Beginning Reactive Programming with Swift

Using RxSwift, Amazon Web Services, and JSON with iOS and macOS

Jesse Feiler

Apress®

Beginning Reactive Programming with Swift

Using RxSwift, Amazon Web Services, and JSON with iOS and macOS

Jesse Feiler

Apress®

Beginning Reactive Programming with Swift: Using RxSwift, