to

`MovieDetailViewController` and ensure that it inherits from `UIViewController` and its programming language is **Swift**, as shown the following screenshot:



For this scene, we have to think about what we are going to display to the user. The title is something essential, the poster is good to show (people love pictures), the genre is useful information as it tells you which type of movie we have selected, and the movie overview, which contains the movie's description. Remember that a **Dismiss** button is also important; this way, the user can return to the previous scene.

Once we have a good idea about what we want to display, we can go to the storyboard to design our scene. Start by selecting the new scene and going to its Identity Inspector by using the combination *command + option + 3*. Set its class to `MovieDetailViewController`, as shown in the following screenshot:

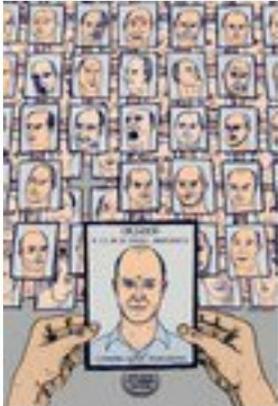


Then, we can start adding new UI components to this scene. Place four labels, one image view, one text view, and one button into this new scene. Organize these components and add the needed constraints until you get a layout similar to this one:

iPhone 6s Plus - iPhone 6s Plus / iOS 9.2 (13C75)

Carrier 7:56 AM

OK, Good



Genre
Drama

Overview

A series of demoralizing auditions and an intense movement workshop push a struggling actor towards the edge.

[Dismiss](#)

After adding the UI components, we have to connect some of them with their corresponding attributes. Open the assistant editor with *command + option + enter* and connect the label that represents the title, the one that represents the genre, the text view, and the button with the following attributes:

```
@IBOutlet weak var titleLabel: UILabel!
@IBOutlet weak var posterImage: UIImageView!
@IBOutlet weak var genreLabel: UILabel!
@IBOutlet weak var overviewText: UITextView!
@IBOutlet weak var dismissButton: UIButton!
```

Do we need any other attributes? At least the movie details, which as we know is a dictionary of `String` and `AnyObject` values. Add the following line to your `MovieDetailViewController` class:

```
var movieDetail:[String:AnyObject]!
```

Eventually, we will need to request the movie's poster and its genre; therefore, we will also need the poster URL, which can be sent from the previous screen, and the API key:

```
let apiKey = "a22160e11a5de46dc792f6d2fa8b434a"
var posterUrl:String?
```

Now, we can start developing the `viewDidLoad` method; starting with the attributes that don't need to be reactive, we can set the movie title and the overview by just assigning them with the following code:

```
override func viewDidLoad() {
    super.viewDidLoad()
    titleLabel.text = movieDetail["title"] as? String
    overviewText.text = movieDetail["overview"] as? String
```

The poster should also be something very straightforward as we just need to assign one image, which is usually small sized; therefore, we can continue with the following code:

```
if let baseUrl = posterUrl, posterPath =
```

```

movieDetail["poster_path"] as? String, url = NSURL(string:
baseUrl + posterPath),
    data = NSData(contentsOfURL: url) {
        self.posterImage.image = UIImage(data: data)
}

```

The next step can be the **Dismiss** button. To dismiss the current View Controller, we have to call the `dismissViewControllerAnimated` method, which receives as argument a boolean value that indicates whether the View Controller should disappear with an animation or not and a second argument that is the completion handler, which in this case will always be nil.

We can use `rac_liftSelector` again for this process. Although the second argument is always nil, which can be easily achieved using `RACSignal.return`, the first argument is not that straightforward, even knowing that it will be always a true value. If we create a return signal, which means that whenever the scene starts, it will be dismissed immediately, we have to synchronize it with the event of tapping on the button. Although, when we press a button, a signal is thrown, but its value is not a boolean. So what's the solution? What if we map the button's touch-up signal into a boolean? Problem solved; now we can continue the `viewDidLoad` method with the following code:

```

let buttonSignal = dismissButton
    .rac_signalForControlEvents(.TouchUpInside)
    .map { (input: AnyObject!) -> AnyObject! in
        return true
    }

self.rac_liftSelector(Selector("dismissViewControllerAnimated:
completion:"), withSignalsFromArray: [buttonSignal,
RACSignal.`return`(nil)])

```

Finally, we need a signal for the genre. First, we have to understand that we have only an array of IDs that represent the genre; these IDs are integers that don't represent anything to the user. We have to request a list of possible genres and check their names against this list.

To make life easier, we will develop a separated method only for requesting

the genre list. Right now, we will assume that this method will be called `getGenreSignal`; that way, we can finish with the `viewDidLoad` method.

Assuming that this signal will be an HTTP request, we have to ensure that it is delivered on the main thread; otherwise, we might have problems updating the UI components. Based on this assumption, we can start coding with the following line:

```
getGenreSignal().deliverOn(RACScheduler.mainThreadScheduler())
```

Then we can subscribe to this signal with `subscribeNext`:

```
.subscribeNext { (input:AnyObject!) -> Void in
```

Next, we need to receive the input; if you check the JSON result, it comes with only one field called `genres` and it is an array of objects. Each of these objects contains two fields, ID and name; thus, all we have to do is filter the objects that are valid to us and create an array with only the genre names. The following code shows exactly how it works:

```
        let json = input as! [String:  
[[String:AnyObject]]]  
        let genres = self.movieDetail["genre_ids"] as!  
[Int]  
        let genre = json["genres"]?.filter({ (element:  
[String : AnyObject]) -> Bool in  
            let id = element["id"] as! Int  
            return genres.contains(id)  
        }).map({ (element: [String : AnyObject]) ->  
String in  
            return element["name"] as! String  
        })
```

Do these map and filter functions belong to ReactiveCocoa? No; however, they belong to functional programming, and you should get used to this type of solution rather than using traditional loops.

Finally, we can create a new string with the genres that belong to the current movie by using `joinWithSeparator`; once this is done, the `viewDidLoad` method has reached its end:

```
    self.genreLabel.text =
genre?.joinWithSeparator(", ")
} // end subscribeNext
} // end viewDidLoad
```

Have a look at this `viewDidLoad` method; how many var declarations do we have? The answer is none inside this method.

Implementing the genre signal

Finally, we have to create the `getGenreSignal` function, which was used in the previous section. Here, as we know, this is a private function that returns a `RACSignal` and it will do a request to a specific URL; thus, we can start opening this function with the following code:

```
private func getGenreSignal() -> RACSignal {
    let url = NSURL(string:
"https://api.themoviedb.org/3/genre/movie/list?api_key=\
(self.apiKey)")!
```

Continue creating a signal, which is the one that will be returned, and using the `NSURLSession` for requesting the genre list with the following code:

```
return RACSignal.createSignal({ (subscriber:
RACSubscriber!) -> RACDisposable! in
    let task =
NSURLSession.sharedSession().dataTaskWithURL(url) {
        (data: NSData?, response: NSURLResponse?, error: NSError?) -> Void in
```

Once the request is done, we have to validate it. We can use the `guard` statement for controlling the input:

```
guard error == nil else {
    subscriber.sendError(error!)
    return
}
guard let data = data else {
    subscriber.sendError(NSError(domain:
"app", code: 1, userInfo: nil))
    return
}
```

Finally, we can convert the received data into a Swift object (or objects); remember that this conversion can still fail, which will make us trap this error and send it as a signal error rather than propagating it as an exception:

```
do {
    let json = try
```

```
NSJSONSerialization.JSONObjectWithData(data, options:  
NSJSONReadingOptions(rawValue: 0))  
    subscriber.sendNext(json)  
    subscriber.sendCompleted()  
} catch let raisedError as NSError {  
    subscriber.sendError(raisedError)  
}  
}  
} // end dataTaskWithURL
```

As you know, a data task doesn't do anything without calling the `resume` method, and we all agree that it is a good time for doing this:

```
task.resume()
```

Finally, we can return the disposable that is required by the function definition. In this case, we have to cancel the request that is being done. After that, this function is done:

```
return RACDisposable(block: { () -> Void in  
    task.cancel()  
})  
})  
} // end getGenreSignal  
} // end MovieDetailViewController
```

This View Controller is already done; however, as you can see, it requires some information that need to be sent from the first scene to the second one.

Changing a few details in the first scene

The second scene is done, but now we have to adapt the first scene in such a way that we can send some information that is needed by the

`MovieDetailViewController` method. Click on the `ViewController.swift` file, and go to the end of the `getPosterSignal` method. As you can see, this signal returns only one URL, which represents the poster; however, now we need two poster sizes, one that is for the table and the second one for the `movie_detail` scene.

Swift allows us to return more than one variable by using tuples; however, Swift tuples are not compatible with the `AnyObject` type. A good solution for this problem is using `RACTuple`. We can create our own `RACTuple` object. When mapping the configuration into a URL, instead of returning only one URL, we can return a tuple of two URLs. Go to the `getPosterSignal` method and replace the `return` statement of the last `map` call with this one:

```
return RACTuple(objectsFromArray: ["\$(baseUrl)\"  
    (posterSizes[0]), "\$(baseUrl)\$(posterSizes.last!)"])
```

As you can see, we can create a `RACTuple` with an initializer that receives an array as an argument:

Now that we've changed the return type of this `map` function, we have to change where it was called. Go to the `viewDidLoad` method and look at `subscribeNext` of `getPosterSignal`, and now replace the previous implementation with this one that stores two URLs:

```
self.getPosterSignal()  
    .delay(20)  
    .subscribeNext { (input) -> Void in  
        let tuple = input as! RACTuple  
        let url = tuple.first as! String  
        let urlBig = tuple.second as! String  
        self.posterUrl = url  
        self.posterUrlBig = urlBig
```

```
}
```

There is a new attribute in the scene; it means that we have to declare it. Let's do it by adding this new line to the beginning of the `ViewController` class:

```
var posterUrlBig:String?
```

Do we have to do anything else? Actually, we need to do another step, which is sending the information from the current scene to the next one. How can we do it? Again, we can use `rac_signalForSelector`, but this time, we have to trap the selector `prepareForSegue:sender:`. This method is called every time we switch from the current scene to the next one. If we filter it for ensuring that the next scene is the `MovieDetailViewController` (in the future, we might have more scenes), then we can send the movie details and the poster URL with the following code, which should be placed at the end of the `viewDidLoad` method:

```
self.rac_signalForSelector(Selector("prepareForSegue:sender:"))
)
    .filter { (input: AnyObject!) -> Bool in
        let tuple = input as! RACTuple
        let segue = tuple.first as! UIStoryboardSegue
        return segue.destinationViewController is
MovieDetailViewController && tuple.second is
[String:AnyObject]
    }.subscribeNext { (input) -> Void in
        let tuple = input as! RACTuple
        let segue = tuple.first as! UIStoryboardSegue
        let viewController =
segue.destinationViewController as! MovieDetailViewController
        viewController.movieDetail = tuple.second as!
[String:AnyObject]
        viewController.posterUrl = self.posterUrl
    }
```

Great, our app is done! Execute it and search for your favorite movie.

Summary

In this chapter, we learned some new concepts. We learned how to create a signal and control it by sending it next values, errors, and completion. Using it in asynchronous calls is a perfect sample of the benefits of reactive programming. Have a look at our code, and make sure that there aren't too many variables or `if` statements.

A good sample of the usage of Reactive Programming is the usage of `rac_liftSelector`, which is a different way of calling methods. Using this function, we just need to define which signal combination is necessary for triggering a method—doesn't matter where these signals were sent.

We also learned how to observe a property, making it propagate its change to other parts of the application. New methods were also displayed here, such as `filter`, `subscribeOn`, `deliverOn`, `retry`, and `delay`.

Now that we have a good amount of knowledge about ReactiveCocoa, let's take a look at how we can test it in the next chapter.

Chapter 5. Enhance Your Application Using RAC Extensions

Even if there is nobody at home, even if the doors and windows are locked, even if you don't hear anything, remember: you are not alone! What does this mean? Though you may know how to develop using ReactiveCocoa, you have to remember that your application might need to use some extra features, such as a sensor, frameworks, and so on.

So far, we have learned how to develop an application with ReactiveCocoa; now, we are going to learn how to use ReactiveCocoa's extensions. This way, we can add more features to our application and still use it with the reactive way of programming.

In this chapter, we will cover the following topics:

- Installing Reactive Extensions (Rx)
- Investigating their usage
- Using ReactiveCoreData

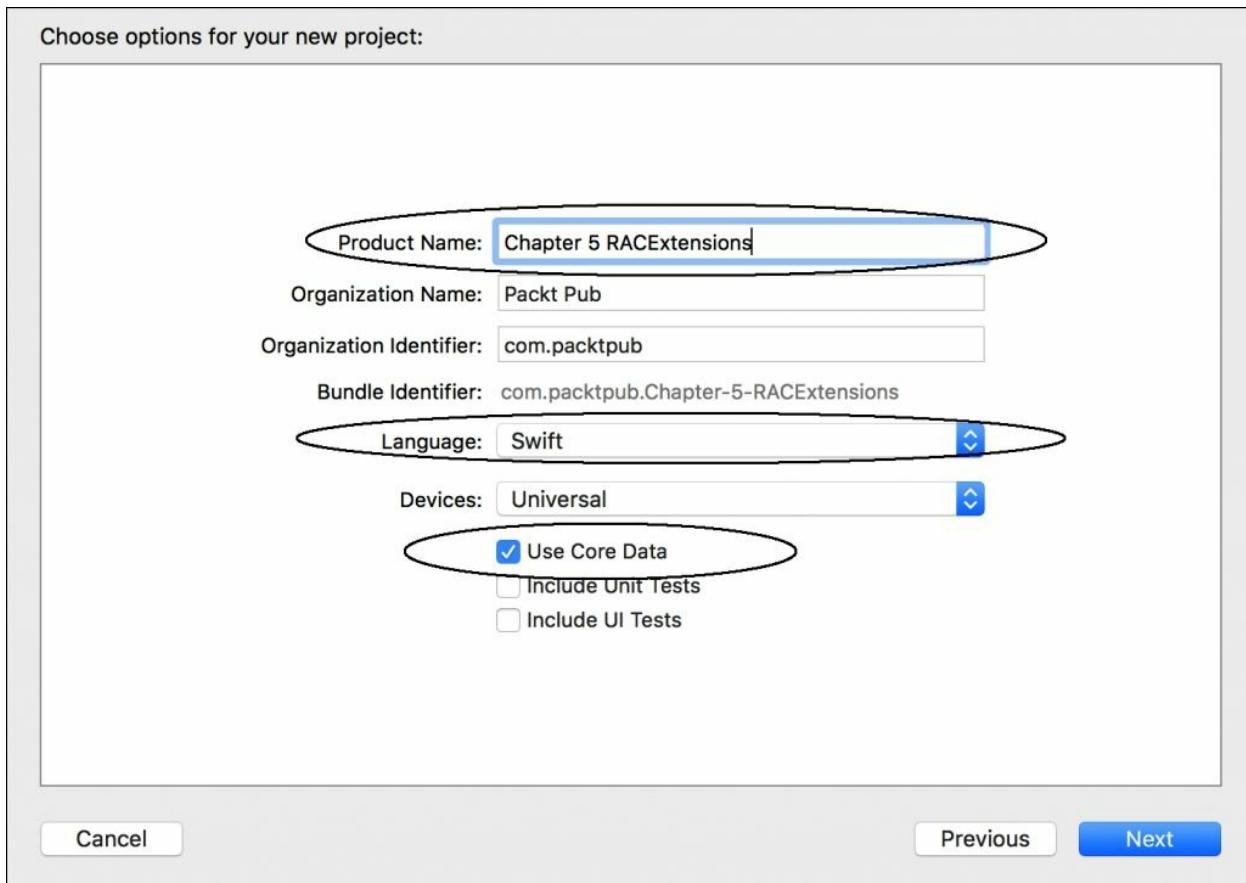
An overview of the project

In this chapter, we will create a small application that takes some pictures automatically when we move at a certain distance. It will save the pictures into the photo gallery and record the coordinates that we have taken these pictures at so that we can check the path afterwards.

For this project, we will use core data through a ReactiveCocoa extension called **ReactiveCoreData** and other auxiliary extensions. Here, you will learn how to deal with ReactiveCocoa extensions as many of them have been executed for different ReactiveCocoa versions, and due to this, they work differently.

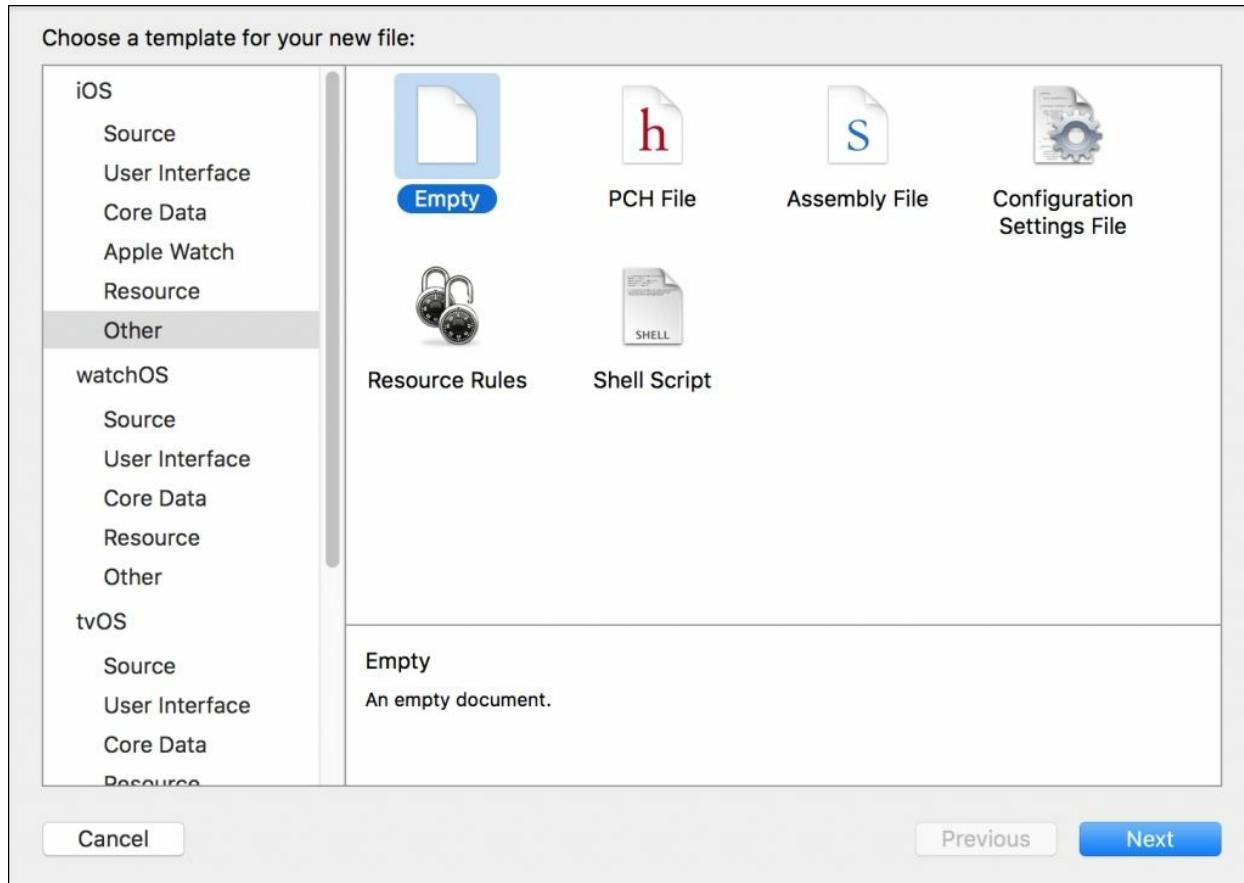
Setting up the project and installing extensions

Open Xcode, and create a new **Single View Application**. Set its name to `Chapter 5 RACEExtensions`, ensure that **Swift** is the main language, and that the **Use Core Data** checkbox is checked:



This time, we are going to install ReactiveCocoa using CocoaPods as it facilitate installing other extensions. As we learned in [Chapter 2, Installing ReactiveCocoa and Using It with Playground](#), using CocoaPods requires the creation a file with the pods that are required in our project.

Close your project but not Xcode. Create a new file with Xcode by pressing *command +N*. This time, select an **Empty** file that is located in the **Other** section, as shown in the following screenshot:



Call this file `Podfile`, and place it in the same folder as the project file (`Chapter 5 RACEExtensions.xcodeproj`). Here, we have to set the pods that we will need for our application:

- **ReactiveCocoa**: It is obvious that we will need this framework
- **DRPReactiveCoreLocation**: A ReactiveCocoa extension that uses a core location using reactive programming
- **UIGestureRecognizer + ReactiveCocoa**: A ReactiveCocoa extension that uses gesture without selectors
- **RACPhotos**: A framework that allows us to save and retrieve photos from a gallery using ReactiveCocoa signals

- **ReactiveCoreData**: An extension that uses Core Data with ReactiveCocoa

Now that we know the extensions, we can complete `Podfile` with the following lines of content:

```
use_frameworks!
pod 'ReactiveCocoa', '~> 4.0.4-alpha-1'
pod 'DRPReactiveCoreLocation'
pod 'ReactiveCoreData'
pod 'UIGestureRecognizer+ReactiveCocoa'
pod 'RACPhotos'
```

Before we continue, remember that there are a few details that we have to bear in mind when installing these extensions or any other ReactiveCocoa extension (or even any third-party framework):

- Some extensions were executed for ReactiveCocoa 2 when it was developed for Objective-C; therefore, you might have to use some methods with different names.
- ReactiveCocoa updates its version faster than most of its extensions, which means that you sometimes have to change the extension's code to make it compatible with the new ReactiveCocoa version. For example, ReactiveCocoa 3 used to have the `|>` operator; now, it doesn't exist anymore along with some other stuff, such as replacing occurrences of `SinkOf<Event<T, E>>` with `Event<T, E>.Sink`.
- Sometimes, the extension doesn't have a `podspec`; thus, you have to download it and place it in your project manually or use the `carthage` package system.
- Swift is a programming language that mutates very frequently. If you have had experience with updating your project from Swift 1 to 1.1 or 1.2, or from 1.x to 2.0, you will know what it means better than anyone else. So, be aware that you may need to upgrade the extension yourself.

Now that we know about the possible issues that we might across, we can open a finder window, use the `command + shift + U` combination to open the `Utilities` folder, and then open the terminal.

Go to your project folder, as we learned in [Chapter 2, Installing ReactiveCocoa and Using It with Playground](#), by typing `cd` followed by the project's folder.

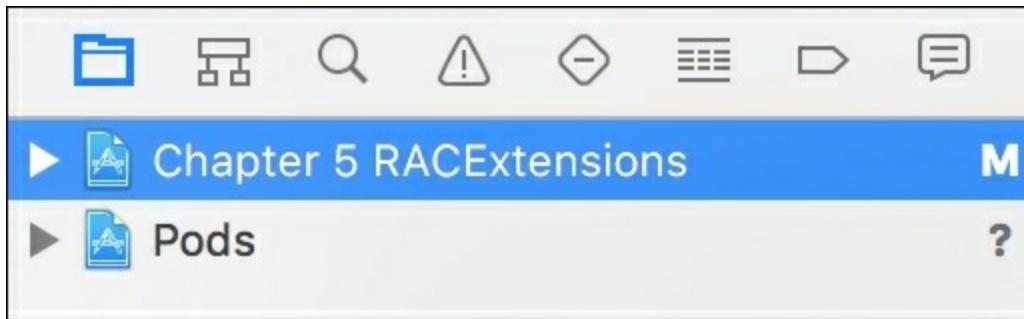
Tip

Remember that the trick to receive the project folder in the terminal is dragging it from the finder window to the terminal.

Press **enter** to enter the desired folder, and type `pod install` followed by the *enter* key.

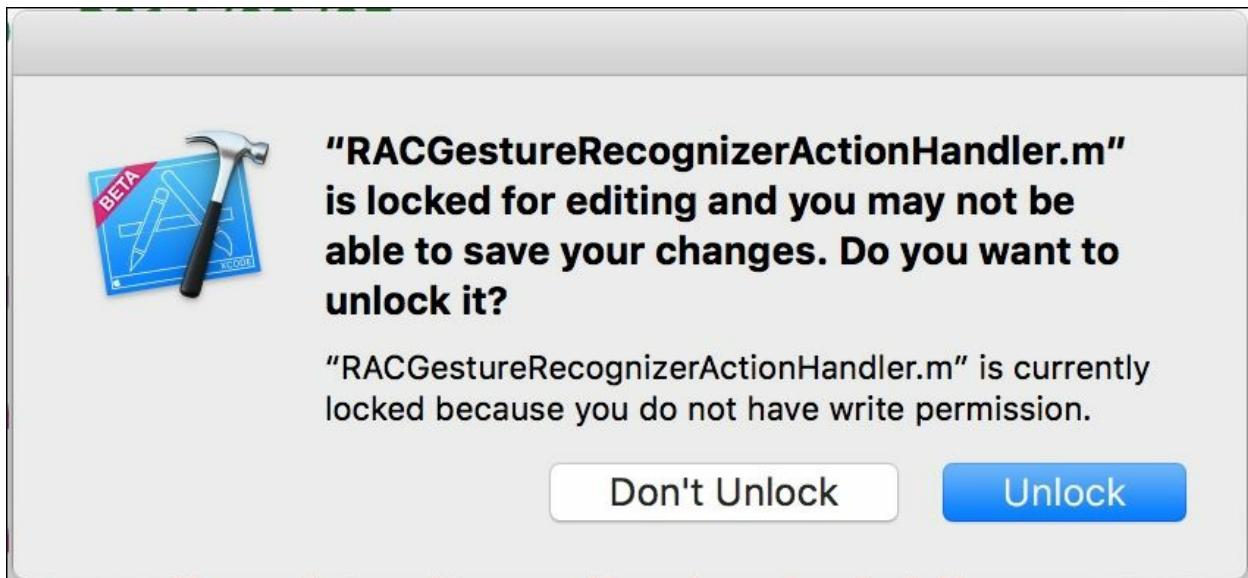
The process will take a few seconds to complete. Once it is done, you can open the current folder by typing `open .` in the terminal (yes, the dot has to be typed as well). A new finder window will appear, and there, you have to look for the file called `Chapter 5 RACEExtensions.xcworkspace`. Double-click on this file to open it.

Once the project is open, you will notice that in **Project Navigator**, you have your project, and under it, you have another project called **Pods**, as shown in the following screenshot:



Compile your project with *command + B*, and a few errors will appear due to the reasons explained earlier. It is possible that you may have to change the angle brackets to quotes, for example, `#import <ReactiveCocoa.h>` may have to be changed to `#import "ReactiveCocoa.h"`. When making this change, a dialog will appear confirming whether you really need to unlock the

file in order to write in it. Click on **Unlock**, and repeat the operation every time it is necessary:



A few other issues might occur due to the extension version that you are using; unfortunately, not all of them are trivial, and you have to update a check by yourself to see whether it works or not.

Tip

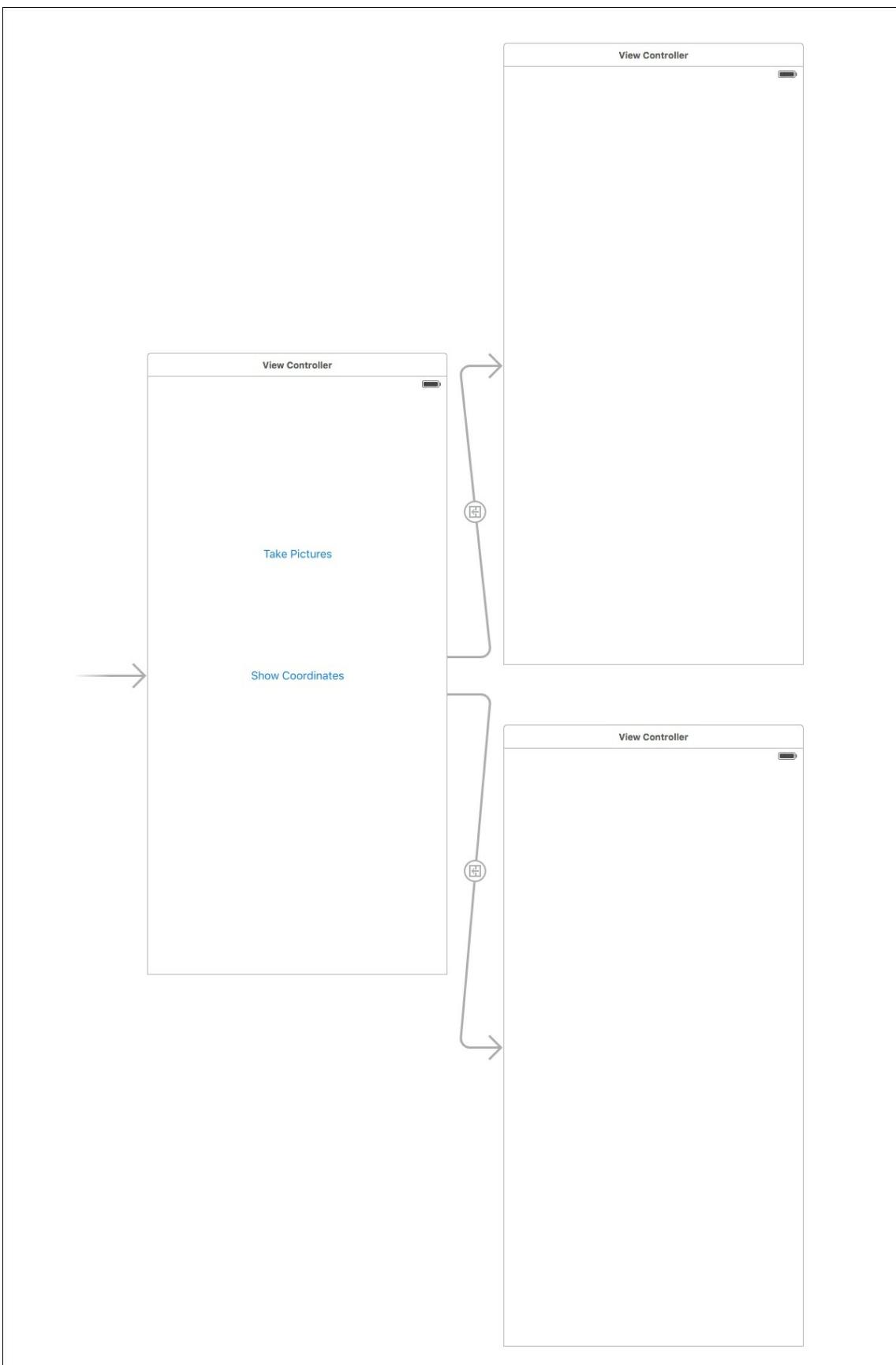
Don't be afraid of getting errors; in the real world, you will get errors, and it's your job to fix them. Try to get used to them as usually the issues repeat.

Once you've fixed every error and the application is compiled and executed, we can say that the setup is done. It is a good time to commit your changes.

Mocking up the first scene

The first scene for this application is very basic, and it has almost no code. The only features that we need are two buttons to open the next few scenes. The first scene allows us to use GPS and start taking pictures, and the second one is to display pictures.

Add two buttons to the current scene, and then add two view controllers to your storyboard. Change the first button's title to `Take Pictures` and the other one to `Show Coordinates`. Now, *control*-drag from one button to one of the new view controllers added to the scene, and select `show` when the options appears. Repeat the same operation with the second button together with the second View Controller. Add some constraints to place the buttons correctly on the screen. The current storyboard should be similar to what is shown in the following screenshot:



Note

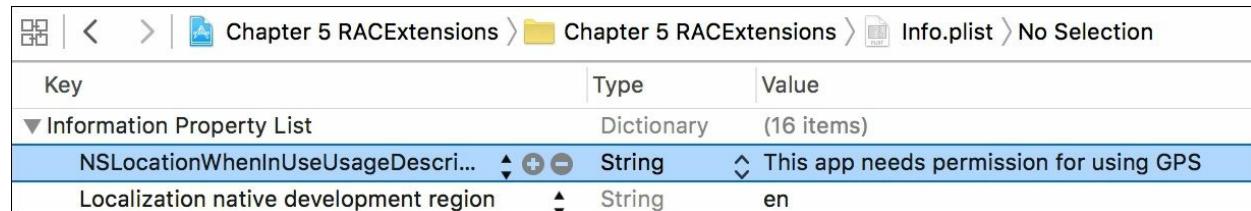
Even a small scene, such as this one, can have some artwork in order to make it more user friendly; however, this task (artwork/UX) is out of scope of this book. Feel free to improve the visual quality of any application in this book.

For this first scene, we don't have to do anything else; therefore, let's move on to the next scene.

Retrieving information from GPS

In this application, GPS will allow us to track where the user has been; this will give us an idea about where the pictures might have been taken. As we are going to use GPS, we need to add a record in `info.plist` that gives the user a message saying that we are going to use this sensor. This is a requirement as of iOS 8.

Click on the `info.plist` file located in your **Project Navigator**. Add a new record and set `NSLocationWhenInUseUsageDescription` as key and as value a phrase like `This app needs permission for using GPS`. Here, you have a sample:



The screenshot shows the Xcode Project Navigator with the path: Chapter 5 RACExtensions > Chapter 5 RACExtensions > Info.plist. The Info.plist file is selected, showing a table with two entries:

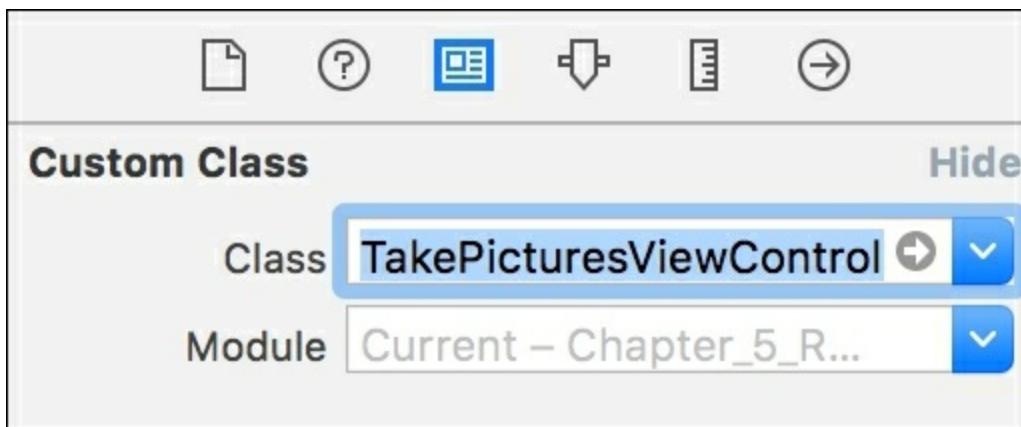
Key	Type	Value
▼ Information Property List	Dictionary	(16 items)
NSLocationWhenInUseUsageDescri...	String	▽ This app needs permission for using GPS
Localization native development region	String	en

We have used `NSLocationWhenInUseUsageDescription` instead of `NSLocationAlwaysUsageDescription` as our extension uses only the first one. If you need to use the second one, you have the advantage of being able to easily change `DRPReactiveCoreLocation` in the source code.

Now, we are ready to create a new View Controller that will use GPS. Using *command + N*, add a new **Swift** file called `TakePicturesViewController.swift`. Just add the basic code to make it visible from the storyboard; this means that you have to import `UIKit` and create a class that inherits from `UIViewController`, as shown in the following code:

```
import UIKit
class TakePicturesViewController: UIViewController { }
```

Return to the storyboard. Here, we have to set the right class for the scene that is connected to the button titled **Take Pictures**; otherwise, it will be connected with the help of a simple `UIViewController`. To do this, click on such a scene, and go to **Identity Inspector** with the *command + Option + 3* shortcut. Change the class name to `TakePicturesViewController`. After pressing *enter*, the module name will change automatically, which is a good sign in case you want to know whether you're typing a valid name or not:



Add two buttons to this scene. Place the first one in the center of the scene and the second one under it. For the first button, set its title to `Start tracking`, and for the second one, just set it to `Dismiss`. Connect these buttons with the following properties:

```
@IBOutlet weak var startTrackingButton: UIButton!
@IBOutlet weak var dismissButton: UIButton!
```

Return to the `TakePicturesViewController.swift` file, and open the `viewDidLoad` method. Starting with a simple task, let's just add an event for the **Dismiss** button. Here, we just need to dismiss the current View Controller when the user taps this button. As we have already learned, we can do this with a method called `rac_signalForControlEvents`. Place the following code to return to the main screen:

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

```

dismissButton.rac_signalForControlEvents(.TouchUpInside)
    .subscribeNext({(_) -> Void in
        self.dismissViewControllerAnimated(true,
completion: nil)
    })
}

```

This code works, but are we missing something? Let's have a look: add the deinitializer to this View Controller with just a log message:

```

deinit {
    print("deinit")
}

```

Press play, open the new scene, and click on the **Dismiss** button. Was deinit called? It doesn't look so. The reason is that we have a retain cycle. This problem is very common when using ReactiveCocoa and you may not be aware of it. This time, let's say that the object itself is not owned by the subscriber handler. Update the subscription code with the following highlighted code:

```

    .subscribeNext({[unowned self] (_) -> Void in

```

Repeat the test, and now check whether the View Controller is dismissed by checking the log window. Once this concept is clear, we could do a visible example. Bear in mind that we are going to use it everywhere.

Now, we can start coding the part that will use the GPS location. If you go to DRPReactiveCoreLocation's official website, which is hosted by GitHub, you won't find very much documentation; therefore, you have to discover how it works by reading the source code for yourself. Some frameworks offer you some sample code, but that's not the case here.

Note

Do not expect to find good documentation or any documentation, for that matter, for any third-party framework (not only ReactiveCocoa extensions). Unfortunately, figuring out how a framework works by reading its code is a task that's commonly performed.

Import the `DRPReactiveCoreLocation` framework by adding the following line of code, and ensure that the project compiles:

```
import DRPReactiveCoreLocation
```

So far, we know that we can use this framework. Now, we can retrieve the location manager provided by the `DRPReactiveCoreLocation` class using the `sharedLocationManager` method, and set it for a property by following this code:

```
let locationManager =  
DRPReactiveCoreLocationManager.sharedLocationManager()
```

Then, we can start requesting updates with a method called `startUpdatingLocation`. This method should be called outside any signal as it also requests permissions. When the track button is pressed, we will start using the camera. Next, we can add the following code to `viewDidLoad`:

```
self.locationManager.startUpdatingLocation()  
  
startTrackingButton.rac_signalForControlEvents(.TouchUpInside)  
  
.subscribeNext {[unowned self] (_) -> Void in  
// TODO complete this part  
}
```

The location manager has a function called `authorizationChangedSignal`, which returns a signal with the authorization status to use GPS. Every time the authorization changes, this signal is triggered. As we just need to enable the tracking button when we have authorization for that. The framework we use accepts only `AuthorizedWhenInUse`, and this is what we should map. Here is the final code for this part:

```
locationManager.authorizationChangedSignal()  
.map({ (input) -> AnyObject in  
let status = input as! NSNumber  
return status ==  
NSNumber(int:CLAuthorizationStatus.AuthorizedWhenInUse.rawValue)  
}).subscribeNext { [unowned self](input) -> Void  
in
```

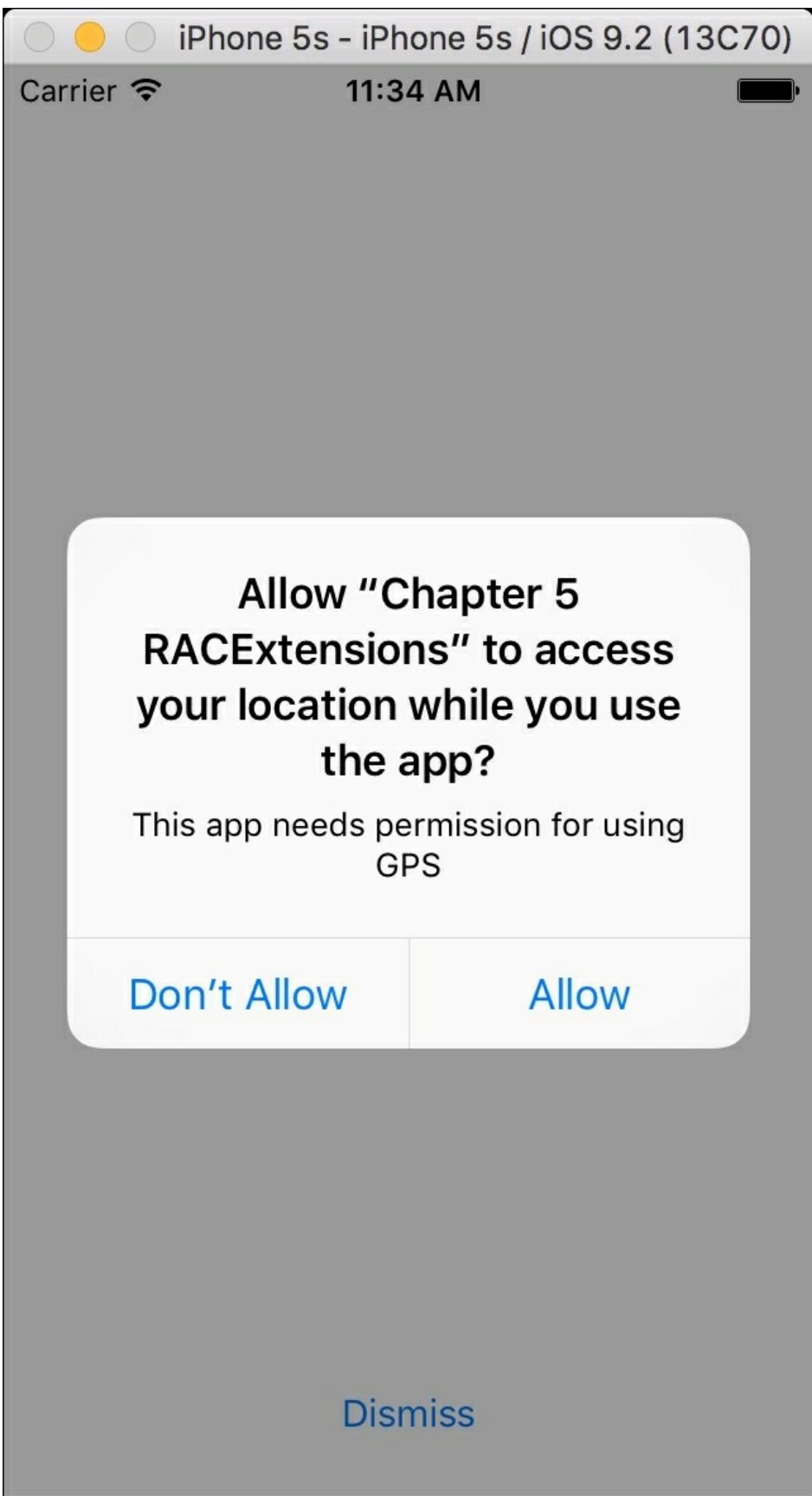
```
    let authorized = input as! Bool
    self.startTrackingButton.enabled = authorized
}
```

If you've never worked with Core Location and have had to deal with its authorization system, you might think that we've typed a lot of code; however, if you have experience with the Core Location authorization and have also used notifications to detect whether the authorization has changed (with an ugly switch), you can appreciate that we haven't written very much code.

Why was it necessary to use the filter input as `NSNumber` instead of using it as an enum? The answer is very simple: remember that this framework was developed in Objective-C where it is very common to use integers as enum. Have a look at the `CLAuthorizationStatus` definition, and you will see that it is defined with this code:

```
typedef NS_ENUM(int, CLAuthorizationStatus)
```

Compile the project and run it. When the first screen opens, click on the **Take Pictures** button. Here, a dialog will appear asking you for authorization (take a look at the following screenshot), and after accepting the authorization, you will receive a log message saying **Authorized**:



Signaling

Once we have a subscriber for the authorization, we can start using another signal for the current position. With the extension that we are using, we can receive this signal using the `locationChangedSignal` method.

This signal is triggered every time we receive a new position; however, the idea is to take pictures at a certain distance. This means that we are not going to take a picture for every received coordinate, rather, we have to compare the current position with the last accepted one and check the difference in their distance.

One easy way of doing this is by adding an attribute in the `TakePicturesViewController` class; however, this is not very functional. Here, what we have to do use a signal method called `scanWithStart`. This method receives an initial value as an argument, which, in our case, is the coordinate with latitude zero and longitude zero.

The second argument is a closure that receives the previous value, the new value, and needs to return the accepted value. We may have to check whether their distance is greater than 10 meters, for example; if so, we can return the new value. If not, we'll return the previous one.

Returning the previous value hasn't solved very much so far. We still have the problem of receiving too many coordinates. What if we accept only unique coordinates? Accepting unique coordinates will allow us to take pictures only once at each coordinate. Again, we could solve this issue by adding an attribute to the current class; however, ReactiveCocoa has a solution for this called `distinctUntilChanged`.

After these steps, we can finally subscribe and print the current location of where we should take a picture. Remember that the first location will be the latitude and longitude zero, which can be skipped as we are not supposed to be exactly in this position. Now, place the following code at the beginning of our current `viewDidLoad`:

```

let locationSignal =
self.locationManager.locationChangedSignal()
    .scanWithStart(CLLocation(latitude: 0, longitude:
0))
        { (old, new) -> AnyObject! in
            if let oldLocation = old as? CLLocation,
newLocation = new as? CLLocation
                where
newLocation.distanceFromLocation(oldLocation) > 10.0
{
    return new
}
return old
}
.distinctUntilChanged()

```

Can you see anything right now? Remember that the signal needs a subscriber and we don't have it. Now, we can complete our tracking button by adding a subscriber. Replace the previous comment with the following highlighted code:

```

startTrackingButton.rac_signalForControlEvents(.TouchUpInside)

    .subscribeNext {[unowned self] (_) -> Void in
locationSignal.subscribeNext({[weak self] (input)
-> Void in
        print(input)
})
}

}

```

Rerun your application, and check whether our log can print one coordinate. If you are using a device with a MacBook, you can start walking with them and check whether you receive a log after walking a little bit. If you are using the simulator, open the **Debug** menu, go to the **Location** option, and select the **Freeway Drive** option; you will see that some coordinates are received every few seconds.

Taking pictures with a camera

Now, it's time to use a camera; therefore, this part must be tested with a physical device. To use the camera, we will need the help of `UIImagePickerController` as it is very easy to use. This component is imported in `UIKit`, but eventually, we will need a constant that is defined in `MobileCoreServices`. This is the reason we have to import this framework:

```
import MobileCoreServices
```

Let's add an image picker controller as the property of `TakePictureViewController`:

```
let imagePickerController = UIImagePickerController()
```

As we now have an instance of the image picker controller, we can set it up and leave it ready to take pictures. This component has a signal called `rac_imageSelectedSignal`, which is triggered when a picture is taken. Right now, we just need to log the picture information to make sure that it is working. Place the following code to set up the camera:

```
imagePickerController.sourceType = .Camera
imagePickerController.allowsEditing = false
imagePickerController.mediaTypes = [kUTTypeImage as String]
imagePickerController.showsCameraControls = false

imagePickerController.rac_imageSelectedSignal().subscribeNext
{ (input) -> Void in
    print("PICTURE \\"(input)"")
}
```

When should the camera appear? When the tracking button is pressed! Okay, let's do this. Just call the `presentViewController` method when the button is pressed, and take a picture every time a new coordinate is received; hence, you will have to update the subscriber again for the pressed button with the help of this highlighted code:

```
    self.locationManager.startUpdatingLocation()
startTrackingButton.rac_signalForControlEvents(.TouchUpInside)

    .subscribeNext {[unowned self] (_) -> Void in

self.presentViewController(self.imagePickerController,
animated: true, completion: { () -> Void in

    locationSignal.subscribeNext({[weak self] (input)
-> Void in
        self?.imagePickerController.takePicture()

    })
}
}
```

Run the application, and ensure that instead of printing a location, we print the picture information. Everything looks great except for one small detail. How can we return to the previous scene? Let's solve this question in the next section.

Using gesture recognizers

We will need to go back to the previous screen when we have finished taking the pictures. What about double tapping? To do this, we can use another framework called `UIGestureRecognizer+ReactiveCocoa`, which was included in our pod file.

This framework works in a very easy way; if you check the official website (<https://github.com/kaiinui/UIGestureRecognizer-RACEExtension>), you can take a look at the documentation and see that it is not very difficult. Basically, you have to call a function called `rac_recognizer`, which returns a signal for the gesture recognizer and also uses it as a gesture recognizer for the setup. As we need a double tap, we just need to set `numberOfTapsRequired`, and add this gesture recognizer to the image picker overlay. Once you have received the signal, you can dismiss the image picker. The following code must be placed at the end of `viewDidLoad` to add the tap recognizer:

```
let doubleTapRecognizer =
UITapGestureRecognizer.rac_recognizer()
doubleTapRecognizer.numberOfTapsRequired = 2

self.imagePickerController.cameraOverlayView?.addGestureRecognizer(doubleTapRecognizer)
    doubleTapRecognizer.rac_signal().subscribeNext
{ [unowned self] (_) -> Void in

self.imagePickerController.dismissViewControllerAnimated(false
, completion: nil)
    }
```

Test your application, and check whether everything is working as expected. Can we improve what we've done so far? Let's take a look. When removing the camera from the screen, we have to unsubscribe from the location manager, otherwise it will continue receiving coordinates.

The `subscribeNext` method returns `RACDisposable`, which allows us to dispose of the subscription. Go to the top of `viewDidLoad` and create a variable called `locationSubscription` as an optional of the `RACDisposable`

type, as shown in the following line of code:

```
var locationSubscription:RACDisposable?
```

Now, let's assign it. When executing the subscription, the track button is pressed, and the image picker appears on the screen by adding the following highlighted code:

```
self.presentViewController(self.imagePickerController,  
animated: true, completion: { () -> Void in  
    locationSubscription =  
    locationSignal.subscribeNext({[weak self] (input) -> Void in  
        self?.imagePickerController.takePicture()  
    })  
})
```

Finally, we can dispose this when dismissing the camera. This means that the gesture subscriber needs to dispose the location subscription with the `dispose` method. Add the highlighted line to dispose the location subscription:

```
doubleTapRecognizer.rac_signal().subscribeNext  
{[unowned self] (_) -> Void in  
    locationSubscription?.dispose()  
  
self.imagePickerController.dismissViewControllerAnimated(false  
, completion: nil)  
}
```

Again, test your application, and check whether everything is still working.

Storing pictures

So far, when we move around with our phone, it takes some pictures but they are just dropped. It is time to store our pictures in the photo gallery. Here, we have to use another extension called **RACPhotos**, which was created by Alejandro Martinez.

RACPhotos is an extension that allows us to use part of the Apple Photos framework (which can store pictures in the **Photos** gallery) with ReactiveCocoa.

At this point, this application starts being a challenge. RACPhotos started with ReactiveCocoa 3.0, and it was updated to the version 4.0, which is different compared to previous versions. Does this mean that we can't use it with other extensions? No, we can use it with other extensions. Even if they were executed for different versions, we just need to be aware of that to make them compatible.

Again, not much documentation is provided; however, you can check the source code and see that most of its code are extensions of `PHPhotoLibrary` and `PHAssetCollection`. Pay attention that there is an enumeration for errors called `RACPhotosError`. There is also a sample code of how to use it, which can be useful.

The **Photos** framework, like the Core Location, requires a user's permission to use it; thus, we need to enable the track button only when both permissions are accepted by the user.

To make the location authorization signal compatible with ReactiveCocoa 4, we have to use a method called `toSignalProducer`, which returns a `SignalProducer<AnyObject?, NSError>`. Don't worry about this type and how it works. We will take a more detailed look at this in the next chapter. After that, we will take the same `map` function we had earlier but return `Bool` instead of `AnyObject`. The `subscribeNext` method can be removed now because this signal will be merged with the photos authorization signal. Once you have understood these concepts, we can replace the

previous `authorizationChangedSignal` code with this one:

```
let locationAuthSignal =
locationManager.authorizationChangedSignal().toSignalProducer()
.map({ (input) -> Bool in
    let status = input as! NSNumber
    return status ==
NSNumber(int:CLAuthorizationStatus.AuthorizedWhenInUse.rawValue)
})
```

Then, we can receive the authorization signal from the photos library, map it to a Boolean, map the error, and then observe it. Why do we have to perform these last few steps? Actually, as we are going to use them with a method called `combineLatestWith`, it requires that both signals be compatible in terms of types and error types. This is the reason we map `RACPhotoError` to `NSError`.

The error observation is also necessary as `RACPhotos` considers any authorization status that's different from the authorized one to be an error. In this case, we have to disable the track button. Don't forget that everything that is done with the UI must be done in the main thread. Now, we can add the following code after retrieving the location authorization signal:

```
let photosAuthorization =
PHPhotoLibrary.requestAuthorization()
.map({ (input:PHAuthorizationStatus) -> Bool in
    return input == .Authorized
}).observeOn(UIScheduler()).mapError {
(racPhotosError: RACPhotosError) -> NSError in
    return NSError(domain: "RACPhotosError", code:
123, userInfo: nil)
}.on(error: {[unowned self] (error:NSError) -> ()
in
    self.startTrackingButton.enabled = false
})
```

We can combine both signals, map the result with an `AND` operation, and enable the button with a subscriber. These few last steps can be easily performed with the following code:

```
photosAuthorization.combineLatestWith(locationAuthSignal).map
{ (photoAuthorized, locationAuthorized) -> Bool in
    return photoAuthorized && locationAuthorized
}.startWithNext { (authorized) -> () in
    self.startTrackingButton.enabled = authorized
}
```

It's time to test the application again.

Note

The track button will be enabled only if you accept using Core Location and **Photos**.

Saving pictures to the photo library

The application is working fine so far; however, it is not doing anything with the pictures. Now, it is time to start saving them in the photo library. To do this, we first need to create a signal to create a collection called "Tracking Collection". Creating a collection with RACPhotos is not difficult: we just need to get a signal with a method called `createCollectionWithTitle` of the `PHPhotoLibrary` class. At the beginning of the `viewDidLoad` method, let's place the following code to receive the signal:

```
let collectionTitle = "Tracking Collection"
let collectionCreationSignal =
PHPhotoLibrary.sharedPhotoLibrary().createCollectionWithTitle(
collectionTitle)
```

The signal will be created at the beginning of the `viewDidLoad` method; however, it will start only when the authorization is received. To do this, we have to add another subscriber to the `photoAuthorization` signal. As we are only interested in when the authorization is accepted, we are going to filter it and add a subscriber. Complete the `photoAuthorization` chain with the highlighted code to create the collection:

```
let photosAuthorization =
PHPhotoLibrary.requestAuthorization()
...
    })
.filter { (authorized) -> Bool in
    return authorized
}.on { (authorized) -> () in
    collectionCreationSignal.start()
}
```

Now, we can save the received image into our collection by replacing the previous code in `rac_imageSelectedSignal`, which was only a print, with a new one. This new code must treat the input as a dictionary, search for the `UIImagePickerControllerOriginalImage` key, which contains `UIImage`, fetch the collection that we created earlier, and save the image using the `saveImage` method. The final code is as follows:

```
    imagePickerController.rac_imageSelectedSignal()

    .deliverOn(RACScheduler.mainThreadScheduler()).subscribeNext {
        (input) -> Void in
            let metaData = input as! [String:AnyObject]
            let image =
                metaData["UIImagePickerControllerOriginalImage"] as! UIImage

            PHAssetCollection.fetchCollectionWithTitle(collectionTitle).startWithNext({ (collection:PHAssetCollection) -> () in
                PHPhotoLibrary.sharedPhotoLibrary().saveImage(image,
                    toCollection:collection).start()
            })
    }
}
```

Another test must be performed on our application. Rebuild the application, install it on a device, and walk around a little bit. After a while, press the home button to return to the home screen, and open the **Photos** application. Make sure that you have some new photos there.

Storing coordinates

Now, it is time to leave some breadcrumbs. At this point, we have to store the path we have been working with all this while. For this feature, we will use Reactive Core Data, which is a ReactiveCocoa extension that uses Core Data.

As we are only going to write the coordinates, we should create a class for this purpose. Remember that the objects that are stored using Core Data must inherit from `NSManagedObject`; therefore, let's start creating this new class.

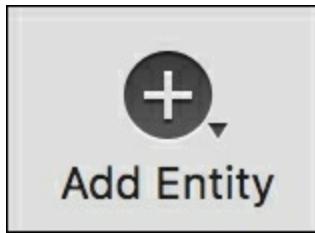
Use *command + N* to add a new file in your project, and call it `Coordinate.swift`. Start importing Core Data to make the `NSManagedObject` inheritance possible:

```
import CoreData
```

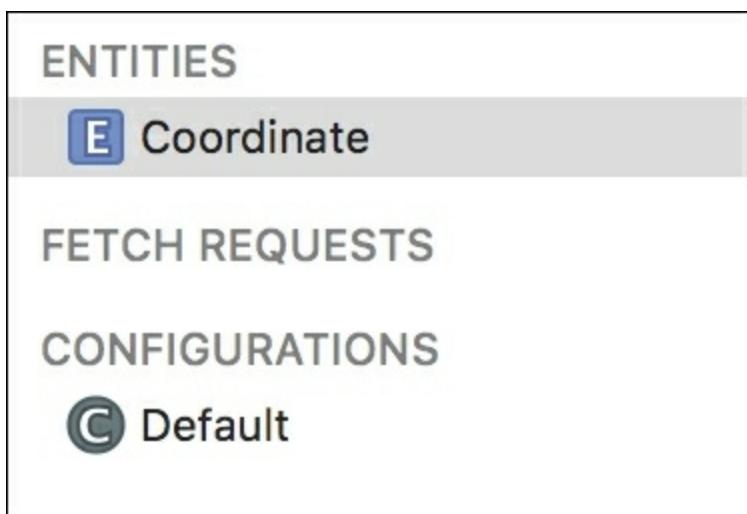
Create a class called `Coordinate` with two properties, `latitude` and `longitude`. Both of these should be `Double` with the `@NSManaged` modifier as they must be stored in the database. Add the `@objc(Coordinate)` modifier to make this class visible with the help of Objective-C, and don't forget that Reactive Core Data was written in Objective-C. The final code is a very small class like this one:

```
@objc(Coordinate)
class Coordinate: NSManagedObject {
    @NSManaged var latitude: Double
    @NSManaged var longitude: Double
}
```

Now, we have to model this data. Click on the `Chapter_5_RACEExtensions.xcdatamodeld` file, which will display a different panel to you. Firstly, you have to create an entity by clicking on the **Add Entity** button:



On the left-hand side of the panel, a new entity will appear. Change its name to **Coordinate** and press *enter*. The left-hand panel must look like what is shown in this screenshot:



Select this new entity, and go to **Data Model Inspector** using the *command* + *Option* + 3 key combination. Change the class name to **Coordinate** as this is the Objective-C class name of the last class we created. **Data Model Inspector** must look like this:

The screenshot shows the Entity configuration dialog box. At the top are three icons: a file, a question mark, and a blue square. Below that is a section titled "Entity". It contains the following fields:

- Name: Coordinate
- Abstract Entity
- Parent Entity: No Parent Entity
- Class: Coordinate
- Module: None

Finally, you just need to add two attributes. Click on the plus (+) sign located at the bottom of the **Attributes** section, and create a new attribute called `latitude` and another one called `longitude`. Change both attribute types to `Double`. This section must look like this:

The screenshot shows the Attributes section. It has a header with "Attribute" and "Type". Below it is a table with two rows:

Attribute	Type
N latitude	Double
N longitude	Double

At the bottom left are two buttons: a plus sign (+) and a minus sign (-).

The modeling is done. Return to the `TakePicturesViewController.swift` file as we are now able to start storing data.

If you've worked with Core Data earlier, you might know that there are a few objects that need to be set up. Fortunately, the code for these objects is already executed in the `AppDelegate` class. We just need to get the managed object context and set it as the main context. After this, we'll be able to use Reactive Core Data. To do this, just place these lines at the beginning of `viewDidLoad`:

```
let appDelegate =  
    UIApplication.sharedApplication().delegate as! AppDelegate  
NSManagedObjectContext.setMainContext(appDelegate.managedObjectContext)
```

This code can be inserted in the `ViewController.swift` file if you like, and you don't need to repeat it in the show coordinate View Controller; however, the idea behind this chapter is to leave it close to the rest of the code; this way, it will be easier to recall.

Now, we just need to complete the location subscription in order to store the data. Before taking the picture, we just need to create a new coordinate with `Coordinate.insert`. Next, we can set its latitude and longitude by copying both from the input argument. Finally, save the context with this new object.

Tip

If you need better performance, you may prefer saving only after a few interactions.

After understanding the steps required to record the current location, we just need to complete `locationSubscription` with the following highlighted lines:

```
locationSubscription =  
locationSignal.subscribeNext({[weak self] (input) -> Void in  
    let location = input as! CLLocation  
    let coordinate = Coordinate.insert()  
    coordinate.latitude =  
    location.coordinate.latitude  
    coordinate.longitude =  
    location.coordinate.longitude
```

```
location.coordinate.longitude  
        try!  
NSManagedObjectContext.currentContext().save()  
  
  
self?.imagePickerController.takePicture()  
    })
```

The scene is done! Press play and walk through your home street for a bit. Right now, you can see the Core Data result; however, if the application doesn't crash, it is a good sign that everything is working fine.

Showing coordinates

Now, we just need to show the coordinates. To do this, we will need a new class as the View Controller. Add a new Swift file to your project and call it `ShowCoordinatesViewController.swift`. Import `UIKit`, the Core Data frameworks and `ReactiveCocoa`:

```
import UIKit
import CoreData
import ReactiveCocoa
```

Create a class called `ShowCoordinatesViewController` that inherits from `UIViewController` and implements the `UITableViewDataSource` protocol. This class will have a property that is an array of coordinates, which will be displayed on the screen; thus, we can start coding this class with the following code:

```
class ShowCoordinatesViewController: UIViewController,
UITableViewDataSource {

    lazy var coordinates = [Coordinate]()
}
```

At this point, the compiler will start complaining about some missing implementations for `UITableViewDataSource`. Let's start with `numberOfRowsInSection`; it must be the same as the coordinates' array size:

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

Next, we need to populate the table view with cells; we just need to use the traditional code that receives a value from the array and use it to set the text for the cell:

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

```

    var cell =
tableView.dequeueReusableCell(withIdentifier: "cell")
    if cell == nil {
        cell = UITableViewCell(style: .Default,
reuseIdentifier: "cell")
    }

    let coordinate = coordinates[indexPath.row]
    cell?.textLabel?.text = "Lat: \(coordinate.latitude),"
Long: \(coordinate.longitude)"

    return cell!
}

```

Return to the storyboard, and add a table view and button to the blank scene we have. Add the required constraints and change its class to `ShowCoordinatesViewController`. Open **Assistant Editor** with *command + Option + enter*, and link the table view and button with two new properties called `tableView` and `backButton`, respectively:

```

@IBOutlet weak var tableView: UITableView!
@IBOutlet weak var backButton: UIButton!

```

Return to the class source code file and open the `viewDidLoad` method. Here, we can start with `backButton` as it just needs to dismiss the View Controller:

```

override func viewDidLoad() {
    super.viewDidLoad()
    backButton.rac_signalForControlEvents(.TouchUpInside)
        .subscribeNext {[unowned self] (input) -> Void in
            self.dismissViewControllerAnimated(true,
completion: nil)
        }
}

```

After the `backButton` action, set the current object as the tableview's data source:

```
tableView.dataSource = self
```

Set the main context for the Core Data with the `AppDelegate` managed object context as we did in the previous scene:

```
let appDelegate =  
    UIApplication.sharedApplication().delegate as! AppDelegate  
    NSManagedObjectContext.setMainContext(appDelegate.managedObjectContext)
```

Now, the coordinate class can give us a signal to retrieve every object of this entity; to do this, we just need to call the `findAll` method. This signal sends the predicate to request for the objects. Then, we can continue with another signal by calling the `executeRequest` method from `NSManagedObjectContext`. We can chain it with `flattenMap`. Finally, the subscriber will receive an array of coordinates and reload the data. Here's the final code for this chain:

```
Coordinate.findAll()  
    .flatMap({ (input) -> RACStream! in  
        let fetchRequest = input as! NSFetchedRequest  
        return  
    NSManagedObjectContext.currentContext().executeRequest(fetchRequest)  
    }) .subscribeNext({ [unowned self] (input) -> Void  
    in  
        self.coordinates = input as! [Coordinate]  
        self.tableView.reloadData()  
    } )
```

Run the application again and open the second scene. Make sure that the table view is populated with the coordinates that you've already taken a look at.

Summary

In this chapter, you learned how to use a few ReactiveCocoa extensions. Extensions allow us to use features that weren't available in first instance using ReactiveCocoa. There are many of them, and some of them were written using old versions of RAC along with a couple of new ones; however, they can still be used together.

We learned how to investigate an extension even if there wasn't very much documentation. This is a good practice as it can give us some idea of how to develop with RAC.

In the next chapter, we will learn how to use ReactiveCocoa in a more modern way, which is done using Swift.

Chapter 6. Using the ReactiveCocoa 4 Style

The first version of ReactiveCocoa for Swift was 3.0; however, it still worked like the Objective-C way of programming, except for a few operators and some other stuff. For this reason, the ReactiveCocoa team decided to create another version quickly taking the advantages of the new features of Swift 2.

In this chapter, we will cover the following topics:

- Creating a signal
- Using a signal producer
- Using the new scheduler

An overview of the project

It is time to use a shopping cart. Have you ever learned about a programming language without a shopping cart? Of course not. After hello world, the shopping cart was invented, and it is now time to develop it with ReactiveCocoa, which will make the building of our application more interesting.

In this project, we will have a shopping application that allows a user to add products to their shopping cart. Here, we have to consider that the products' prices will be set in **Great Britain Pounds (GBP)**, also called Sterling, but the user can change the currency if they wish to.

The currency exchange rate is retrieved from an API called **fixer.io**. Here, we have to remember a few details:

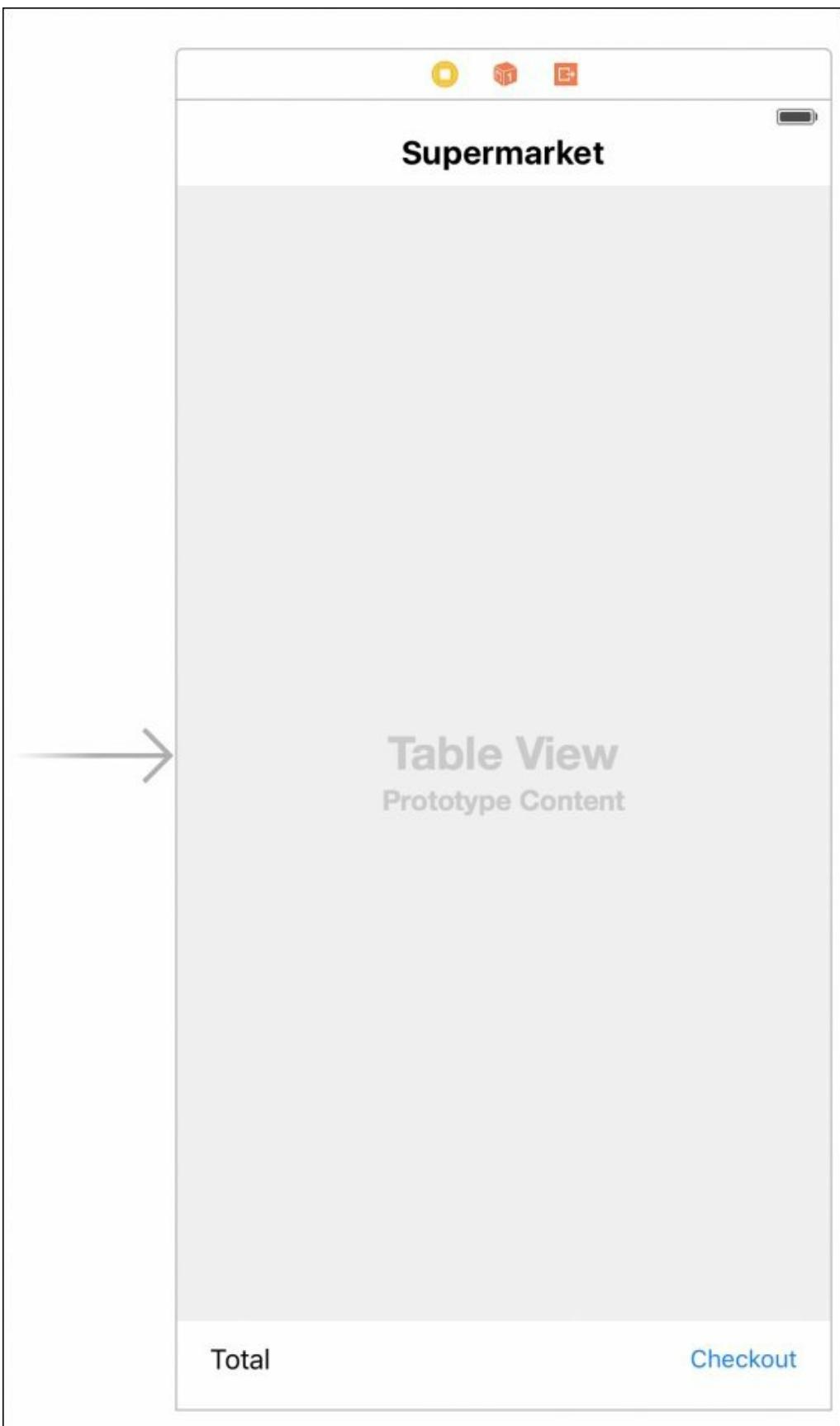
- If there is no Internet connection, the option to change the currency should be disabled. No Internet means no currency change.
- When changing the currency in order to buy a product, everything that is being displayed (and that will be displayed) on the screen must be updated to new values.

Once we have understood these requirements, we can start working on our project.

Setting up the project

Open Xcode, and create a new project called `Chapter 6 Shop Cart`. As usual, ensure that Swift is the main programming language. Install ReactiveCocoa in your favorite way; just ensure that version 4 is installed. For example, if you use CocoaPods, make sure that you specify that it's version 4.

Once you have set up the project, go to the storyboard. Here, we will start by adding a layout where the user can see the whole supermarket catalog on a table view, the **Total** label in the bottom-left corner, and the **Checkout** button in the bottom-right corner. Of course, a label with a screen title would be fine to make it more visible to the user. In short, you can have a simple layout, as shown here:



Now, we have to think about how to display the products. We will have to display a picture, its price, description, a label with the quantity that was added to the shopping cart, a button to add one unit of the product, and a button to remove it.

To do this, just add a table view cell in the only table that we have on the screen. Place an image view, three labels (one for the product name, its price, and the quantity in the basket), and two buttons in the cell. Set the buttons' images to minus (-) and plus (+). You'll find these images in this book's resources. The final cell layout will be similar to what is shown in the following screenshot. Feel free to change this layout if you are a creative person:



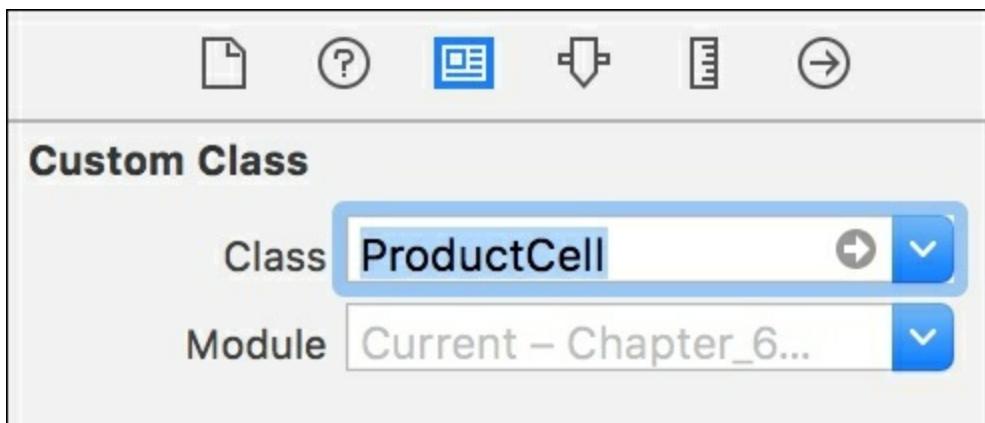
Using **Assistant Editor**, cut the table view, the **Total** label, and the **Checkout** button with their corresponding properties in the `ViewController` class:

```
@IBOutlet weak var catalogTableView: UITableView!
@IBOutlet weak var totalLabel: UILabel!
@IBOutlet weak var checkoutButton: UIButton!
```

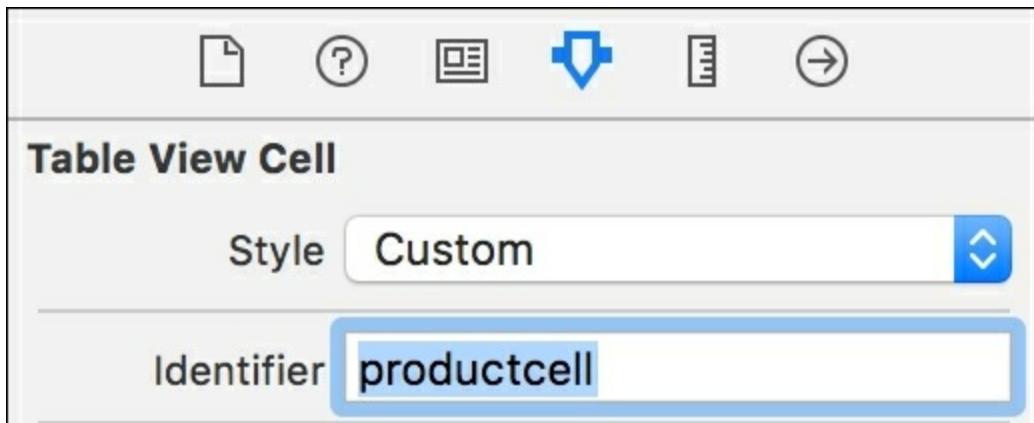
Finally, let's create a class for the table view cell that we have created. Add a new file to your project called `ProductCell.swift`. Import `UIKit`, and create an empty class that inherits from `UITableViewCell` with this code

```
import UIKit
class ProductCell:UITableViewCell {
}
```

Return to the storyboard, select the table view cell, go to its **Identity Inspector** with *command + Option + 3*, and change its class to `ProductCell`, as demonstrated here:



Now, using *command + Option + 4*, go to **Attribute Inspector**, and change the cell identifier to `productcell`:



Open **Assistant Editor**, and connect the UI cell components to their

corresponding attributes:

```
@IBOutlet weak var nameLabel:UILabel!  
@IBOutlet weak var priceLabel:UILabel!  
@IBOutlet weak var productImage:UIImageView!  
@IBOutlet weak var addButton:UIButton!  
@IBOutlet weak var removeButton:UIButton!  
@IBOutlet weak var numberOfItemsLabel:UILabel!
```

Great, the basic setup for the first screen is done. Now, let's start implementing our shopping cart.

Developing the Currency class

In this application, `Currency` is an important class. The output on the screen will change according to the user's currency. The only stored properties we will need are the name, which is a three letter code, such as `USD` for US Dollar and `GBP` for Great Britain Pounds, and its rate according to the base currency.

As mentioned in the beginning of this chapter, the base currency will be `GBP`; however, you can easily change it if you like by changing the setting of the static attribute called `baseCurrency`. The final code for this class is as simple as this one:

```
class Currency: NSObject{
    var name:String
    var rate:Double

    static var baseCurrency:Currency = {
        return Currency(name:"GBP", rate: 1.0)
    }()

    init(name: String, rate:Double) {
        self.name = name
        self.rate = rate
    }
}
```

Why does this class inherit from `NSObject`? The reason for this is that objects of the `Currency` type can only be marked as dynamic if they are Objective-C ready; to do this, you just need to inherit from `NSObject`. We've already seen in previous chapters how properties that are marked as dynamic can be observed when they are changed. As we are interested in observing when a currency changes, we must prepare it to be dynamic.

Creating the Currency Manager

In this application, we need a singleton class that controls everything related to currencies, such as available currencies, the base currency, and the user's currency. Add a new file to your project called `CurrencyManager.swift` and open it. As it is a singleton class, let's start by creating the class with its initializer:

```
final class CurrencyManager: NSObject {
    private static var instance: CurrencyManager?
    private static var dispatch_once_token: dispatch_once_t = 0

    static func sharedCurrencyManager() -> CurrencyManager {
        dispatch_once(&dispatch_once_token) { () -> Void in
            CurrencyManager.instance = CurrencyManager()
        }
        return instance!
    }
    private init() {
        super.init()
    }
}
```

What else do we have to add to this class: we need to know the default currency, which is GBP, and also the current currency. The current currency can be changed, and every time it changes, a signal is triggered. This is the reason we have to add a `dynamic` modifier. The next code must be added after the dispatch token:

```
dynamic lazy var currentCurrency: Currency = Currency(name: "GBP", rate: 1.0)
var defaultCurrency: Currency {
    return Currency(name: "GBP", rate: 1.0)
}
```

Now, we need to detect when the current currency changes. The best way to do this is via a signal, which we can retrieve through a function; therefore, we can create a function that returns a signal that observes when the current currency has changed.

Note

As mentioned in previous chapters, we have to use the `rac_valuesForKeyPath` method for observing a value change; however, it returns `RACSignal`, which is the classic way of using signals. You can convert this object into `SignalProducer`; the philosophy of `SignalProducer` is similar to `RACSignal` but it works a bit differently.

Firstly, `SignalProducer` is not a class. It is a struct that has different limitations, such as you can't inherit from a struct, every attribute is considered immutable, and so on. Secondly, `SignalProducer` takes advantage of generics, which is good since instead of receiving input in the form of `AnyObject`, we can map it to the correct type, making our code safer and easier to maintain:

Add the following code to create a signal producer for the currency change:

```
func currencyChangedSignalProducer() ->
SignalProducer<Currency, NSError> {
    return self.rac_valuesForKeyPath("currentCurrency",
observer: self)
    .toSignalProducer()
    .map({ (input:AnyObject?) -> Currency in
        return input as! Currency
    })
}
```

Does this class have to do anything else? We need to retrieve the available currencies from fixer.io. To do this, we are going to create a private method to receive data from `NSURLSession`. Here, we are going to create a signal rather than a signal producer. The difference is that a signal starts whenever it is created, but a signal producer needs a call to the `start` method.

As we are creating our own signal, we will call a static method named `pipe`. This method returns two values: the signal itself and an observer, which can send an operation (next, fail, complete, and so on) whenever it is necessary.

Once we have received the signal and the observer, we can request for the URL. When the response is received, we can check whether there is any error

and resend it with the `sendFailed` function. Then, we send the received value to the subscriber with `sendNext`. In both cases, this signal is complete with `sendComplete`. Now that we have understood the theory, let's put it into practice by adding the following code at the end of the `CurrencyManager` class:

```
private func signalForUrl(url:NSURL) -> Signal<NSData,  
NSError> {  
    let (signal, observer) = Signal<NSData,  
NSError>.pipe()  
    let task =  
NSURLSession.sharedSession().dataTaskWithURL(url) {  
(data: NSData?, response: NSURLResponse?, error: NSError?) ->  
Void in  
    if let error = error {  
        observer.sendFailed(error)  
    } else {  
        observer.sendNext(data!)  
    }  
    observer.sendCompleted()  
}  
task.resume()  
return signal  
}
```

After this method, we can create a new method to convert the received `NSData` into an array of currencies. Have a look at how the `map` input is not an object of the `AnyObject` type; it now comes according to the result of the previous operation. Don't forget that we need to add the default currency in our array as it is not returned in the JSON message:

```
func signalForRates() -> Signal<[Currency], NSError> {
    let url = NSURL(string: "http://api.fixer.io/latest?
base=GBP")!
    let signal = self.signalForUrl(url)
        .map { (input: NSData) -> [String: AnyObject] in
            let json = try!
                NSJSONSerialization.JSONObjectWithData(input, options:
                    NSJSONReadingOptions.MutableContainers)
            return json["rates"] as! [String: AnyObject]
        }.map { (input: [String : AnyObject]) -> [Currency]
in
    var currencies = [Currency]()
    for (key, value) in input {
        currencies.append(Currency(name: key, rate: value))
    }
    return SignalValue(currencies)
}
```

```
        currencies.append(self.defaultCurrency)
        for (key, value) in input {
            let currency = Currency(name: key, rate:
value as! Double)
            currencies.append(currency)
        }
        return currencies
    }
    return signal
}
```

The code for Currency Manager is now complete, and we are able to detect when the user has changed the currency and apply it anywhere in the application code.

Creating the Product class

A product is something that is not complex; it just needs a few properties, such as the unit type (a bag, can, bottle, and so on), its name, code, image and its price. However, we have to remember that a product has a base price, which is based on the default currency and the user price. This is the price in the current user's currency.

Add a new file called `Product.swift`, and start adding a basic code. Import the `ReactiveCocoa` framework, and create an enumeration to define its unit:

```
import ReactiveCocoa

enum ProductUnitType {
    case Unknown
    case Bag
    case Can
    case Bottle
    case DozenLikeWhenYouBuyBananas
}
```

Now, we have to start creating the `Product` class. This class must inherit from `NSObject` as we will need to call some `ReactiveCocoa` operations, such as `rac_valuesForKeyPath`. If the product price varies, the price that is shown to the user in the desired currency also varies; therefore, it must be observable (dynamic). When the user's price changes, the total of the shopping cart also changes; this means that the user price must also be observable. The following is the start code for the `Product` class with its properties:

```
class Product: NSObject {
    var code:String
    var name:String?
    dynamic var price:Double = 0
    dynamic var userPrice:Double = 0
    var type:ProductUnitType = .Unknown
    var imageName:String?
    var priceSignal:SignalProducer<Double, NSError>!
    var userPriceSignal:SignalProducer<Double, NSError>!
```

The last two signals return `NSError`, which is the default error type when you

convert `RACSignal` to a signal producer; however, you can use `NoError` in cases like this as no errors are expected.

Another property that can help us is the product description, which can be implemented by overriding the computed property description. This belongs to the `CustomStringConvertible` protocol, which is already a part of `NSObject`:

```
override var description: String {
    let unit:String
    switch self.type {
        case .Bag:
            unit = "Bag of "
        case .Bottle:
            unit = "Bottle of "
        case .Can:
            unit = "Can of "
        case .DozenLikeWhenYouBuyBananas:
            unit = "Dozen of ";
        default:
            unit = ""
    }

    let name:String = self.name ?? ""
    return "\(unit)\(name)"
}
```

Next, we will initialize the object. The only mandatory field is the product's code as the other properties are optional or they are already initialized due to their declaration; so, let's start by taking a look at its code:

```
init(code:String) {
    self.code = code
    super.init()
```

Continuing with `priceSignal`, we observe the `price` property, map it to a `Double` value, and every time the price changes, we update the user price:

```
self.priceSignal = self.rac_valuesForKeyPath("price",
observer: self).toSignalProducer().map({ (input) -> Double in
    return input as! Double
```

```

        } )

    priceSignal.startWithNext { [weak self] (price) -> ()
in
    self?.refreshUserPrice()
}

```

Note

Make sure we haven't used `subscribeNext` since we need to use `startWithNext`. Remember that `priceSignal` is `SignalProducer`, which needs to call any `start` method to make it work.

After understanding this concept, we can execute the same task with `CurrencyManager`. Whenever the currency changes, the user price must be changed too:

```

CurrencyManager.sharedCurrencyManager()
    .currencyChangedSignalProducer()
    .startWithNext { [weak self] (currency:Currency) ->
() in
    self?.refreshUserPrice()
}

```

The last part of the initializer involves creating a signal producer for the user price; the idea behind this is the same as `price`, but we use the `userPrice` property instead:

```

    self.userPriceSignal =
self.rac_valuesForKeyPath("userPrice", observer: self)
    .toSignalProducer()
    .map({ (input:AnyObject?) -> Double in
        return input as! Double
    })
} // end init

```

We have to create a method called `refreshUserPrice`. This method is very straightforward as we just need to check the current currency rate and update the user price. Just add the following code after the initializer:

```

func refreshUserPrice () {
    let rate =

```

```
CurrencyManager.sharedCurrencyManager().currentCurrency.rate
    self.userPrice = rate * self.price
}
} // end Product
```

The `Product` class is now complete. The objects of the `Product` type will initially be stored in a class called `Catalog`; this class should be just an array wrapper. Create a new file called `Catalog.swift`, and add the following code for this wrapper:

```
class Catalog {
    private var items = [Product]()
    var count:Int {
        return items.count
    }

    init() {
        var product = Product(code: "1")
        product.name = "Peas"
        product.price = 0.95
        product.type = .Bag
        product.imageName = "peas"
        self.items.append(product)

        product = Product(code:"2")
        product.name = "Eggs"
        product.price = 2.10
        product.type = .DozenLikeWhenYouBuyBananas;
        product.imageName = "eggs";
        self.items.append(product)

        product = Product(code:"3")
        product.name = "Milk"
        product.price = 1.30
        product.type = .Bottle
        product.imageName = "milk"
        self.items.append(product)

        product = Product(code:"4")
        product.name = "Beans"
        product.price = 0.73;
        product.type = .Can;
        product.imageName = "beans";
        self.items.append(product)
    }
}
```

```
}

subscript(index:Int) -> Product {
    return items[index]
}

func indexForCode(code:String) -> Int? {
    for (index, value) in items.enumerate() {
        if value.code == code {
            return index
        }
    }
    return nil
}
}
```

In this code, we have implemented the subscript function, which allows us to use the catalog as an array, such as `catalog[1]`, which will give us the second product in the catalog. We are now ready for the most crucial step: dealing with the shopping cart!

Implementing a shopping cart

A catalog is a collection of products; however, it doesn't control the number of items that the user has chosen. Thus, we need a new class that acts as a collection of products, knows the number of items in it, and has the feature of adding and removing a product. This class is the shopping cart! The shopping cart, like the product, has a total that's based on the default currency and another total based on the user's currency; therefore, it needs two dynamic properties. Items will be stored in `NSCountedSet` as the order that the product was chosen in doesn't matter, only the number of times it was added to the shopping cart.

After knowing these basic concepts, we can start coding the shopping cart. Add a new file called `ShopCart.swift`, import the `ReactiveCocoa` framework, create the `ShopCart` class, and then create its properties:

```
import ReactiveCocoa

class ShopCart: NSObject {
    dynamic var total: Double = 0
    dynamic var userTotal: Double = 0
    var items: NSCountedSet = NSCountedSet()

    private var totalSignal: SignalProducer<Double, NSError>!
    private var userTotalSignal: SignalProducer<Double,
NSError>!

}
```

Continuing with the initializer, the first thing we can do is update the total every time the currency changes. To do this, we just need to take the currency signal producer and add a subscriber with `startWithNext`:

```
override init() {
    super.init()

    CurrencyManager.sharedCurrencyManager()
        .currencyChangedSignalProducer()
        .startWithNext { [weak self] (currency: Currency) ->
() in
            self?.refreshUserTotal()
    }
}
```

Now, we need a function to add and remove a product from the shopping cart. In both cases, we need to update the total and return the number of items that we have in the cart:

```
func addProduct(product:Product) -> Int{
    let currentValue = self.items.countForObject(product)
    items.addObject(product)
    let finalValue = self.items.countForObject(product)
    self.total += Double(finalValue - currentValue) *
product.userPrice;
    return finalValue
}

func removeProduct(product: Product) -> Int{
    let currentValue = self.items.countForObject(product)
    items.removeObject(product)
    let finalValue = self.items.countForObject(product)
    self.total -= Double(currentValue - finalValue) *
product.userPrice
    return finalValue
}
```

A quick question: have we used ReactiveCocoa here? Yes, we have. Whenever you change the total, the total signal is automatically triggered, which updates the user total and triggers the user total signal.

As we've set the user signal as `private`, we have to create a method to access it. It should be as easy as this code:

```
func signalForUserTotal() -> SignalProducer<Double,
NSError> {
    return self.userTotalSignal;
}
```

Finally, we have to implement `refreshUserTotal`, which follows the same algorithm as one for the product we saw earlier:

```
private func refreshUserTotal() {
    let rate =
CurrencyManager.sharedCurrencyManager().currentCurrency.rate
    self.userTotal = self.total * rate;
}
} // end ShoppingCart
```

The models are complete. Now, we can return to the graphical part.

Resuming the ViewController class

The `ViewController` class has been incomplete until now. We have the implementation of `ShopCart` and `Catalog`, so it might be good to start adding them as properties. Go to the `viewController` class, and add the following properties:

```
var catalog = Catalog()  
var shopCart = ShopCart()
```

Now, we have to subscribe to the `ShopCart` signal. Here, we have to be aware that the total might not be delivered on the main thread as we don't know what caused a change in the total. To ensure that we are going to act in the right thread, we use the `observeOn` method, which is equivalent to `deliveryOn` used in `RACSignal`. In this situation, we use `RACScheduler`; however, we now need an object that implements the `SchedulerType` protocol. In this case, there is a class specifically for this purpose called `UIScheduler`.

Note

There are more schedulers, such as `QueueScheduler`, which implement the `DateSchedulerType` protocol and support scheduling for a future date, `ImmediateScheduler` that just executes it, and `TestScheduler` that is used for testing.

Now, we can start the `viewDidLoad` method by updating the **Total** label whenever the currency changes:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
  
    self.shopCart.signalForUserTotal()  
        .observeOn(UIScheduler())  
        .startWithNext {[weak self] (input:Double) -> ()  
in  
    let currency =  
CurrencyManager.sharedCurrencyManager().currentCurrency  
    if let weakSelf = self {  
        let totalText = String(format:"Total:
```

```

%.02f %@", weakSelf.shopCart.userTotal, currency.name)
    weakSelf.totalLabel.text = totalText
}
}

```

What else should we do in `viewDidLoad`? We have set the table view data source to the current object:

```

self.catalogTableView.dataSource = self
} //end viewDidLoad

```

Next, we have to create the table view data source methods. Starting with the number of rows in a section, we just need to return the number of items in the catalog. The final code is as easy as this:

```

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

```

Now, let's get to the other mandatory method: `cellForRowAtIndexPath`. Here, we have to do a few things. Firstly, we have to receive the current cell, the product for this cell, and then assign its description to the cell label:

```

func tableView(tableView: UITableView,
cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell{
    let cell =
tableView.dequeueReusableCellWithIdentifier("productcell",
forIndexPath: indexPath) as! ProductCell
    let product = self.catalog[indexPath.row]
    cell.nameLabel.text = product.description
}

```

Once we these constants, every time the user's price changes, we refresh the **Price** label. Remember that this is a UI operation, and it must be executed on the main thread. Take a look at this code to detect when there is a price change:

```

// Every time the price changes we have to refresh the
price label
product.userPriceSignal
    .observeOn(UIScheduler())
    .startWithNext { [weak cell] (double:Double) -> ()
in

```

```

        let currency =
CurrencyManager.sharedCurrencyManager().currentCurrency
            let priceText = String(format:"For only %.02f %@", product.userPrice, currency.name)
                cell?.priceLabel.text = priceText
}

```

We also need to set the plus (+) button action. To do this, we are going to use the `rac_signalForControlEvents` method as we did in the previous chapters, convert `RACSignal` into `SignalProducer`, add the subscriber with the method `on`, and start the signal with the `start` method. The following code receives `RACSignal` for the `TouchUpInside` event, and then it converts it into `SignalProducer`. Then, it sets the next action, which is adding the product to the shopping cart and updating the `numberOfItems` label, and it finally starts the signal:

```

// add product
cell.addButton
    .rac_signalForControlEvents(.TouchUpInside)
    .toSignalProducer()
    .on(next:{[weak self, weak cell] (_) -> () in
        if let weakSelf = self {
            let numberOfItems =
weakSelf.shopCart.addProduct(product)
            cell?.numberOfItemsLabel.text =
String(format:@"%d", numberOfItems)
            cell?.numberOfItemsLabel.sizeToFit()
        }
    })
.start()

```

Could we have used `startWithNext`? Of course, we could. This is just another way of using a signal producer. You can use the `on` method to set the next action as well as the complete, failure, and other actions. Once your producer is prepared to be used, you can enable it with the `start` method. The remove button works using this code:

```

// remove product
cell.removeButton
    .rac_signalForControlEvents(.TouchUpInside)
    .toSignalProducer()
    .on(next: {[weak self, weak cell] (_) -> () in

```

```
        if let weakSelf = self {
            let numberOfItems =
weakSelf.shopCart.removeProduct(product)
            cell?.numberOfItemsLabel.text =
String(format:@"%d", numberOfItems)
            cell?.numberOfItemsLabel.sizeToFit()
        }
    })
.start()
```

Finally, we can set the cell image if it's available and return the cell:

```
if let imageName = product.imageName {
    cell.productImage.image = UIImage(named:imageName)
}
return cell
} // end cellForRowAtIndexPath
} // end ViewController
```

The first scene is done; now it's time to test it. Have a look at whether you can add and remove products from your basket; make sure that the total and the number of products are automatically updated:

iPhone 5 - iPhone 5 / iOS 9.2 (13C70)

Carrier 9:13 PM

Supermarket

	Bag of Peas For only 0.95 GBP	- 2 +
	Dozen of Eggs For only 2.10 GBP	- 1 +
	Bottle of Milk For only 1.30 GBP	- 0 +
	Can of Beans For only 0.73 GBP	- 0 +

Total: 4.00 GBP [Checkout](#)

The application works but something isn't right. The currency is always the same. This problem will be solved on the next scene.

Creating the checkout scene

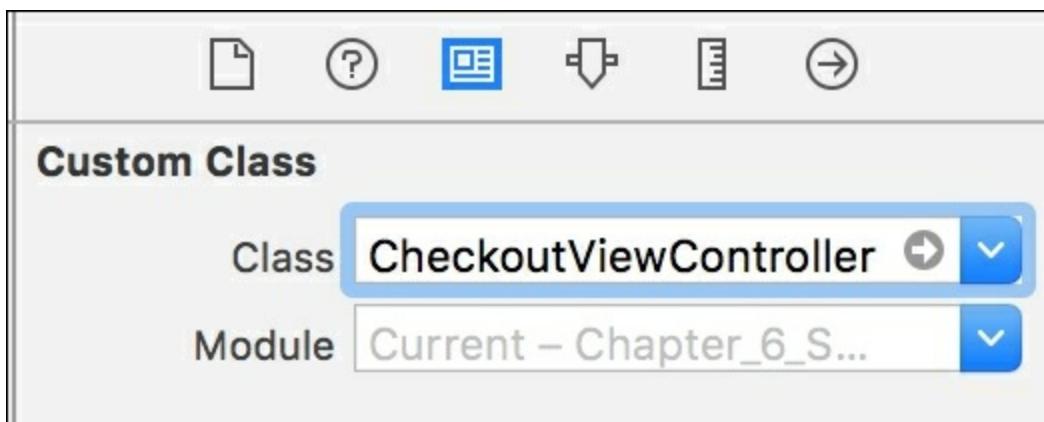
The checkout scene has a layout similar to the first one. The difference is that in the table view, we will show products in the shopping cart, and it will allow the user to change the current currency.

Firstly, add a new **Swift** file to your project, and call it `CheckoutViewController.swift`. Here, we just need to create a class to let the storyboard know which class the second scene belongs to. Go ahead and import `UIKit` and the `ReactiveCocoa` framework. After this, create a class that inherits from `UIViewController` and implements `UITableViewDataSource`, as demonstrated here:

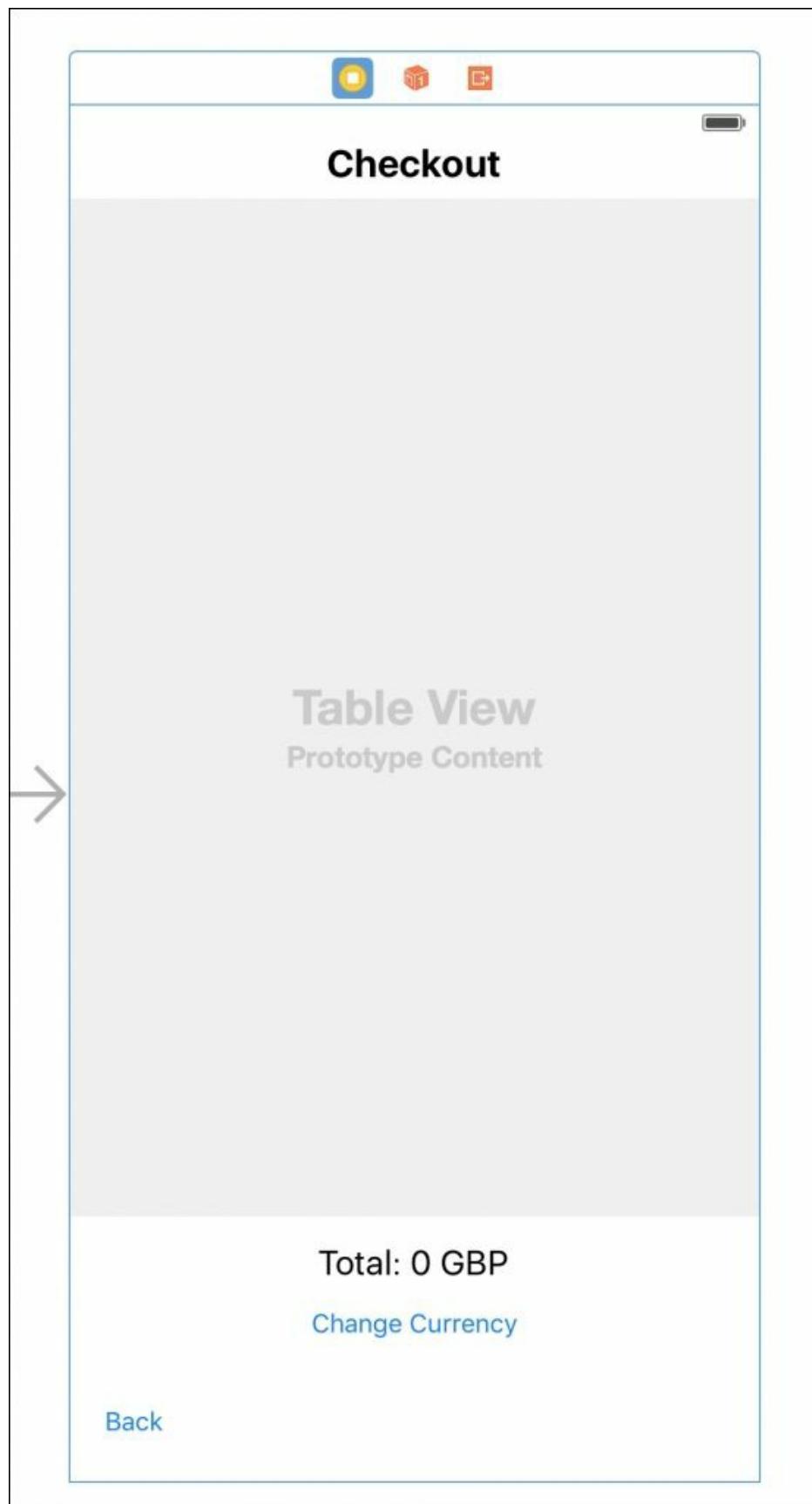
```
import UIKit
import ReactiveCocoa

class CheckoutViewController: UIViewController, UITableViewDataSource { }
```

Return to the storyboard, and add a new View Controller to it. Select the new View Controller, and change its **Class** to `CheckoutViewController` by going to `Identity Inspector` using *command + Option + 3* and updating the **Class** field, as demonstrated here:



You need to *control*-drag from the **Checkout** button of the first scene to the second scene, and select the **Show** option when the menu appears; it will make the second scene appear when the **Checkout** button is pressed. Place two labels in the new scene (one for the scene title and one for the total), one table view, and two buttons (one to change the currency and the other to return to the first scene). The final layout should be similar to what is shown here:



Open **Assistant Editor** with *command + Option + enter*, and connect the UI components to their corresponding properties using this code:

```
@IBOutlet weak var backButton: UIButton!
@IBOutlet weak var changeCurrencyButton: UIButton!
@IBOutlet weak var productsTableView: UITableView!
@IBOutlet weak var totalLabel: UILabel!
```

Return to standard editor with *command + enter*, and set the constraints that you think are necessary. Once you are happy with your constraints, go to the `CheckoutViewController` file.

Besides UI properties, we will create an array with products and their quantities and a shopping cart. For the products and their quantities, we will create a new `private` class; this will make our lives easier when using the table view. The shop cart will be taken from the previous scene.

In `viewDidLoad`, we can start by converting `NSCountedSet` into an array of `PurchasedProducts`; you can do this using the traditional `for` loop, but if you prefer using a more functional way, you can use the `reduce` loop like this:

```
self.purchasedProducts = shopCart.items
    .reduce([PurchasedProduct](), {
        combine: { (var purchasedProducts:
[PurchasedProduct], item:AnyObject) -> [PurchasedProduct] in
            let product = item as! Product
            let quantity =
self.shopCart.items.countForObject(product)
            let purchasedProduct =
PurchasedProduct(product: product, quantity: quantity)
            purchasedProducts.append(purchasedProduct)
            return purchasedProducts
        }
    })
```

Next, we can set `backButton` using `rac_signalForControlEvents`, converting it to a signal producer and subscribing to it using `startWithNext`:

```
self.backButton
```

```

    .rac_signalForControlEvents(.TouchUpInside)
    .toSignalProducer()
    .startWithNext { [weak self](_) -> () in
        self?.dismissViewControllerAnimated(true,
completion: nil)
    }
}

```

Our `changeCurrencyButton` is a little bit more complicated as it needs to continue the signal with the help of another signal. With `RACSignal`, we used a method called `flattenMap`, but the signal producers use one called `flatMap`, which is very similar, except that it receives a new argument (the first one) called a **strategy**. In this case, we will use `concat`; this means that the producers should be concatenated so that their values are sent in the order of the producers themselves. Alternatively, you can use `Merge` or `Latest`.

Now, the `on` method will handle two events: the `failed` method, which for any reason shows that we couldn't retrieve information from the server, and the traditional subscriber. After the `on` method is complete, we can start the signal. The final code looks like this:

```

self.changeCurrencyButton
    .rac_signalForControlEvents(.TouchUpInside)
    .toSignalProducer()
    .flatMap(FlattenStrategy.Concat) { (_) ->
Signal<[Currency], NSError> in
    return
CurrencyManager.sharedCurrencyManager().signalForRates()
}
    .observeOn(UIScheduler())
    .on(failed: { [weak self] (_:NSError) -> () in
        self?.showNoInternet()
    }) { [weak self] (input:[Currency]) -> () in
        self?.showRates(input)
}
.start()
}

```

From the shopping cart, we have to observe the user's price; don't forget that after changing the currency, the total of the shopping cart will change as well, and the user should be updated about it:

```
self.shopCart
```

```

        .signalForUserTotal()
        .observeOn(UIScheduler())
        .startWithNext { [weak self] (input:Double) -> ()
in
            if let weakSelf = self {
                let currency =
CurrencyManager.sharedCurrencyManager().currentCurrency
                    let totalText = String(format:"Total:
%.02f %@", weakSelf.shopCart.userTotal, currency.name)
                    weakSelf.totalLabel.text = totalText;
            }
        }
    }
}

```

The last part of `viewDidLoad` is simply the assigning of the table view data source:

```

        self.productsTableView.dataSource = self
    } // end viewDidLoad
}

```

For `numberOfRowsInSection`, we can just implement this easy method:

```

// MARK: - Table view data source
func tableView(tableView: UITableView,
numberOfRowsInSection section: Int) -> Int{
    return self.purchasedProducts.count
}

```

For `cellForRowAtIndexPath`, it is a bit more tricky. The cell needs to be updated when the user changes the currency. Here, we will use the `on` method without tags; it is the same as using the `next: tag`:

```

func tableView(tableView: UITableView,
cellForRowAtIndexPath indexPath: NSIndexPath) ->
UITableViewCell{
    var cell =
tableView.dequeueReusableCellWithIdentifier("cell")
    if cell == nil {
        cell = UITableViewCell(style: .Subtitle,
reuseIdentifier: "cell")
    }
    let purchasedProduct =
self.purchasedProducts[indexPath.row]

    // observing any price changing
}

```

```

        purchasedProduct.product.userPriceSignal
            .observeOn(UIScheduler())
            .on { [weak cell] (input:Double) -> () in
                let currency =
                    CurrencyManager.sharedCurrencyManager().currentCurrency
                    let total = purchasedProduct.product.userPrice *
                    Double(purchasedProduct.quantity)
                    cell?.textLabel?.text = String(format: "%d %@ for
                    %.02f %@", purchasedProduct.quantity,
                    purchasedProduct.product.description, total, currency.name)
            }.start()

        return cell!
    }
}

```

We still need to implement two methods. The first one, called `showRates`, shows the user different currency options. Whenever the user chooses a currency, it must be assigned to the Currency Manager:

```

// MARK: - Private
private func showRates(rates:[Currency]) {
    let alertController = UIAlertController(title: "Change
    Currency", message: "Please choose your currency",
    preferredStyle: .ActionSheet)

    for currency in rates {
        let action = UIAlertAction(title: currency.name,
        style: .Default, handler: { (_) -> Void in

    CurrencyManager.sharedCurrencyManager().currentCurrency =
    currency
        })
        alertController.addAction(action)
    }

    let action = UIAlertAction(title: "Cancel", style:
    .Cancel, handler: nil)
    alertController.addAction(action)
    self.presentViewController(alertController, animated:
    true, completion: nil)
}

```

We are once again using ReactiveCocoa implicitly. When you set the currency, it changes the products and the shopping cart, which changes the label that

displays the total.

Finally, you can implement the method that displays a message telling you that an error has occurred:

```
private func showNoInternet() {
    let alertController = UIAlertController(title: "Error", message: "Unable to retrieve currencies", preferredStyle: .Alert)
    let action = UIAlertAction(title: "Dismiss", style: .Cancel, handler: nil)
    alertController.addAction(action)
    self.presentViewController(alertController, animated: true, completion: nil)
}

} // end Checkout View Controller
```

Is the application complete now? The answer is *yes*. We still need to send the shopping cart from one scene to another. Return to `ViewController.swift`, go to the `viewDidLoad` method, and add this code before the data source assignation:

```
self.rac_signalForSelector(Selector("prepareForSegue:sender:"))
)
.toSignalProducer()
.map({ (input:AnyObject?) -> RACTuple in
    return input as! RACTuple
})
.map({ (arguments:RACTuple) -> UIStoryboardSegue
in
    return arguments.first as! UIStoryboardSegue
})
.filter({ (segue:UIStoryboardSegue) -> Bool in
    return segue.destinationViewController is
CheckoutViewController
})
.startWithNext { (segue) -> () in
    let checkoutViewController =
segue.destinationViewController as! CheckoutViewController
    checkoutViewController.shopCart =
self.shopCart
}
```

```
}
```

This code checks when the `prepareForSegue` method is called, which means that the next scene will appear, and whenever it does, we take the first argument. The first argument is `UIStoryboardSegue`, and if its destination View Controller is of the `CheckoutViewController` type, we can set its `shopCart` property before loading such a View Controller. Actually, this application doesn't need to check the View Controller class as there is no other possibility of a new scene; however, we must develop applications that are prepared for future changes.

What have we done with this code? The `rac_signalForSelector` method observes when the `prepareForSegue` selector is called, and then we convert `RACSignal` to `SignalProducer`. The input is `RACTuple` with the method arguments in it; this refers to the storyboard segue and sender. As we don't need the sender, we just map the segue. Knowing that there is no other possible View Controller apart from `CheckoutViewController`, it is a good idea to filter it to avoid future problems. Finally, the subscriber can transfer the shopping cart from the current View Controller to the next one.

Testing the application

The application is complete; however, we can't say that our work is done until we test the application and ensure that everything is working as expected. Run the application, add some items to the basket, and press **Checkout**; you should see a view similar to what is shown in this screenshot:

iPhone 5 - iPhone 5 / iOS 9.2 (13C70)

Carrier 

12:03 AM



Checkout

2 Dozen of Eggs for 4.20 GBP

2 Bag of Peas for 1.90 GBP

Total: 6.10 GBP

[Change Currency](#)

Back

Now, try to change the currency. If you are testing using iOS 9, you will receive our error message; this is a good sign. It means that the fail subscriber has worked as expected:



iPhone 5 - iPhone 5 / iOS 9.2 (13C70)

Carrier



12:07 AM



Checkout

2 Dozen of Eggs for 4.20 GBP

2 Bag of Peas for 1.90 GBP

Error

Unable to retrieve currencies

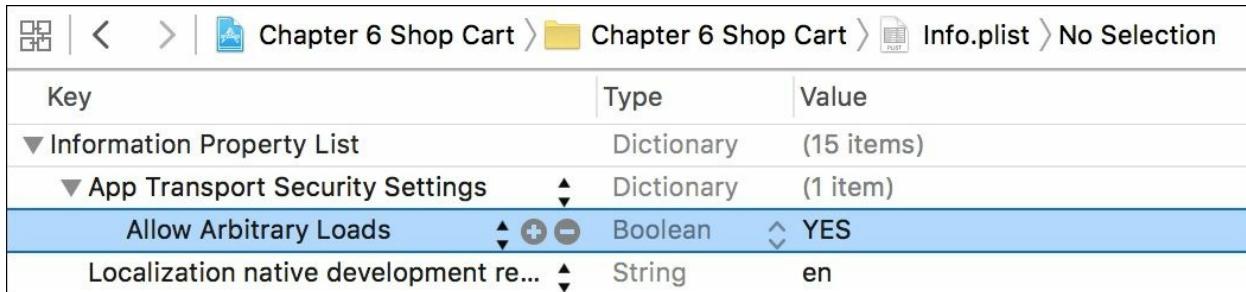
[Dismiss](#)

Total: 6.10 GBP

[Change Currency](#)

[Back](#)

Let's fix this error by going to the `Info.plist` file, which is located in the **Project Navigator**. Add a new record with the **App Transport Security Settings** key and a subrecord with the **Allow Arbitrary Loads** key. Set the subrecord value to **YES**, as demonstrated here:



The screenshot shows the Xcode Project Navigator with the path: Chapter 6 Shop Cart > Chapter 6 Shop Cart > Info.plist. The table below displays the contents of the Info.plist file.

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
▼ App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES
Localization native development re...	String	en

Rerun your application, add some items to the basket, press **Checkout** and then tap **Change Currency**. Now, a list of currencies is displayed on the screen. Choose one, and make sure that every price on the screen changes according to the user's currency:

iPhone 5 - iPhone 5 / iOS 9.2 (13C70)

Carrier 

12:18 AM



Checkout

2 Bag of Peas for 3.97 AUD

2 Dozen of Eggs for 8.78 AUD

Total: 12.75 AUD

[Change Currency](#)

Back

Tap the **Back** button, and ensure that the whole catalog has been updated as well as the first **Total** label:

iPhone 5 - iPhone 5 / iOS 9.2 (13C70)

Carrier 12:20 AM

Supermarket

	Bag of Peas For only 1.99 AUD	- 2 +
	Dozen of Eggs For only 4.39 AUD	- 2 +
	Bottle of Milk For only 2.72 AUD	- 0 +
	Can of Beans For only 1.53 AUD	- 0 +

Total: 12.75 AUD [Checkout](#)

Congratulations, your shopping cart is complete!

Summary

In this chapter, you learned a new way of development using ReactiveCocoa. We used `signal` and `SignalProducer` rather than `RACSignals`. These structs work with different methods from `RACSignal`, and they have a few advantages, for example, the usage of generics make it easier to know the input type.

Now, we can use `UISchedule` for executing the subscription on the main thread and also create our own type of schedule if necessary.

We also took a look at how `signal` and `SignalProducer` are very similar; with `SignalProducer`, we only define a signal until it starts.

In the next chapter, you will learn how to debug and profile with ReactiveCocoa, something that is not as trivial as it's made out to be.

Chapter 7. Testing Your Application

Imagine one day you deliver everything on time, no issues are found, there are no memory leaks, no client complaints, no specification changes, and suddenly...you wake up. Let's face the reality: it doesn't matter what framework, philosophy, or methodology you use, there is always something to fix, and it is necessary to test the application and debug it.

Reactive programming is considered a good way of programming; however, some people say that it is harder to debug. This is not exactly true; in this chapter, we are going to see that you can debug it like any other application.

This chapter assumes that you are using ReactiveCocoa 4.0, which requires Swift 2. If you are using a previous version of ReactiveCocoa or a previous version of Swift, don't worry, the concepts are the same; you just need to adapt the current code to according to your version.

In this chapter we will cover:

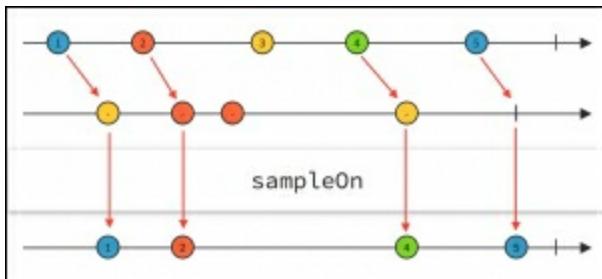
- Checking the expected results
- Unit tests
- UI tests
- Profiling with Instruments

Checking the expected results

When developing with ReactiveCocoa, you will probably need to use some functions that combine multiple signals, such as `combineLatest` or `zip`. You can investigate the results of these functions using a page called RAC Marbles, whose URL is <http://neilpa.me/rac-marbles/>. So how does this page work?

It is very simple: first, you have to choose the function whose operation you want to check. After this, you will see a diagram with arrows that represent the timeline and some circles that represent the time when the signal is received. Underneath the function's name, you can see another timeline with other circles that represent the results.

Let's see an example of how it works: imagine that you want to know the behavior of a function called `sampleOn`. Click on this function name on the left-hand side, and a diagram will appear. Have a look at the numbers contained inside the input circles and their respective values in the output:



As you can see, the `sampleOn` method works by propagating the signal only when the second signal also sends a *next* call; in this case, you can observe that signals 1, 2, and 4 have a next signal that comes before the main signal propagates a new *next* call.

The vertical line that crosses each timeline arrow at the end represents `sendComplete`. As you can see, just after the first signal sends the value 5, the second signal triggers `sendComplete`, and it also propagates the value.

Value number **3** is dropped as number **4** arrives before the second signal sends a next call. The second signal also sends a next call, which is not propagated; have a look at the third circle: it doesn't re-propagate signal **2** since it was already consumed.

What happens if the first signal completion happens before the second completion? Will signal **5** be propagated or not? What happens if the second signal starts first? Whenever you have doubts about this, you just need to move the signals and the complete signal, and you will have the result immediately. As you can see in the next diagram, it doesn't matter if the first signal sends its completion before the second one; the signal is still propagated, and the second signal starting before the first one isn't a problem either. The behavior is still the same:



Great, now you have a good friend who can help you with the signals' behavior by displaying the expected result! Unfortunately, not every signal method or function is available on this page; there are a few of them that are on the to-do list.

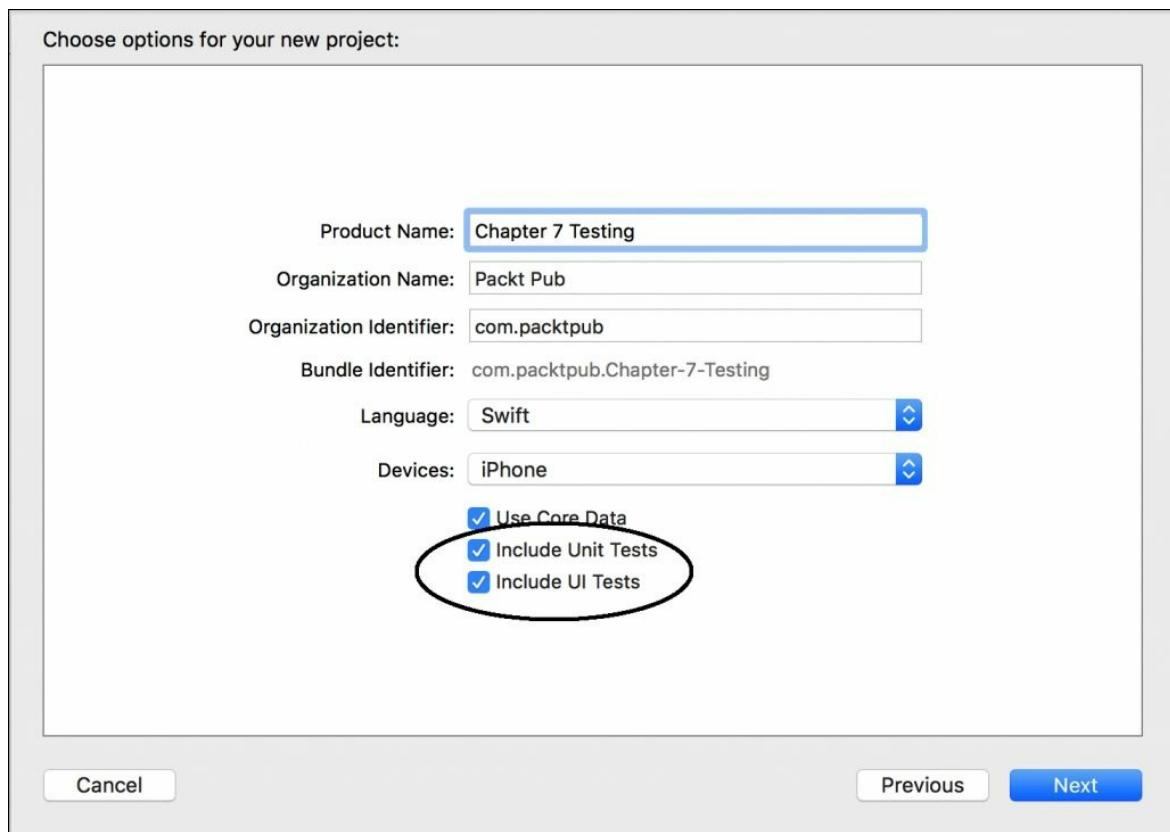
Creating unit tests

Performing unit tests is the traditional way of testing your application and trying to detect failures when a commit is pushed to the server. This way, you can reduce the number of bugs in your application and also ensure that your software is working as expected. Basically, it consists of creating functions that call your object methods and check whether the final result is the one expected.

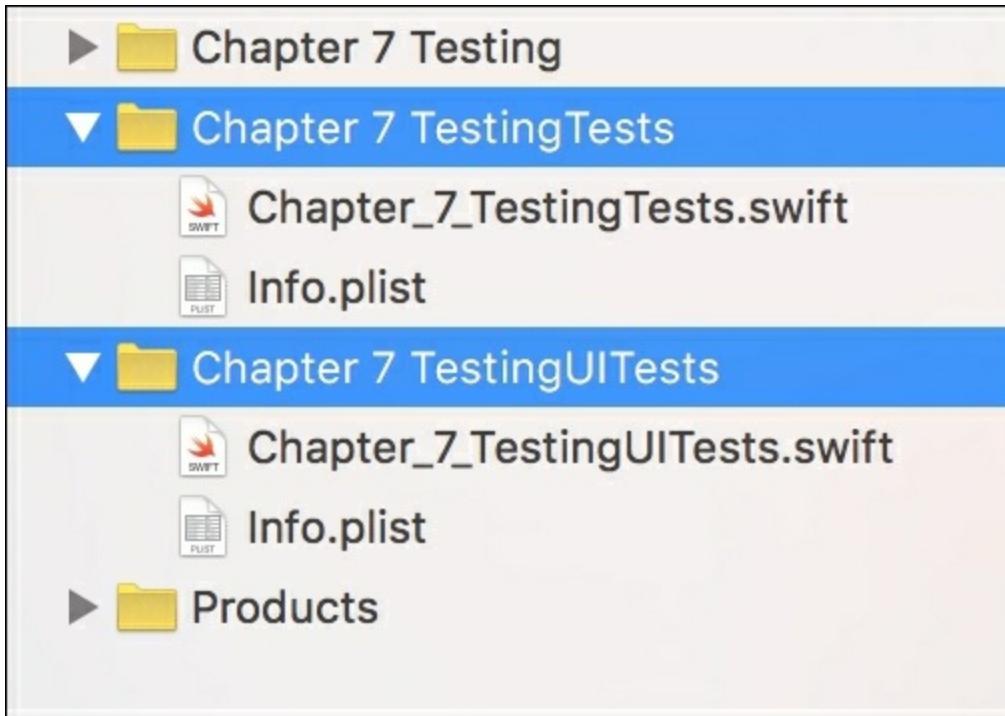
There also are methodologies based on unit tests, such as test-driven development and behavior-driven development. These methodologies assume that for every development cycle, you have to create the unit tests first.

Xcode comes with a built-in framework called **XCTest**, which is the one we are going to use in this section. If you prefer using a different framework, feel free to do so; the concepts should be similar. The steps for creating unit tests are as follows:

1. Open **Xcode** and create a new project called `Chapter 7 Testing`. Ensure that **Swift** is the main language and the checkboxes for **Include Unit Tests** and **Include UI Tests** are checked, as demonstrated in this screenshot:



2. Install ReactiveCocoa using your favorite method; after this, you can verify that your project has two groups for testing, one called Chapter 7 Testing Tests and the other called Chapter 7 Testing UITests:



3. In this section, we are going to work with non-UI tests; therefore, don't expect any graphical result. Click on the file `Chapter_7_TestingTests.swift`; here, you will see a class called `Chapter_7_TestingTests` with a few methods to be implemented.
4. The method `setUp` is called before each test is executed; it is used for initializing objects that are commonly used in the `test` functions for this test case. If you are testing a signal, you can create it in the `setUp` function.
5. There is another method called `tearDown`, which is called after each test execution. Here, you can release some resources or, for example, send a completion message ensuring that a signal is not on halfway.
6. The test function itself just comprises methods that receive no arguments and return `void`. These methods must start their names with the word `test`, such as `testExample` and `testPerformanceExample`, which are functions created by default.

If a test function reaches its end without any interruption from `XCTAssert` functions, it is considered a successful test. What are `XCTAssert` functions? An `XCTAssert` function is a function that receives a Boolean value as argument; if it is `false`, it interrupts the test and marks it as a failed test. There are other functions that receive other arguments rather than receiving a Boolean to check whether the current value or values are the expected ones, such as `XCAssertEqual`, which expects that the two values passed as arguments are valid if they are equal.

Use the following steps for this example:

1. First, import ReactiveCocoa at the beginning of the current file:

```
import ReactiveCocoa
```

2. Add a new method called `testMutableValue`; here, we are going to confirm the status of a value and ensure that the final value is something based on the values of a mutable property that propagated its value. Start implementing this function by creating two mutable properties with the following code:

```
func testMutableValue() {
    let firstValue = MutableProperty<Int>(10)
    let secondValue = MutableProperty<Int>(0)
```

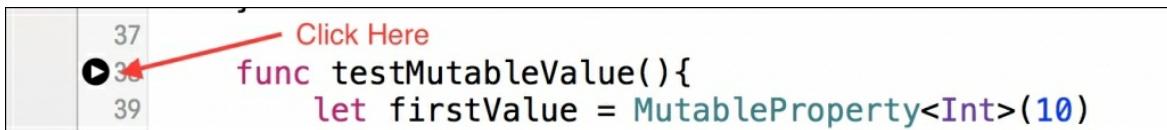
3. Then, create a signal for the first value, which adds its new value to the second value every time we have a change in the first value:

```
firstValue.producer.startWithNext { (input:Int) ->
() in
    secondValue.value += input
}
```

4. After this, we have to remember that `startWithNext` adds a subscriber to the signal and also triggers it; this means that `secondValue` will be increased by the current value, which is `10`; therefore, if the `secondValue` is `10`, we can consider the test successful until now:

```
XCTAssertEqual(secondValue.value, 10)
} // end testMutableValue
```

5. You can run the current test by clicking on the play icon located on the left-hand side of the method header, like the following screenshot:



6. You can also execute the test case using the key combination *command* + *U*; however, it will execute every test function, which can take too long if you have many of them.
7. After executing the test function, you will see that the play icon transforms into a green icon with a tick, which means that the test has finished successfully; however, if you get a red icon with a cross inside, it means that the test has failed, and the failed assertion will be highlighted.
8. If you think that your test is still incomplete, you can check what else you can do in your test to complete it. Remember that continuing with the test function makes sense only if you can add more assertions. In this case, we can check whether if we continue changing the value of `firstValue` it still changes `secondValue`. Add the following code to the test function in order to ensure that the signal still continues changing the second variable:

```
XCTAssertEqual(secondValue.value, 10)
firstValue.value *= 5
firstValue.value = secondValue.value / 10
XCTAssertEqual(secondValue.value, 66)
} // end testMutableValue
```

9. Run your test and check its result; once you are happy with it, you can move on to the next test function.

Note

Ideally, test cases must be run on a continuous integration server every time you push your code to the version control system. There are many continuous integration servers on the market, such as Jenkins, Team City, and Go Server.

Configuring these servers is out of the scope of this book, but it is worth having a look at them.

Don't think that running the test cases only once is enough; every time you add a new code into your application or library, you should rerun your unit tests.

Tip

If you want to add your unit test on a continuous integration server, have a look at the command line `xcodebuild test`.

Using signals for checking the results

Let's imagine that we are developing a class that extracts numbers from a file. Every time a number is found, a next call is done. When we reach the file's end, a completion call is sent. How can we develop and test this class? We can do this by performing the following steps:

1. Firstly, we have to add a new file to our project (not in the test units) and name it `FileNumbersReader.swift`. Start by importing `ReactiveCocoa` with the following line:

```
import ReactiveCocoa
```

2. Now, open this class and create a method called `signalProducerForFile`. This method returns a signal producer, which will send the numbers found on the file whose name is passed as the argument. Once we have understood the idea of this class, we can create its skeleton with the following code:

```
class FileNumbersReader {  
    func signalProducerForFile(filePath:String) ->  
        SignalProducer<Int, NSError> {  
            } // end signalProducerForFile  
        } // end  
FileNumbersReader
```

3. We can now create the signal producer with an initializer that receives a start handler as an argument; this way, we can specify when the events are sent. Place the following code inside the `signalProducerForFile` method:

```
return SignalProducer<Int, NSError> { (observer:  
    Observer<Int, NSError>, disposable:  
    CompositeDisposable) ->  
        () in  
            } // end SignalProducer
```

4. The first part of the implementation of this signal is specifying what should be done when this method finishes. It doesn't matter whether we return from this handler with an error or no errors; both cases must send a completion event. Swift allows us to specify this once with a statement called `defer`. Place the following code at the beginning of the signal

producer initializer handler:

```
    defer {
        observer. d()
    }
```

5. Using `NSFileManager`, we can retrieve the full contents of the argument file. If the file doesn't exist or can't be opened for whatever reason, we have to send an error event; otherwise, we can continue with the handler implementation. For cases like this, Swift gives us the `guard` statement, which checks for a current status, and if it is not valid, it exits from the current function:

```
let fileManager = NSFileManager.defaultManager()
guard let content =
fileManager.contentsAtPath(filePath)
else {
    let error = NSError(domain: "FileNumbersReader",
code: 100,
userInfo: [NSLocalizedStringKey: "Can't open
file"])
    observer.sendFailed(error) return }
```

6. As the file content is returned as `NSData`, we have to convert it to `String`. If the conversion is not possible for whatever reason, another error must be sent; thus, we can continue with our code with another `guard` statement:

```
guard let stringContent = String(data: content,
encoding:
    NSUTF8StringEncoding) else {
    let error = NSError(domain:
"FileNumbersReader", code: 101, userInfo:
[NSLocalizedStringKey: "File content is not
a text"])
    observer.sendFailed(error)
    return
}
```

7. Finally, we can split the string into an array of strings and convert each of these tokens into an integer. If the conversion is possible, we send a next event; if it is not, we just ignore it. Place this code at the end of the signal producer initializer handler:

```

let tokens = stringContent.characters.split { $0 ==
"
    "}.map( String.init )
        tokens.forEach { (value:String) -> ()
in
    if let number = Int(value) {
        observer.sendNext(number)
    }
}

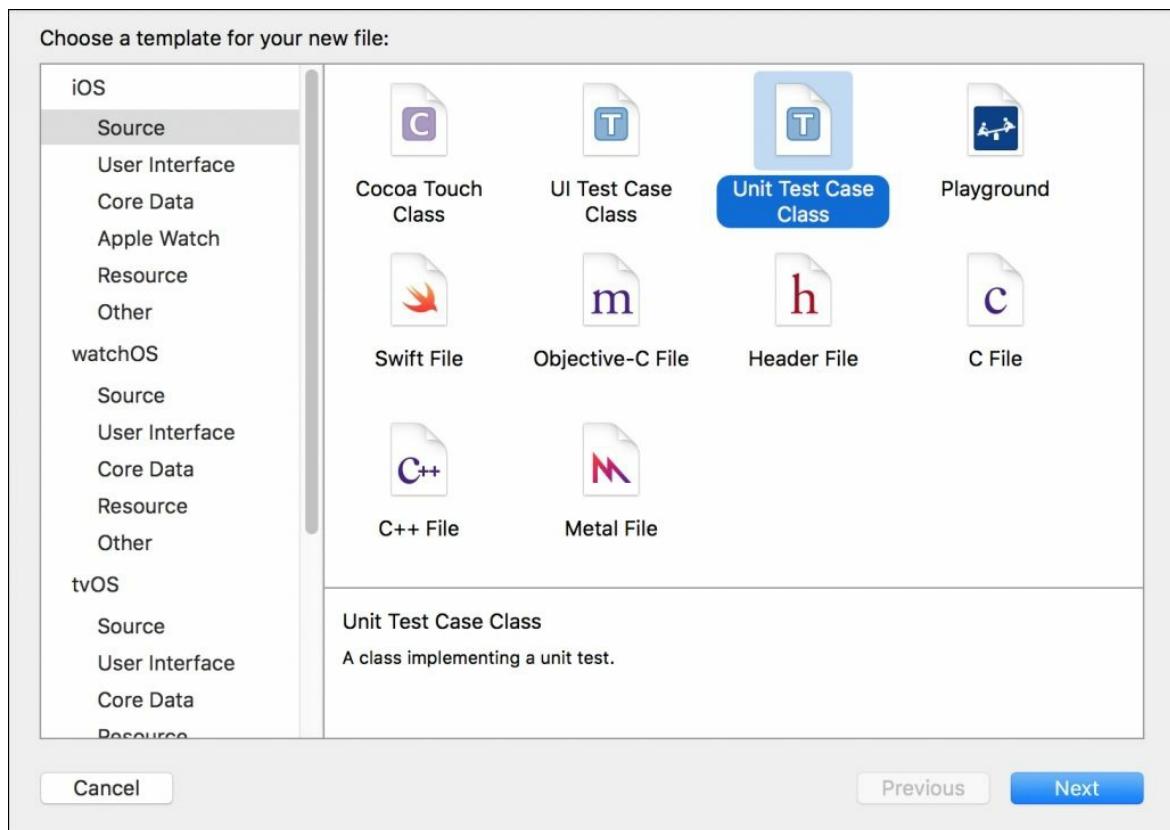
```

If you've never used this method for splitting a string, here's an explanation: we basically convert a string into another class called `String.CharacterView`, which is a collection of characters. The `split` method returns the character sequence until the `handler` method returns `true`; in this case, this is when it finds a whitespace. These sequences can be recast again to string using the `map` method with the string initializer as an argument. This whole process doesn't change the initial string; we are just using functional programming again. Our method is done; now we can think about the tests that we have to do to make sure that this code works as expected. Basically, we can have three tests:

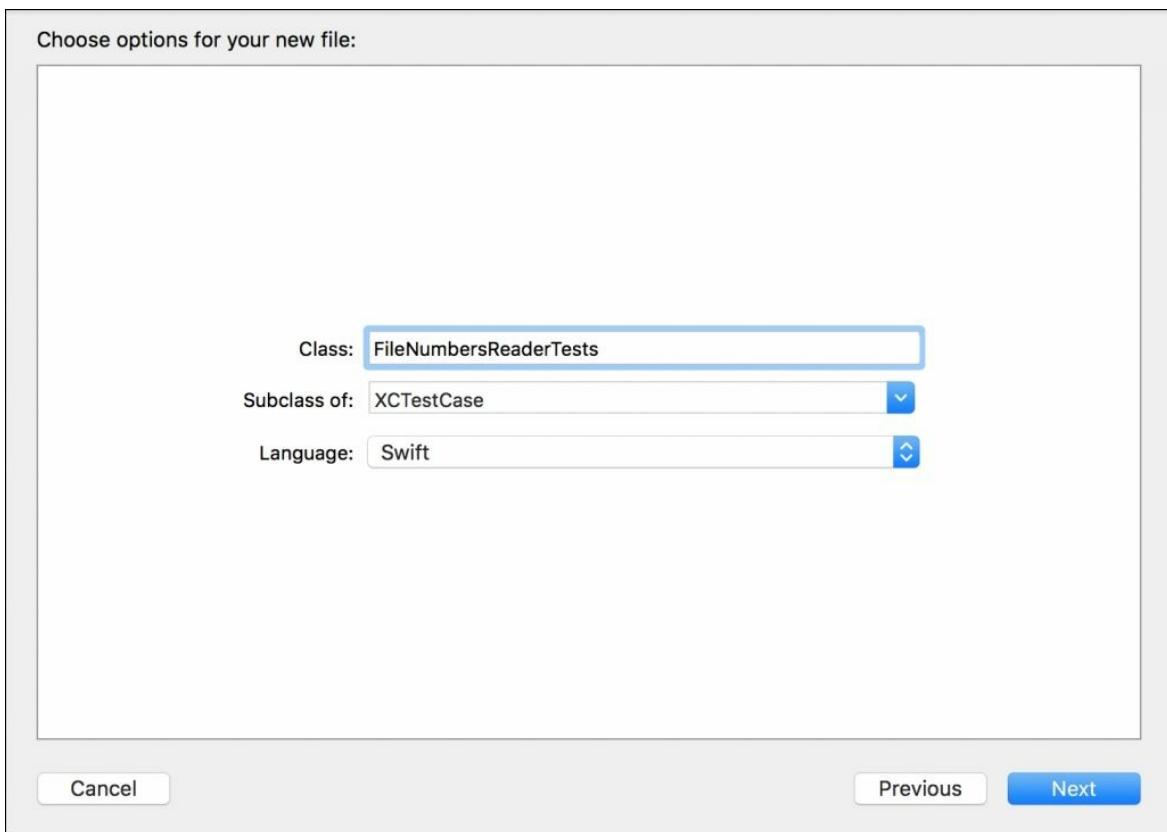
- Test whether the signal triggers an error when a file doesn't exist
- Test whether the signal triggers an error when the file content is not convertible to `String`
- Test whether the subscription is triggered with the correct values

Once you have this in mind, we can proceed with the test development using the following steps:

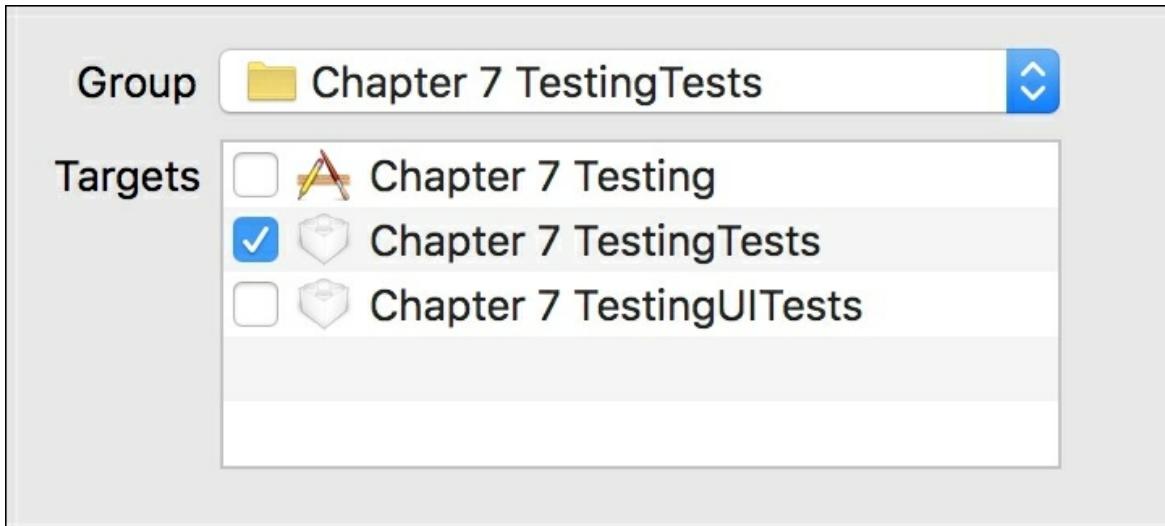
1. Add a new file to your target, `Chapter 7 TestingTests`. This time, instead of choosing a **Swift file** or a **Cocoa Touch class**, select **Cocoa Unit Test Case Class**, as demonstrated in the next screenshot:



2. In the next dialog, set the class name to `FileNumbersReaderTests`, make it a subclass of `XCTestCase`, and use the **Swift** programming language, like in the following sample screenshot:



3. The last dialog asks you where to save your file, the group it belongs to, and its targets. Save the file with the rest of the Swift files that belong to this project; the group should be `Chapter 7 TestingTest` and the target has to be `Chapter 7 TestingTest` only. Before pressing the **Create** button, ensure that your settings are like the next screenshot shows:



4. A new test case class is created with two `test` functions. Remove both functions as we are not going to use them, and import the module `Chapter_7_Testing` at the beginning of the file; this will allow us to access the application's classes and use them in our test case class:

```
@testable import Chapter_7_Testing
```

5. Now, create a new test in your test case class and call `it testFileNotFound`. Here, we have to obtain the signal producer for a nonexisting file and ensure that the next subscription is not called and the fail subscription is triggered with an error with the code `100`. The first approach can be using the following code:

```
func testFileNotFound() {
    let fileNumbersReader =
    FileNumbersReader()
    let signalProducer =
    fileNumbersReader.signalProducerForFile
    ("nonexistingfile.txt")
    signalProducer.on(failed: { (error:NSError) -> () in
        XCTAssertEqual(100, error.code,
        "Wrong
        error code")
    }) { (_:Int) -> () in
```

```

        XCTAssert(false, "Wrong way")
    }.start()
}

```

Is anything missing in this test? Actually, this test is not 100% reliable. Imagine that the expected failure is not triggered: this test won't fail. This doesn't mean that it is wrong; it just means that it can be improved.

How can we improve this test? The first idea could be to add a `print` function with some message, and then we can check whether the message is printed or not. This idea might be interesting when creating the test as a developer; however, it is not very useful if you are running this test with continuous integration as it won't detect whether the message is printed or not.

Is there any other solution? As this test is a synchronous test, we can add a Boolean variable that starts with false and switches to true inside the failure handler. At the end of this test, we have to perform an assertion that checks whether such variables end with the right value. Complete this test by adding the following highlighted code:

```

func testFileNotFound() {
    var errorFound = false

    let fileNumbersReader = FileNumbersReader()
    let signalProducer =
fileNumbersReader.signalProducerForFile("nonexistingfile.txt")

    signalProducer.on(failed: { (error:NSError) -> () in
        XCTAssertEqual(100, error.code, "Wrong error
code")
        errorFound = true

    }) { (_:Int) -> () in
        XCTAssert(false, "Wrong way")
    }.start()
    XCTAssertTrue(errorFound, "No error was triggered")
}

}

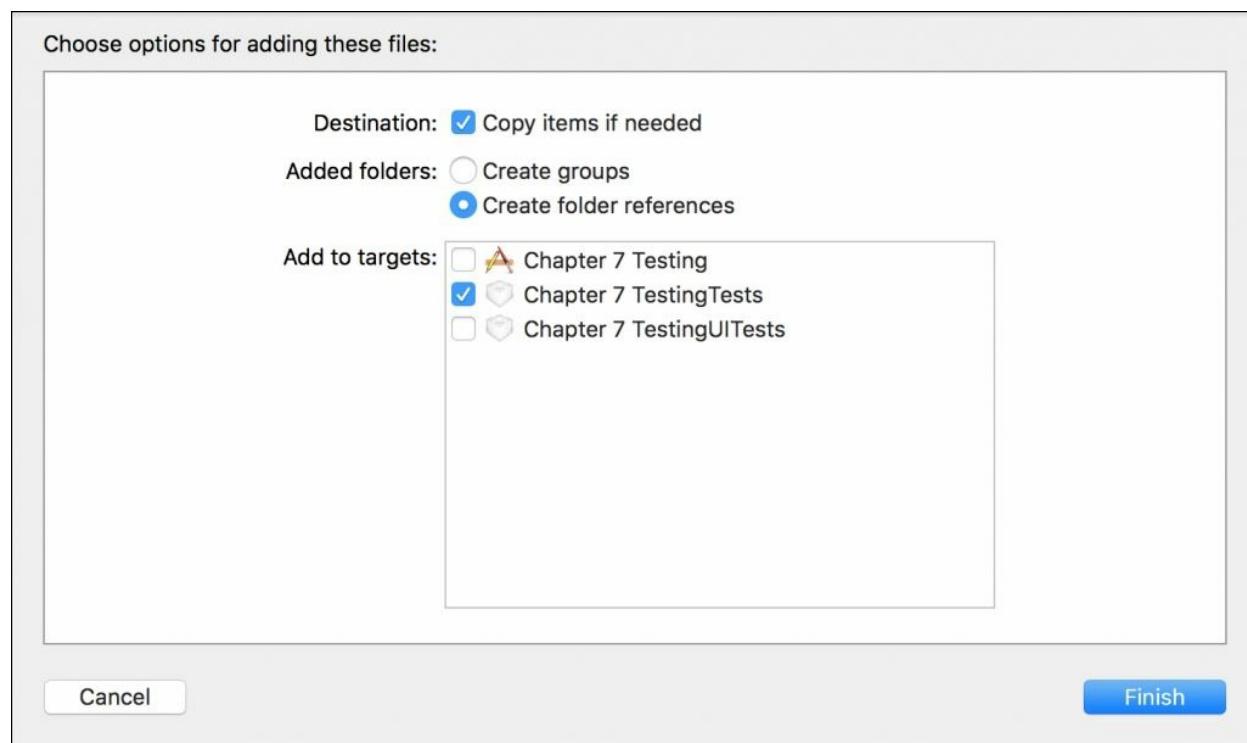
```

Now this test is more reliable. Run it and check that it gets a successful result.

This test is done; it is time to start the second test. Check if we get an error whenever the file can't be converted into a string.

The idea is that this test is similar to the previous one; however, we need to have a file made only for this test. Any file that can't be converted to UTF8 should be sufficient, such as a picture.

Drag the file called `pato.png` located in this book's resources to your tests; if you haven't downloaded this file, you can download any image from the Internet, as we are just trying to use a file that fails to be converted to a String. When copying the file, ensure that its target is `Chapter 7 TestingTests`, not the application target. Don't add files to your application that are only required for the unit test; it will make your application grow unnecessarily. The next screenshot shows an example of how this file should be set when you copy it into your project:



Once the file has been copied, return to `FileNumbersReaderTests.swift`.

Create a new test function called `testBinaryFile`. This test is very similar to the previous one except for two details: the expected error has the code `101` rather than `100` and the file can't just be set with its name. As we have copied this file on the test target, we have to use its location on the bundle specific for this target. To do this, we can instantiate an `NSBundle` object and ask for its `resourcePath` property. Add the following code to your test case, and pay attention to the highlighted code, which marks the lines that are different from the previous test:

```
func testBinaryFile() {  
  
    var errorFound = false  
    let fileNumbersReader = FileNumbersReader()  
  
    let path = NSBundle(forClass:  
self.classForCoder).resourcePath!  
    let fullPath = "\(path)/pato.png"  
  
    let signalProducer =  
fileNumbersReader.signalProducerForFile(fullPath)  
    signalProducer.on(failed: { (error: NSError) -> () in  
        XCTAssertEqual(101, error.code, "Wrong error  
code")  
  
        errorFound = true  
    }) { (value:Int) -> () in  
        XCTAssert(false, "Wrong way")  
    }.start()  
    XCTAssertTrue(errorFound, "No error was triggered")  
}
```

The second test is done; run it and check whether it is working and finishing with a success status.

The third test is a bit more complex as we need to test each value, and it requires another file, which is considered a valid file. Let's start with the file: you just need to drag the file `numbers.txt` into your test's target folder.

Return to the `FileNumbersReaderTests.swift` file and import the `ReactiveCocoa` framework by adding the following code to the beginning of

the file:

```
import ReactiveCocoa
```

Here, we will add a new test case called `testValues`. How can we do this? It's very simple: we just need to sync up the signal that we are testing with another signal that sends the values in the expected order.

How can we do this? Start by opening the function and creating a new signal producer that just sends the expected values in their corresponding order. This signal is created based on an array iteration, with the following code:

```
func testValues() {
    let signalValues = SignalProducer<Int, NSError>{
(observer: Observer<Int, NSError>, disposable:
CompositeDisposable) -> () in
    [10, 20, 30, 40].forEach({ (value:Int) -> () in
        observer.sendNext(value)
    })
    observer.sendCompleted()
}
```

Then, create the signal producer for the file `numbers.txt`, similar to the way we did in the previous test:

```
let fileNumbersReader = FileNumbersReader()
let path = NSBundle(forClass:
self.classForCoder).resourcePath!
let fullPath = "\(path)/numbers.txt"
let signalProducer =
fileNumbersReader.signalProducerFromFile(fullPath)
```

Now we can set the fail subscription with a handler that marks the test as failed, because we are not expecting any failure at this point:

```
signalProducer.on(failed: { (error: NSError) -> () in
    XCTAssert(false, "Unexpected error: \
(error.localizedDescription)")
})
```

Then, we have to sync this signal with the first one, meaning that for each event of one signal, we have to merge it with one event of the other signal and

see whether they have the same value. For cases like this, we have a method called `zipWith`, which pairs the results of two signals:

```
.zipWith(signalValues)
```

Finally, we can start the signal with a subscription. As the signal was zipped before, the input now is two arguments, one of each signal; therefore, we just need to check whether they are the same or not with an assertion, with this final code:

```
.startWithNext { (value:Int, expectedValue:Int) ->
() in
    XCTAssertEqual(value, expectedValue, "Value \
(value) is different from it's expectation \
(expectedValue)")
}
} // end testValues
```

Run this test and check whether it returns a successful result.

Testing an asynchronous signal

The previous tests were synchronous, which means that when starting the signal, we expect to receive every call to the signal subscriber and the test completion; only after this does the test function continues executing. As we know, reactive programming is very commonly used when we have asynchronous calls, which means that when we start a signal, it doesn't block the code following it.

A good example of an asynchronous signal is an HTTP request. Here, we are going to create another class in our application called `JokerRequester`. As its name says, it is going to request a joke with an API and return its JSON object.

Add to your application a new file called `JokerRequester.swift`. Here, we are going to create a class with only one method called `signalProducerForJoke`. This method will perform an HTTP request, similar to the one we did in the previous chapter, but using a different URL. Place this code in your file to create the URL request:

```
import ReactiveCocoa

class JokeRequester{
    func signalProducerForJoke(id:Int) ->
SignalProducer<[String:AnyObject], NSError> {
    return SignalProducer<[String:AnyObject], NSError>{
(observer:Observer<[String : AnyObject], NSError>,
composite:CompositeDisposable) -> () in
        let url = NSURL(string:
"http://api.icndb.com/jokes/\(id)")!
        let session = NSURLSession.sharedSession()
        session.dataTaskWithURL(url, completionHandler: {
(data: NSData?, response: NSURLResponse?, error: NSError?) ->
Void in
            defer { observer.sendCompleted() }
            guard error == nil else {
                observer.sendFailed(error!)
                return
            }
            do {
                let json = try
```

```

        NSJSONSerialization.JSONObjectWithData(data!, options:
NSJSONReadingOptions(rawValue: 0)) as! [String:AnyObject]
            observer.sendNext(json)
        }catch let error {
            observer.sendFailed(error as NSError)
        }
    }).resume()
}
}
}

```

This code basically creates `SignalProducer`, which calls a website, and whenever we get the response, we convert it to a JSON message and send it with `sendNext`. If any errors happen halfway, we propagate the error with `sendError`.

Build your application with *command + B* to check that you have no syntax errors. Then, return to your unit test group and add a new unit test called `JokeRequesterTest`. Import the application module with the following code:

```
@testable import Chapter_7_Testing
```

Now, you can add a new test to your test case called `testJoke40`, which will request the joke with the ID number 40, expecting the following JSON message:

```
{
    "type": "success",
    "value": {
        "id": 40,
        "joke": "A handicapped parking sign does not signify that this spot is for handicapped people. It is actually in fact a warning, that the spot belongs to Chuck Norris and that you will be handicapped if you park there.",
        "categories": []
    }
}
```

Basically, all you have to do is to instantiate a new `JokerRequester` object, get the signal producer for joke number 40, and perform some assertions on its subscriber. These three steps are done with this code:

```

func testJoke40() {
    let jokeRequester = JokerRequester()
    let signalProducer =
jokeRequester.signalProducerForJoke(40)
    signalProducer.startWithNext { (input:[String :

```

```

AnyObject]) -> () in
    XCTAssertNotNil(input["value"], "no value field")
        if let value = input["value"] as?
[String: AnyObject] {
            XCTAssertEqual(value["joke"] as? String, "A
handicapped parking sign does not signify that this spot is
for handicapped people. It is actually in fact a warning, that
the spot belongs to Chuck Norris and that you will be
handicapped if you park there.")
        }else {
            XCTAssertTrue(false, "can't convert the
'value' field")
        }
    }
}

```

Run this test and check whether it ends successfully. However, is this test really being tested? Let's check again by adding a Boolean variable that ensures that the subscriber was called. Add the highlighted code to your test for this improvement:

```

func testJoke40() {
var tested = false

    let jokeRequester = JokeRequester()
    let signalProducer =
jokeRequester.signalProducerForJoke(40)
    signalProducer.startWithNext { (input:[String :
AnyObject]) -> () in
        XCTAssertNotNil(input["value"], "no value field")
        if let value = input["value"] as?
[String: AnyObject] {
            XCTAssertEqual(value["joke"] as? String, "A
handicapped parking sign does not signify that this spot is
for handicapped people. It is actually in fact a warning, that
the spot belongs to Chuck Norris and that you will be
handicapped if you park there.")
        }else {
            XCTAssertTrue(false, "can't convert the
'value' field")
        }
tested = true
}
}

```

```
        XCTAssertTrue(tested, "invalid test")  
    }  
}
```

What happened? Your test failed because it ended before the request was completed. How can we solve this problem? Actually, `XCTest` has a feature called an expectation. An expectation is a variable that can tell whether the test has reached its end.

How does it work? Firstly, you have to create an expectation with a method called `expectationWithDescription`; after this, you have to check where your test should be considered finished and call an expectation method called `fulfill`. Finally, you have to call a method called `waitForExpectationsWithTimeout` at the end of your test; this will wait until the `fulfill` method is called or until it reaches a timeout. Five seconds for the timeout are more than enough. Remove the previously amended code and replace it with this new highlighted code:

```
func testJoke40() {  
    let expectation =  
expectationWithDescription("Expectation")  
  
    let jokeRequester = JokeRequester()  
    let signalProducer =  
jokeRequester.signalProducerForJoke(40)  
    signalProducer.startWithNext { (input:[String :  
AnyObject]) -> () in  
        XCTAssertNotNil(input["value"], "no value field")  
        if let value = input["value"] as?  
[String:AnyObject] {  
            XCTAssertEqual(value["joke"] as? String, "A  
handicapped parking sign does not signify that this spot is  
for handicapped people. It is actually in fact a warning, that  
the spot belongs to Chuck Norris and that you will be  
handicapped if you park there.")  
            expectation.fulfill()  
        }  
        else {  
            XCTAssertTrue(false, "can't convert the  
'value' field")  
        }  
    }  
}
```

```
    waitForExpectationsWithTimeout(5, handler: nil)  
}
```

Now run this test and have a look at something interesting. The test code is correct; however, it indicates that there is a failure on our application, as the request couldn't be performed because we haven't set **Application Transport Security** in the application's `info.plist` file. Fix it in a way that we learned in the previous chapter, run the test again, and check whether it finishes correctly. This is a good example of when a unit test can help you find incorrect behaviors in your application.

Testing the UI

As we learned at the beginning of this book, you can use ReactiveCocoa to change UI components whenever necessary. Unit tests are excellent for testing non-UI components; however, they lack support for UI components.

Until Xcode 6, the default way of testing the UI was using an instrument for automation. Now, Xcode 7 brings to you a new way of testing the UI, where you can record your UI usage and complete the test by changing or adding only a small part of the code if necessary.

To perform this test, we need an application with something in the UI. Let's try something simple: what about an application where I can press a button and a new joke appears?

The jokes to be presented are going to have their IDs stored in a file; thus, you have to add a new empty file (not a Swift file) called `selectedjokes.txt` to your application. Open it and just add the following numbers:

15 21 29 3

Feel free to add more numbers, which are joke IDs, if you want to have more fun. Now go to your storyboard and add a button and label to the only view we have. Connect both UI components to the View Controller, with the following names:

```
@IBOutlet weak var jokeLabel: UILabel!
@IBOutlet weak var requesterButton: UIButton!
```

Now, we have to go to the `ViewController.swift` file, import ReactiveCocoa, and complete the method `viewDidLoad`. Firstly, we have to retrieve the signal producer for the file reader. This operation is similar to the one we had in the unit tests:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let fileNumbersReader = FileNumbersReader()
    let path = NSBundle(forClass:
```

```

self.classForCoder).resourcePath!
    let fullPath = "\u{path}/selectedjokes.txt"
    let fileNumbersReaderSignal =
fileNumbersReader.signalProducerForFile(fullPath)

```

After that we can get the button signal producer and keep it on a constant.

```

let buttonSignal =
requesterButton.rac_signalForControlEvents(.TouchUpInside)
    .toSignalProducer()

```

These signals need to be unified. This means that every time we read a joke number, it shouldn't be displayed except when the button is pressed, and vice versa. What does this sound like? The reality is that it is again a use of `zipWith`. Let's combine both signals with the following code:

```
fileNumbersReaderSignal.zipWith(buttonSignal)
```

Is that all? Of course not; this combination of signals must continue with a new signal, which is the signal producer from `JokeRequester`. We can use the `flatMap` method to indicate that the `JokeRequester` signal is the one that will continue from now on. As we just want `jokeNumber`, we can ignore the second signal input by naming it with an underscore:

```

.flatMap(.Latest) {
    (jokeNumber:Int, _:AnyObject?) ->
SignalProducer<[String:AnyObject], NSError> in
    let jokeRequester = JokeRequester()
    return
jokeRequester.signalProducerForJoke(jokeNumber)
}

```

We can discard most of the JSON object returned by the server; the only part that we are interested in is the joke itself, which is a string. Let's do this JSON-to-string conversion by adding the following `map`:

```

.map { (input:[String : AnyObject]) -> String in
    let value = input["value"] as!
[String:AnyObject]
    let joke = value["joke"] as! String
    return joke
}

```

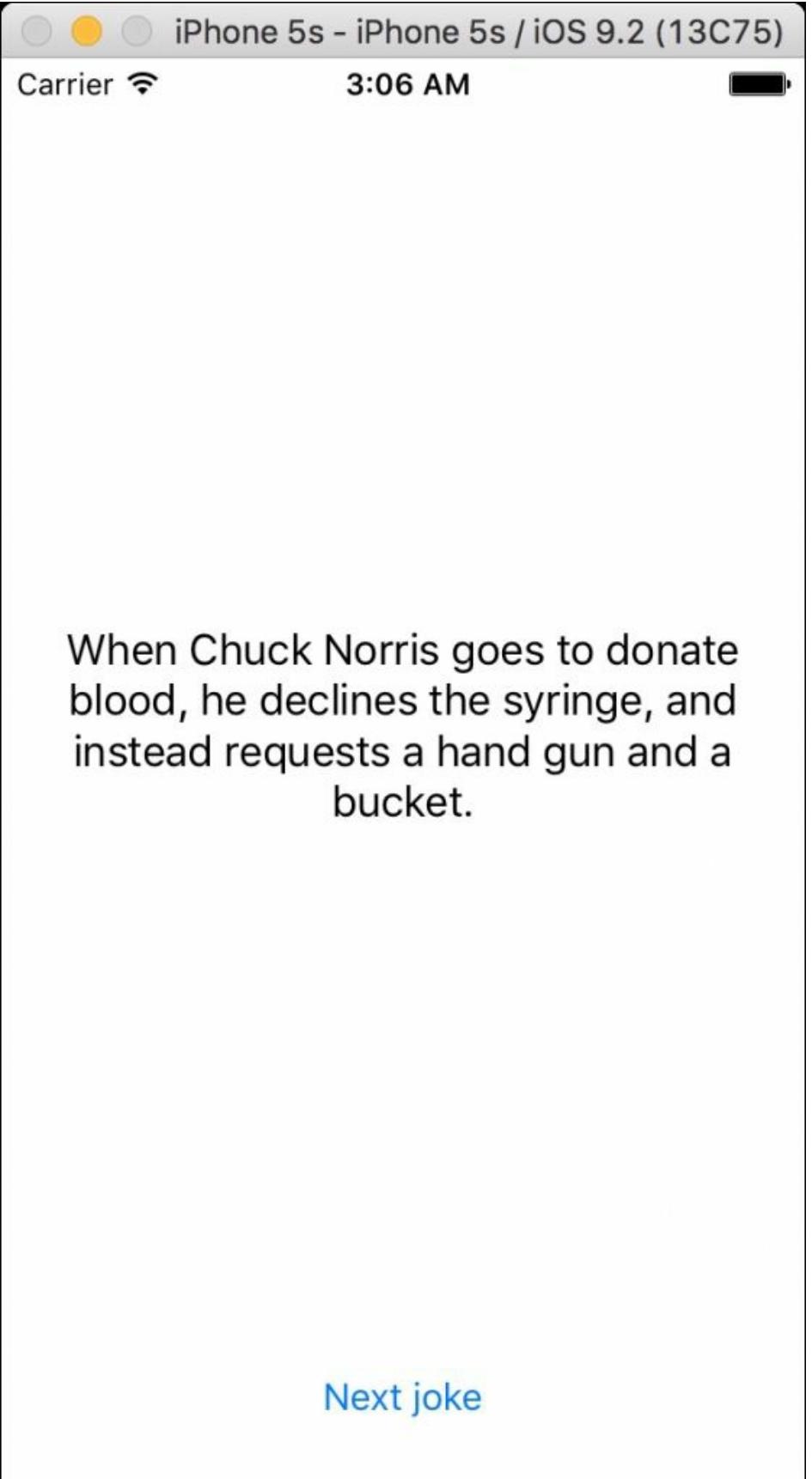
One last detail before presenting it on the screen: the joke is returned with an HTTP request, which runs in the background; therefore, before presenting it, we have to send it to the main thread using `UIScheduler`, which is the object that sends the execution to the UI thread, which as we know is the main thread:

```
.observeOn(UIScheduler())
```

Finally, we can get the joke and present it on the only label that we have on the screen, and that's all for this application:

```
.startWithNext {[weak self] (joke:String) -> () in
    self?.jokeLabel.text = joke
}
} // end viewDidLoad
```

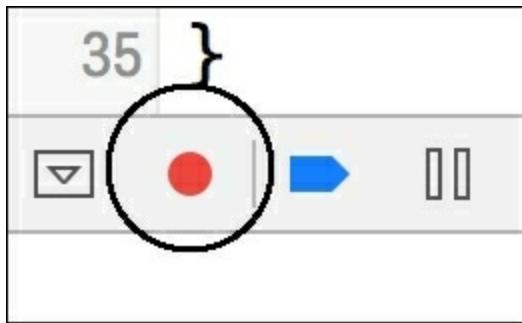
Run the application, and when you press the button, a new joke will appear until we have no more jokes. Your final screen should look something like the following screenshot:



Now, it is time to create a UI test for this application. Go to the UI test group and click on the `Chapter_7_TestingUITests.swift` file. Rename `testExample` as `testBasicOperation`, like this code:

```
func testBasicOperation() {  
    // Use recording to get started writing UI tests.  
}
```

The next step is recording UI usage. To do this, click on the red button located on the debug bar:



Ready to record? I hope so, as the application will start and once it starts, you just have to click on the next button and the label until you get the last joke. Stop recording by clicking again on the record button, and check the generated code, which should be similar to this:

```
func testBasicOperation() {  
  
    let app = XCUIApplication()  
    let nextJokeButton = app.buttons["Next joke"]  
    nextJokeButton.tap()  
    app.staticTexts["When Chuck Norris goes to donate  
blood, he declines the syringe, and instead requests a hand  
gun and a bucket."].tap()  
    nextJokeButton.tap()  
    app.staticTexts["Chuck Norris doesn't shower, he only  
takes blood baths."].tap()
```

```
    nextJokeButton.tap()
    app.staticTexts["Teenage Mutant Ninja Turtles is based
on a true story: Chuck Norris once swallowed a turtle whole,
and when he crapped it out, the turtle was six feet tall and
had learned karate."].tap()
    nextJokeButton.tap()

    let
chuckNorrisDoesnTReadBooksHeStaresThemDownUntilHeGetsTheInform
ationHeWantsStaticText = app.staticTexts["Chuck Norris doesn't
read books. He stares them down until he gets the information
he wants."]

chuckNorrisDoesnTReadBooksHeStaresThemDownUntilHeGetsTheInform
ationHeWantsStaticText.tap()
    nextJokeButton.tap()

chuckNorrisDoesnTReadBooksHeStaresThemDownUntilHeGetsTheInform
ationHeWantsStaticText.tap()
    // Use recording to get started writing UI tests.
}
```

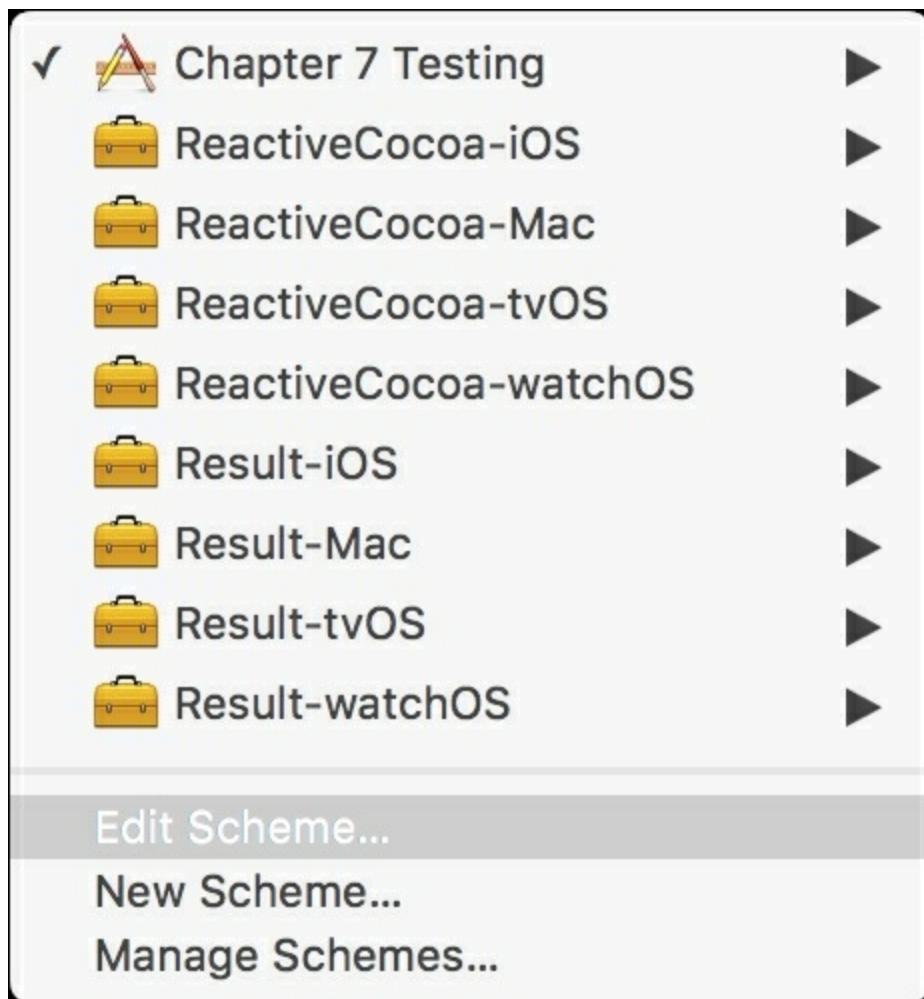
Run this test the same way you run other tests, and check whether the result is a success. Change any number in `selectednumbers.txt` and see that the UI test now fails as you don't receive the same texts.

Profiling with Instruments

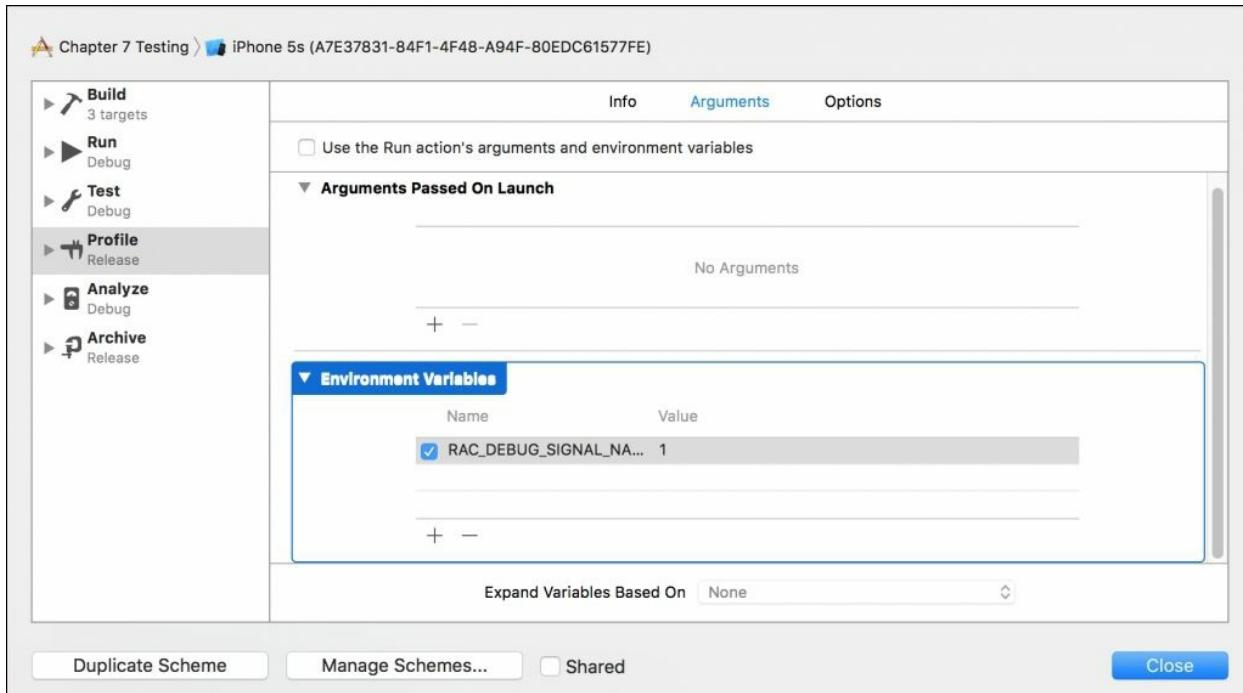
Unit tests are not everything; they cannot be done to localize problems such as memory leaks. Here is where another tool comes to help us find other kinds of problems: Instruments. ReactiveCocoa comes with two templates for Instruments: one of them is called **Disposable Growth** and the other is called **Signal Events**; both of them are located in a subfolder called `Instruments` in your ReactiveCocoa directory.

Their usage is not very intuitive and you have to get used to them for them to start being useful.

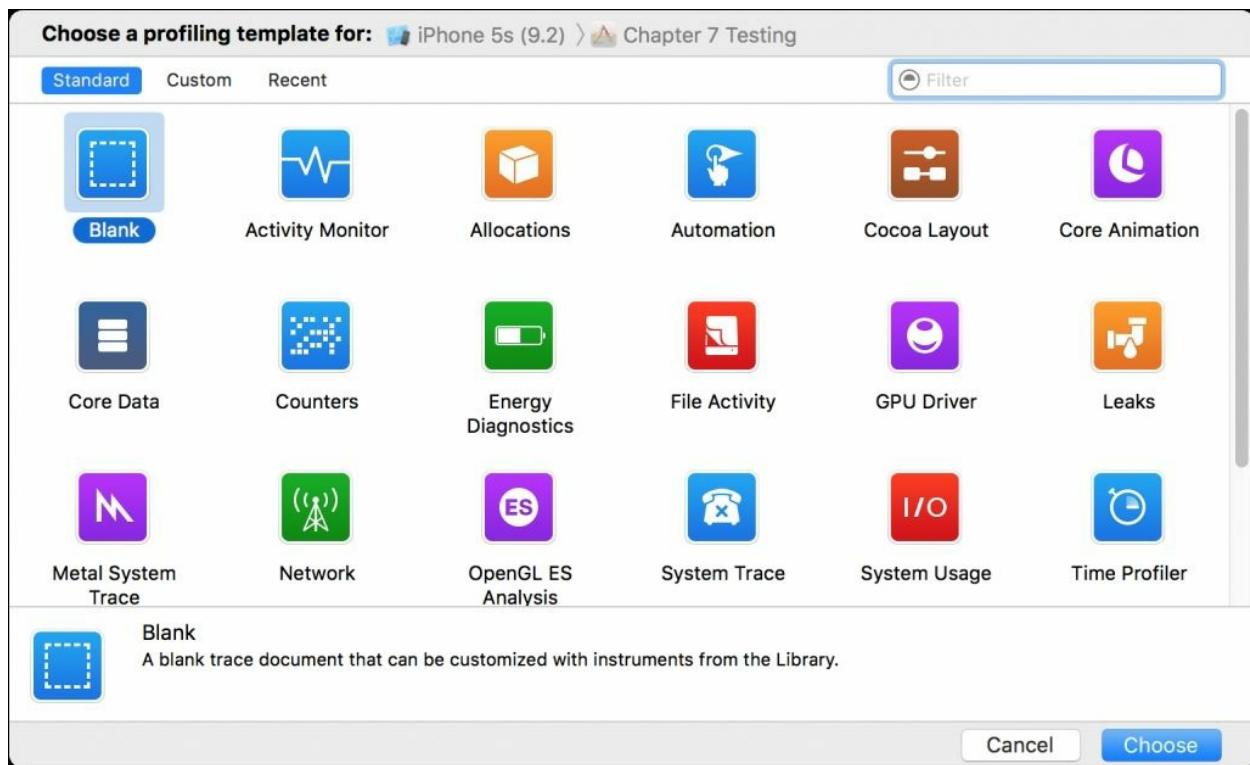
Click on both instrument templates; we have to start like this to be able to locate both `Instruments` in the library, so you can close `Instruments` now if you want to. Go to Xcode and go to the **Edit Scheme...** section, as demonstrated in the following screenshot:



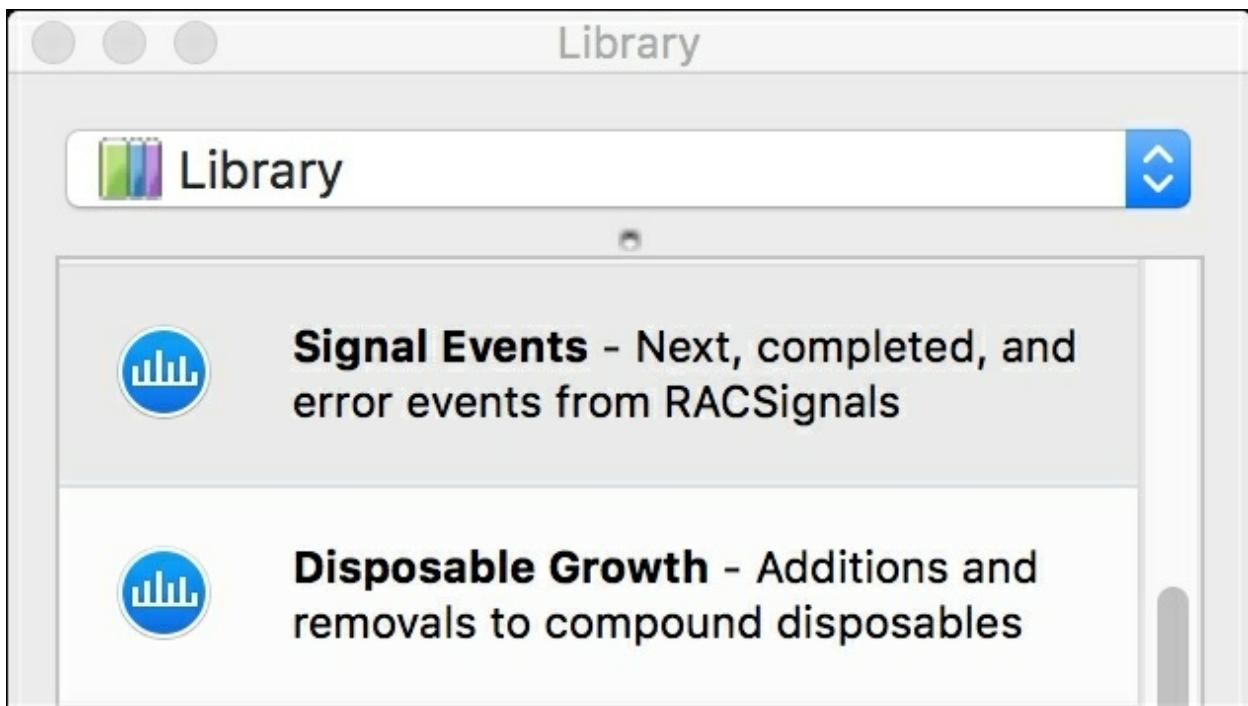
In the **Profile** section, unselect the option **Use the Run action's action arguments and environment variables**, and add a new environment variable called `RAC_DEBUG_SIGNAL_NAMES` and set its value to `1`, as shown in this screenshot:



Start profiling with *command + I*; it might take some seconds as it needs to recompile and open Instruments. Select the **Blank** instrument, as our template won't appear on this screen:



Once you have Instruments open, go to your library (if it is not visible, press *command + L*) and double-click on both **Disposable Growth** and **Signal Events**:



Press the record button, which is located at the top left corner of Instruments, and click on it once or twice to retrieve some information.

While you interact with the simulator (these Instruments don't work on devices), you will start receiving information about what is going on; this way, you can have a better idea of how many signals there are and where you are receiving them:

Signal Events				
#	Caller	Event	Subscriber	Signal
0	-[UIApplication sendAction:to:fro...]	next	<RACSubscriber: 0x7fd1e9c41c...	("+++++++" ... <UIButton: 0x
1	-[UIApplication sendAction:to:fro...]	next	<RACSubscriber: 0x7fd1e9c41c...	("+++++++" ... <UIButton: 0x

Great, now we know how to use two more Instruments, and they will help us when we try to detect bugs or when something is not behaving well in our application.

Summary

In this chapter, we learned how to use different tools for debugging. We started with RAC Marbles, which is a tool that allows us to visualize the behavior of a ReactiveCocoa method.

Then, we learned how to write unit tests, which is a common procedure for detecting bugs when some code has changed. Some development methodologies are based on unit tests, such as TDD.

Next, we had a session on doing UI tests, which are very similar to unit tests but are focused on checking whether the user operation works as expected.

Finally, we had a look at `Instruments`, which can give us more information about the application's status. Even if everything looks fine, you should use `Instruments` for checking what's going on behind the scenes, such as memory leaks or low performance. ReactiveCocoa comes with two templates for `Instruments`.

In the next chapter, we will learn how to convert a traditional application into an application that is ReactiveCocoa-based.

Chapter 8. Migrating a Real Application to ReactiveCocoa

We have learned how to use ReactiveCocoa. However, let's face reality: we don't start a new project everyday; therefore, you may need to upgrade your existing project to use reactive programming. Now, it's time to learn about how you can add ReactiveCocoa to an existing project like we did when we migrated an application to ReactiveCocoa.

Here, we will use an existing application with its source code, analyze it, and make the necessary changes to make it work with reactive programming.

In this chapter, we will cover the following topics:

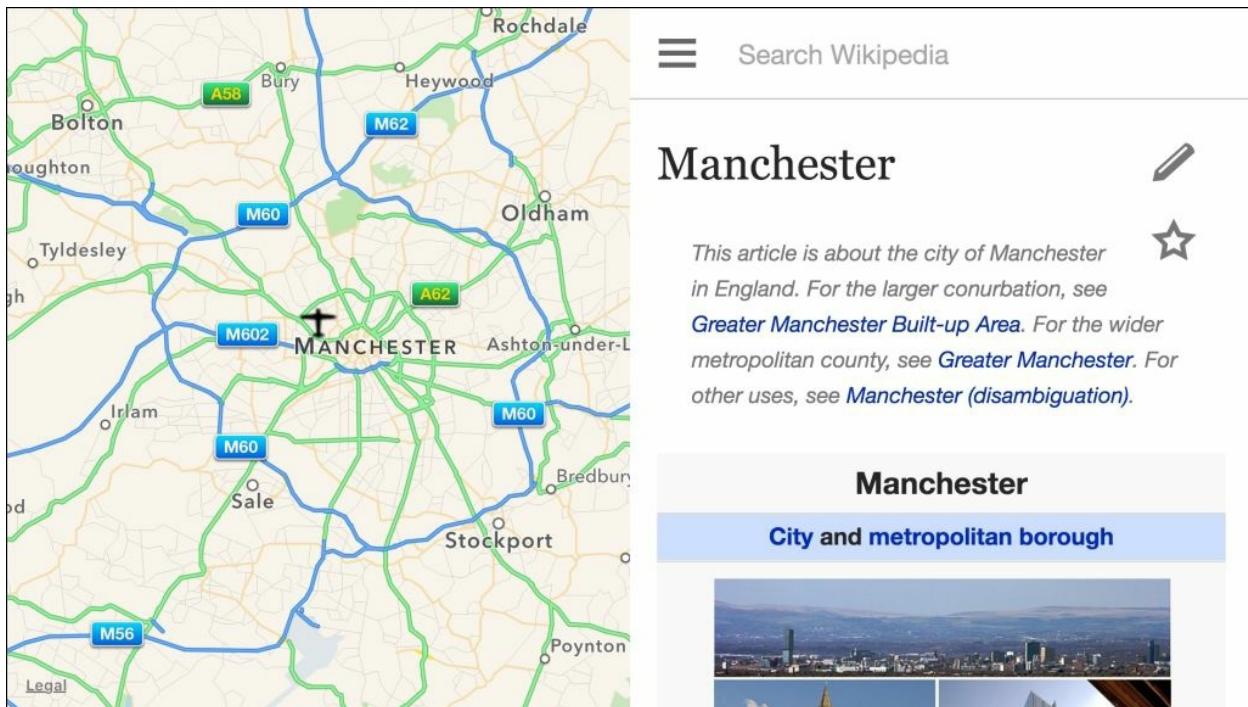
- Analyzing an existing code
- Migrating to ReactiveCocoa

Knowing the application

For this chapter, you have to download the application, Chapter 8 Airplane, from this book's resources. This application is about an airplane that flies over cities, and every time this airplane starts flying over a new city, it requests information about this city on the Internet and displays it on the left panel.

The airplane can be controlled by the user using the accelerometer; this means that you have to turn your physical device to turn the airplane. This detail is important as it shows us that we shouldn't use this application with the simulator; otherwise, it may be a bit restricted.

Download the application and run it, ensure that it is working and you can see the airplane moving and displaying information like in the following screenshot:



Once you have checked whether the application is working, we need to know

how it was developed. There is no sign of any Podfile or Carthage; however, it uses a local Git repository. There is no sign of usage of Core Data. There is only one View Controller that uses Core Location, Geocoder, MapKit, and WebView. This View Controller works as a delegate of the Core Location manager, of the map view, and also as the delegate of the airplane.

You can see there is an image on `Assets.XCAssets` called `Airplane` that contains a class that represents an airplane and a protocol that represents its delegate. This class uses **Core Motion** to detect the device's rotation and `NSTimer` to move the airplane.

There is also another file called `UIImageExtension.swift`, which contains an extension for `UIImage` that allows us to rotate the image. As it is only one method and doesn't work asynchronously, we will not worry about it.

After these notes, we more or less have an idea about the asynchronous calls that we have in this application, how this application works, and how we can make these changes.

Something that is also important is profiling the current application. Take a look at its current status, such as the amount of memory that this application consumes, the CPU usage, and so on. This step is important: imagine that you detect a memory leak during half migration; it will be much easier to find its origin if we know that it is only related to the new code.

Tip

Don't think that because one application is working this means that it has no internal problems adding ReactiveCocoa.

First, you have to install ReactiveCocoa into your application. Ensure that every change is committed, so if for any reason you have to perform a rollback, then it will be easy.

As this application doesn't use CocoaPods or Carthage, you should investigate why. Even if it is not very often, some applications don't use CocoaPods or Carthage for any problem that they had in the past. If this is not the case and

your team is okay with using these package systems, then go on and install it using your favorite one. However, if there is anything that blocks you from using these package systems, use the traditional way (`git submodule`). For this example, we will use the `git submodule` command. Open your terminal, go to this project's folder as we learned in [Chapter 2, Installing ReactiveCocoa and Using It with Playground](#), and add the submodule with the following command:

```
git submodule add  
https://www.github.com/ReactiveCocoa/ReactiveCocoa
```

This process may take a while. Once the shell returns the prompt for you, you can enter the `ReactiveCocoa` folder and call the bootstrap with the following commands:

```
cd ReactiveCocoa  
script/bootstrap
```

Open the Xcode project and the current folder on a finder window with the `open .` (`open` and a dot) command. Add `ReactiveCocoa` and `Result` projects to your application project.

As this application doesn't contain any Objective-C code, you don't have to worry about setting the **Embedded Content Contains Swift Code** record in the **Build Settings** section to **YES**. If you do this, it won't hurt. However, you still have to go to the general settings and add `ReactiveCocoa` and `Result` as embedded binaries for this project.

You can finish your installation by going to the `ViewController.swift` file and import the `ReactiveCocoa` framework by adding the following line before the class declaration:

```
import ReactiveCocoa
```

It doesn't matter right now if this file will need the `ReactiveCocoa` framework, this is just to test and ensure that the installation worked. You can commit your files again.

Creating a new framework

Do you think that it is easier fighting against a lion or against 20 kittens? Here, we have the same problem. Changing a small application such as this one is not difficult; however, changing a huge application may be very complicated. However, what if we divided the application into small chunks?

One good starting point is changing parts of your application and making it work with ReactiveCocoa, and a good way to do this is creating a framework. It is also a good idea to create this framework to reuse your code for future developments.

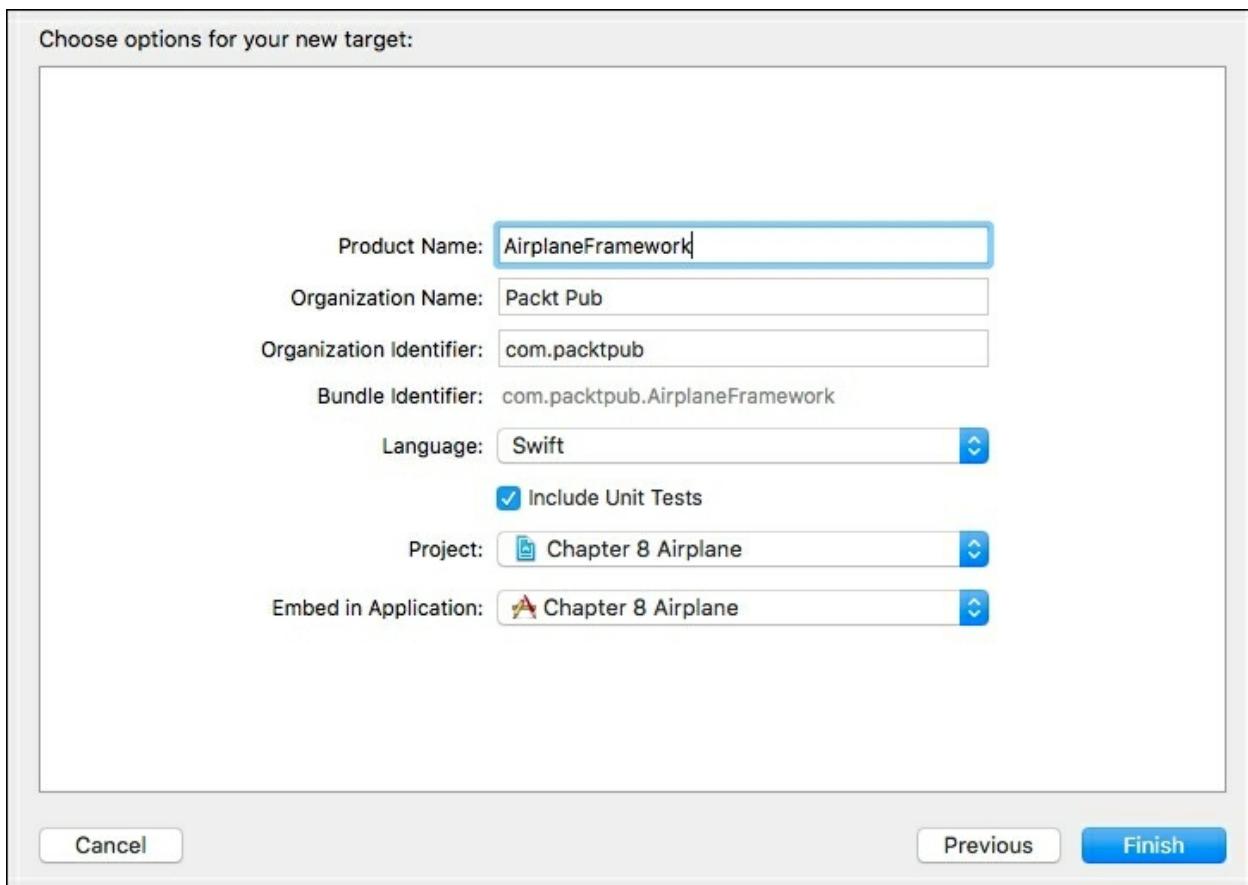
Note

As Swift doesn't support static libraries, this option mustn't be provided when using ReactiveCocoa.

Set a new target for your project, but this time pay attention as you have to select the **Framework & Library** section, and select the **Cocoa Touch Framework** option as it is demonstrated in the following screenshot:



When the dialog that requests data for this framework appears, you have to set a few details. Call this framework `AirplaneFramework`. Now, ensure that `Swift` is the main language, check **Include Unit Tests** because this is a good way to test a specific feature as we saw in [Chapter 7, Testing Your Application](#). Of course, this project has to be `Chapter 8 Airplane`, and **Embed in Application** should be `Chapter 8 Airplane` as you can see in the following screenshot:



Starting with the GPS, we can create an extension for the `CLLocationManager` class. The easiest method to do this is the authentication method. We can start by trying to detect whether the authentication has changed, and then we can request it if it is necessary.

Add a new file to `AirplaneFramework` called `CLLocationManagerExtension.swift`. Start importing `ReactiveCocoa` and `Core Location`, as we will need both of them:

```
import CoreLocation
import ReactiveCocoa
```

Now, open an extension for the `CLLocationManager` class, indicating that it now implements the `CLLocationManagerDelegate` protocol. This is important as we want to get rid of this delegate pattern; therefore, each

location manager needs to assign itself as delegate:

```
extension CLLocationManager: CLLocationManagerDelegate {
```

Now, we need a function that returns the corresponding signal producer, which will trigger every time the authorization status has changed. Let's call this `rac_authStatusChanged`. This function must be public as it will be used outside the framework.

Tip

Try to name your ReactiveCocoa functions with the prefix `rac_`. This will make it easy for you to locate them when typing them.

Create a function header that returns a signal producer based on the value of `CLAuthorizationStatus`. Paste the following code to open the function:

```
public func rac_authStatusChanged() ->
SignalProducer<CLAuthorizationStatus, NSError> {
```

Due to the selector for the authorization, `changed` will be called only if there is a delegate set. We have to start by checking whether this is `nil`. In this case, we have to set the current object as its own delegate:

```
if delegate == nil {
    self.delegate = self
}
```

Now, we can detect when the selector for authorization is called; this will return the following `RACSignal` object for us:

```
return self.rac_signalForSelector(
Selector("locationManager:didChangeAuthorizationStatus:"),  
fromProtocol: CLLocationManagerDelegate.self)
```

This `RACSignal` will be converted to a signal producer with `toSignalProducer`. This way, we can develop using the new ReactiveCocoa way of development. Then, we have to map it

to CLAuthorizationStatus:

```
.toSignalProducer()
.map { (input:AnyObject?) -> CLAuthorizationStatus
in
    let tuple = input as! RACTuple
    if let value = tuple.second as? NSNumber{
        return CLAuthorizationStatus(rawValue:
value.intValue) !
    }else {
        return CLAuthorizationStatus.NotDetermined
    }
}
```

Now that this function is done, we have to update the code on the application. First, remove the code that sets the location manager, the request for authorization, and the `didChangeAuthorizationStatus` method.

Then, we have to add the equivalent code to the one that was just deleted. The difference is that the code now will use a signal producer instead of the MVC pattern. Your `viewDidLoad` method now should look like the following:

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

mapView.delegate = self
airplane.delegate = self

let authChanged =
locationManager.rac_authStatusChanged()

authChanged.filter({ (input:CLAuthorizationStatus) ->
Bool in
    return input == .NotDetermined
}).startWithNext { [weak self]
(auth:CLAuthorizationStatus) -> () in

self?.locationManager.requestWhenInUseAuthorization()
}

authChanged.filter { (input:CLAuthorizationStatus) ->
Bool in
    return input == .AuthorizedWhenInUse
}
```

```

        }.startWithNext { [weak self]
(auth:CLAuthorizationStatus) -> () in
    self?.locationManager.startUpdatingLocation()
    self?.mapView.showsUserLocation = false
}
}

```

Can we test this? For this specific test, we can use the simulator as you can just reset the simulator contents (on the simulator menu and by selecting the **Reset Contents and Settings...** option) if anything fails and try again. Run your application on the simulator and you will see that the authorization request works. However, something else is broken now: the location updates.

It is part of the process to break stuff while performing the migration, but the important part is how to continue. If you are creating signals for the location manager, then just focus on this and do not start on other signals; otherwise, fixing the application may get complicated.

In case of needing many functions for a delegate, you may need to create signals for all of them and test them with unit tests.

Tip

Migrating by writing unit tests first, like people do in **Test-Driven Development (TDD)**, is a good way to ensure that you have everything that you need. It is good practice.

Once we have understood this principle, return to `CLLocationManagerExtension.swift`. Here, we just need to follow the example of the previous method and create a similar one:

```

public func rac_updateLocation() ->
SignalProducer<[CLLocation], NSError> {
    if delegate == nil {
        self.delegate = self
    }
    return
self.rac_signalForSelector(Selector("locationManager:didUpdate
Locations:"),
fromProtocol: CLLocationManagerDelegate.self)
    .toSignalProducer()

```

```

        .map({ (input: AnyObject?) -> [CLLocation] in
            let tuple = input as! RACTuple
            return tuple.second as! [CLLocation]
        })
    }
}

```

Now, we just need to return to View Controller, delete the current implementation of the `didUpdateLocations` method, and add the equivalent code in `viewDidLoad`. Remember that, in this case, we want only one location for the start position. This is somewhat important, as the migration should not only be replacing code, but also trying to understand the idea of the code. In this case, we can use the `take` function to ensure that we will call this subscriber only once. Add the following code at the end of `viewDidLoad`:

```

locationManager.rac_updateLocation()
    .filter({ (locations:[CLLocation]) -> Bool in
        return locations.count > 0
    })
    .take(1)
    .startWithNext { [weak self] (locations:
[CLLocation]) -> () in
        if self != nil {
            self!.locationManager.stopUpdatingLocation()
                let span = MKCoordinateSpanMake(180 /
pow(2, 10) * Double(self!.mapView.frame.size.height) / 256.0,
0)

            self!.mapView.setRegion(MKCoordinateRegionMake(locations.first!
.coordinate, span), animated:false)

            self!.airplane.takeOffFrom(locations.first!)

            self!.mapView.addAnnotation(self!.airplane.annotation!)
        }
    }
}

```

Run the application now, and it should work as it did before. This means that the first step to migrate this application is done.

Replacing the airplane delegate

Checking the airplane code, you can figure out that it has its own delegate. This delegate is called every fifth of a second. How can we replace this code with a reactive one? First, you have to check that `NSTimer` calls a selector, which is not the philosophy of ReactiveCocoa, and then this selector does some stuff.

This time, we need to create two signals: one for `NSTimer`, and another one for the airplane. Why don't we create only one signal? Whenever you have the opportunity, try to make your code more modular because this will make your code easier to maintain. Creating a signal for `NSTimer` allows you to reuse this signal in future versions of this software or even in a different project.

Add a new file to `AirplaneFramework` and call it `NSTimerExtension.swift`. Here, we can add a new public function that creates a signal, which is triggered every time the timer is fired. When the subscriber is called, it is not necessary to send any information; this is the reason that this signal is based on the `void` value. Add the following code to create this signal producer:

```
import ReactiveCocoa

extension NSTimer {
    public class func
    rac_signalWithRepeatedInterval(interval:NSTimeInterval) ->
    SignalProducer<Void, NoError>{
        return SignalProducer<Void, NoError>(
            { (observer:Observer<Void, NoError>,
disposable:CompositeDisposable) -> () in
                let fireDate = CFAbsoluteTimeGetCurrent()
                let timer =
CFRunLoopTimerCreateWithHandler(kCFAllocatorDefault, fireDate,
interval, 0, 0, { (timer:NSTimer!) -> Void in
                    observer.sendNext()
                })
                CFRUNLoopAddTimer(CFRUNLoopGetCurrent(),
timer, kCFRunLoopCommonModes)
            })
    }
}
```

The previous code basically creates a new signal producer, which calls the `CFRunLoopTimerCreateWithHandler` function, which creates a timer with the specified time. Whenever this timer is called, we call the `sendNext` function.

Excellent, at this point, the signal done. Now, it is time to change the take off function. Again, we have to analyze the behavior of this function. The original code when the airplane took off triggered a timer, and this timer called the airplane delegate. The idea is simple, but now we need to think about how we can refactor it.

If we start with one void signal (the `NSTimer` signal), then it needs to continue with another signal that sends the current airplane with its position to the next subscriber. Starting with one signal and continuing with another one sounds like the usage of the `flatMap` method. That's right, whenever you have a case of starting with one signal and continuing with another, think about `flatMap`. Let's see how we can implement this.

Go to the `Airplane.swift` file. Here, we will start creating a new method that will replace the `takeOff` method. Start by creating its header that, as we know, requires the start position and returns a signal producer:

```
func rac_takeOffSignal(location:CLLocation) ->
SignalProducer<Airplane, NSError>{
```

To make things easier, move the accelerometer updates to the top of this function. This way, we can ensure that it will be called only once and not every time the timer is fired:

```
    self.location = location
    motionManager.startAccelerometerUpdates()
```

Then, we can return the signal that comes from `NSTimer`, respecting the same time interval (0.2 seconds). If you think that you have to change this interval, then try do it eventually. It's safer keeping the same constants and trying to achieve the same behavior because if anything acts differently, then we would have to check whether it was caused due to the constant change or for any other

reasons:

```
return NSTimer.rac_signalWithRepeatedInterval(0.2)
```

Once this signal is triggered, we need to convert the signal error from `NoError` to `NSError`. This is where we have to use the `flatMapError` function. In this function, we can just instantiate a new signal with the `NSError` error type, and it starts with the current signal value. As the timer signal is void-based, we just need to send a void value with parenthesis:

```
.flatMapError({ (error:NoError) ->
SignalProducer<(), NSError> in
    return SignalProducer<(), NSError>(value: ())
})
```

Now, this signal must be converted to the correct signal producer type. Here, we have to use the `flatMap` function, as follows:

```
.flatMap(FlattenStrategy.Latest, transform: {
[weak self]() -> SignalProducer<Airplane, NSError> in
```

This transform handler needs to return a signal producer that can be instantiated with a handler that receives the observer to send the next signal:

```
return SignalProducer<Airplane, NSError>({ [weak
self] (observer:Observer<Airplane, NSError>,
disposable:CompositeDisposable) -> () in
```

Now, this is very straightforward as the code is basically the same as the original take off function; however, at the end, we have to send the current object to the next subscriber:

```
if self != nil {
    let directionRadians = self!.direction
* M_PI / 180.0

    let xOffset = cos(directionRadians) *
self!.speed
    let yOffset = sin(directionRadians) *
self!.speed

    let longitude =
```

```
self!.location.coordinate.longitude.advancedBy(xOffset)
        let latitude =
self!.location.coordinate.latitude.advancedBy(yOffset)
        self!.location = CLLocation(latitude:
latitude, longitude: longitude)

        if let accelerationData =
self!.motionManager.accelerometerData {
            if
abs(accelerationData.acceleration.y) > 0.25 {
                if
accelerationData.acceleration.y > 0{
                    self!.turnLeft()
                } else {
                    self!.turnRight()
                }
            }
        }
    observer.sendNext(self!)
}
```

If for any reason the `self` variable is `nil`, then this means that the `airplane` object was already destroyed. In this case, we have to send the completion. After this is done, the code for `rac_takeOffSignal` is over:

```
        else{
            observer.sendCompleted()
        }
    })
}
}
} // end rac takeOffSignal
```

The `takeOffSignal` implementation is completed; now it can be started whenever we get the first location. If you are developing any unit test, then now would be a good moment to implement it. In this case, we have to return to the `ViewController.swift` file.

Remove `AirplaneDelegate` from the View Controller header. This is important, because we are ensuring that when running or testing the application, the old code is not being used. At this point, the class header should be like the following line:

```
class ViewController: UIViewController, MKMapViewDelegate {
```

Go to the `viewDidLoad` method and remove the line that sets the `airplane` delegate and the `takeOff` and `addAnnotation` calls. Then, at the end of the location subscriber, we can start calling the new `take off` method:

```
    self!.airplane.rac_takeOffSignal(locations.first!)
```

This signal needs a subscriber for each time that the signal is called because this means that the airplane has moved. Continue the last code with the following line to add the next subscriber:

```
    .startWithNext({ [weak self] airplane in
```

For a safe code, we have to start checking whether `self` is different from `nil`. This shouldn't happen as `self` represents the only View Controller that we have on our application and it won't be dismissed. However, remember that software development is always a box of surprises. Therefore, we have to ensure that this handler's contents are wrapped with an `if` statement:

```
        if self != nil {
```

Inside this `if` statement, we can start by removing the previous annotation, which contains the last airplane position, and adding it again as this will refresh the airplane icon on the map:

```
            self!.mapView.removeAnnotation(airplane.annotation!)
            self!.mapView.addAnnotation(airplane.annotation!)
```

After adding the annotation, we can continue by setting the `mapView` information:

```
        let span = MKCoordinateSpanMake(180 / pow(2, 10) *
Double(self!.mapView.frame.size.height) / 256.0, 0)
        self!.mapView.setRegion( MKCoordinateRegionMake(
airplane.annotation!.coordinate, span), animated:true)
```

Then, we have to check whether we had an interval of 10 seconds or more. Why is this application doing this? As mentioned before, we have to understand the code meaning. This will allow us to perform a better analysis

when refactoring this part. We are going to use the geocoder reverse location, which works receiving a coordinate as input and returning the corresponding address (or addresses). If we make a request too many times, then the geocoder starts rejecting the requests and replies with `NSError`.

Now, we can continue with the original code, but bear in mind that there is a pending task that is controlling this geocoder timer elapse:

```
if self!.lastTimeGeocoder == nil ||  
self!.lastTimeGeocoder!.timeIntervalSinceNow < -10 {  
    self!.lastTimeGeocoder = NSDate()  
    self!.geocoder  
        .reverseGeocodeLocation(airplane.location) { [weak self]  
(placemarks:[CLPlacemark]?, error:NSError?) -> Void in  
        if let error = error {  
            print(error.localizedDescription)  
            return  
        }  
        if let placemark = placemarks?.first,  
            addressDictionary =  
placemark.addressDictionary,  
            city = addressDictionary["City"] as? String  
        {  
            if city != self?.previousCity {  
                self?.previousCity = city  
                if let url = NSURL(string:  
"https://www.google.com/search?btnI=I&q=wikipedia%20\\(city)"") {  
                    let request = NSURLRequest(URL:  
url)  
                    self?.webView.loadRequest(request)  
                }  
            }  
        }  
    }  
}) // end rac_takeOffSignal  
} // end startWithNext
```

Remove the `hasMoved` method, which belongs to `airplaneDelegate`. Thus, we can be confident that the code that is running has nothing to do with the `airplane` delegate. Rebuild your application and run it again. Test it by

rotating your phone and flying from one city to another. Remember that now you have to test this on a physical device, as the simulator has no feature that allows us to test the accelerometer. It is a good time to commit your changes.

Reorganizing the signals

The current code works perfectly. However, there is a small detail that can make this even better. We will start a signal and its subscriber starts another signal. There is nothing wrong with this. However, cases like this one usually mean that from one signal we must switch to another. This is a good advantage of using functional programming because we are taking advantage of a chain pattern.

Once we know our new goal, we can call the `flatMap` function between the `take(1)` and `startWithNext` calls. The idea is to stop updating the GPS (as we don't need it anymore), take its coordinates and continue with the take off signal. Start updating your code by adding the following highlighted code:

```
.take(1)
    .flatMap(FlattenStrategy.Latest, transform: {
[weak self](locations:[CLLocation]) ->
SignalProducer<Airplane, NSError> in
    })
.startWithNext({[weak self] airplane in
```

Now, we have to fill this `flatMap` method. Basically, its implementation is already done, it is on `startWithNext`. You just need to move its code until the `rac_takeOffSignal` call to `flatMap`.

Now, we have a new case that should never happen, but we have to consider it for compilation and code security reasons: when `self` is `nil`. In this case, we can just return an empty signal. What is an empty signal? It is a signal that whenever it is created, it calls the completion method without calling any subscriber. After knowing how we have to implement the `flatMap` handler, we can do this by adding the following highlighted code:

```
.flatMap(FlattenStrategy.Latest, transform: {
[weak self](locations:[CLLocation]) ->
SignalProducer<Airplane, NSError> in
    if self != nil {

self!.locationManager.stopUpdatingLocation()
let span = MKCoordinateSpanMake(180 /
```

```

pow(2, 10) * Double(self!.mapView.frame.size.height) / 256.0,
0)

self!.mapView.setRegion(MKCoordinateRegionMake(locations.first!
!.coordinate, span), animated:false)

        return
self!.airplane.rac_takeOffSignal(locations.first!)
}
return SignalProducer<Airplane, NSError>.empty

} )

```

Remember that this is not a new code, this code was moved from `startWithNext`. Therefore, the beginning of this method now should be like this:

```

.startWithNext({ [weak self] airplane in
    if self != nil {

self!.mapView.removeAnnotation(airplane.annotation!)
    self!.mapView.addAnnotation(airplane.annotation!)

```

Again, you can rebuild your application and test it. Make sure that everything is working as it did before.

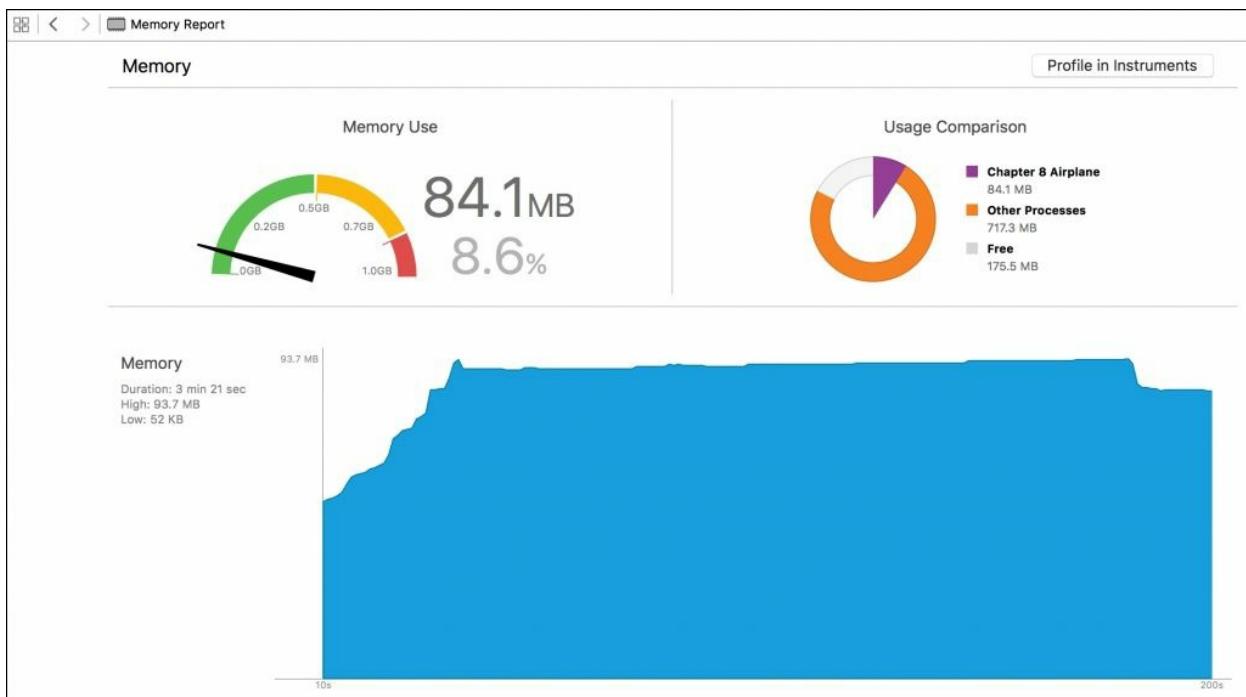
Checking the dark side

Until now, we resettled some codes and this seems to be working. However, at this point, we have to do something very important: profiling. Why? Remember that sometimes we can have some unexpected surprises such as memory leaks.

Tip

Don't migrate the whole application without profiling it. This will make it more difficult to detect where the problem is.

Run your application, and while running it, return to Xcode. Using the key combination, *command + 6*, you will go to the **Debug Navigator**. Click on the **Memory** section and make sure that the application contains a similar graph as it used to have with its original code. The memory should keep plain for a long time, and when we switch from one city to another, the memory first decreases and then increases. You may have some results like the following screenshot:



If you detect that something is worse than before, profile this application on instruments and try to detect where the leak comes from. Don't forget to run the ReactiveCocoa Instruments templates, as we learned to do in the previous chapter.

Once the memory is checked, do the same with the CPU and the other sections. If any results are similar but you know that they can be improved, such as the CPU has too much consumption in this application, then this is okay. Right now, this is not time to achieve a better performance, the idea is only migrating the current application to ReactiveCocoa. A better performance should be attempted only when the whole application is migrated.

After profiling, commit your changes and start looking for the next step.

Splitting the signal again

After `flatMap`, we need to update the annotation on the map by removing the current one and reinserting it on the map (this is the way map view works). This will make us create another `flatMap` call again. Another `flatMap` call can be made, as we can update the `mapView` property and send the next signal.

This idea is similar to the previous one, we just need to move some code from `startWithNext` and add it to a new `flatMap` function. Move this code according to the following sample:

```
.flatMap(FlattenStrategy.Merge, transform: { [weak self] (airplane:Airplane) -> SignalProducer<Airplane, NSError>
in
    if self != nil {
        return SignalProducer<Airplane, NSError>({ [weak self] (observer:Observer<Airplane, NSError>, disposable:CompositeDisposable) -> () in
            if self != nil {

self!.mapView.removeAnnotation(airplane.annotation!)
                self!.mapView.addAnnotation(airplane.annotation!)
                let span = MKCoordinateSpanMake(180 / pow(2, 10)
* Double(self!.mapView.frame.size.height) / 256.0, 0)
self!.mapView.setRegion( MKCoordinateRegionMake(
airplane.annotation!.coordinate, span), animated:true)
                observer.sendNext(airplane)
            }
        })
    }
    return SignalProducer<Airplane, NSError>.empty
})
```

Test your application again and check whether everything is still working.

Waiting for 10 seconds

As mentioned before, the reverse geocoder will be updated every 10 seconds. We can do this using the `NSTimer` signal that we have completed. However, let's try something different now. What about using another scheduler?

We learned in the previous chapter how to use `UIScheduler`. However, there are more schedulers.

ReactiveCocoa brings a protocol called `DateSchedulerType`, which is used to trigger the signal according to time intervals. As it is a protocol, we can't instantiate it. We have to use a class that implements this protocol.

There is one class called `QueueScheduler` that implements `DateSchedulerType`. The only detail that we have to give to this class is the GCD queue that we desire to use. Here, we are going to use the main queue. However, for better performance, you can try to create your own queue afterwards.

Scroll up to the beginning of the `viewDidLoad` method and create a signal that is fired every 10 seconds, called `tenSecondsSignal`, with the following code:

```
let tenSecondsSignal = SignalProducer<Void, NoError>
(value: ())
    .delay(10, onScheduler: QueueScheduler(queue:
dispatch_get_main_queue()))
```

Great! Now we have the new signal that is triggered every 10 seconds. How can this signal sync up with `takeOffSignal` that we already have? The answer is very easy; we just need to use the `sampleOn` function. This function works with two signals. The main signal can trigger any number of calls. However, all of them are dropped until we receive a call from the second signal. When we receive a call from the second signal, it merges with the last call from the first signal.

So, what do we have to do to use `sampleOn`? It is very easy, we just need to

add the following call before `startWithNext`:

```
.sampleOn(tenSecondsSignal)
```

Delete the `lastTimeGeocoder` property, as we are not using this method anymore. Test your application again, and check whether there is any difference between this method and the original code.

As you can see, there is one difference. When the application starts, it doesn't try to load a website as it needs to wait for 10 seconds to start loading it. At this point, you have to decide if this new behavior is acceptable or not. If it is not acceptable, then search for another solution. Implementing your class of type `DateSchedulerType` could be one possible solution. If it is acceptable, then just use it as is and step to the next level.

Reversing the geolocation

Now, we need to move the geocoder into another `flatMap` function. Every time we receive new information, we can resend it as a string. This way we can check whether the user is still in the same city or if they already changed to another one. This string is the current city name. Therefore, the new `flatMap` should be this one and must be placed before the `startWithNext` call:

```
.flatMap(FlattenStrategy.Latest, transform: {
[weak self] (airplane:Airplane) -> SignalProducer<String,
NSError> in
    return SignalProducer<String, NSError>({ [weak
self] (observer:Observer<String, NSError>,
disposable:CompositeDisposable) -> () in
        if self != nil {
            self!.geocoder
                .reverseGeocodeLocation(airplane.location) { (placemarks:
[CLPlacemark]?, error:NSError?) -> Void in
                    if let error = error {
                        print(error.localizedDescription)
                        return
                    }
                    if let placemark =
placemarks?.first,
                        addressDictionary =
placemark.addressDictionary,
                        city =
addressDictionary["City"] as? String
{
                        observer.sendNext(city)
}
}
}
})
})
```

Again, test the code and check whether it is working as expected.

Tip

Don't forget to commit your changes if you are using a version control system.

Avoiding repeated calls

Every time we call the next subscriber, we have to check whether the city has changed using a property called `previousCity`. Our final goal is removing this property. Remember that functional programming doesn't use many variables, but how can we remove this property and still avoid calling the same website again and again? The answer is very easy: signal producers have a method called `skipRepeats`, which has the same functionality that we want with this property. Thus, the final change will be this code:

```
.skipRepeats()  
.startWithNext({ [weak self] city in  
    if let url = NSURL(string:  
"https://www.google.com/search?btnI=I&q=wikipedia%20\(city)") {  
        let request = NSURLRequest(URL: url)  
        self?.webview.loadRequest(request)  
    }  
}) // end startWithNext
```

Summary

This is the last chapter of this book. We learned an important task: how to add ReactiveCocoa to an existing application. As you can appreciate, we had to perform this upgrade step by step, starting with analyzing the code and creating signals and functions in some independent parts.

We should try to reduce the usage of the MVC pattern and test the application frequently. Once this application is done, try to review it. You will probably find something that can be fixed or improved.

That's all for this book. I hope you enjoyed it and wish that after reading this book, you will want to start a new project with ReactiveCocoa.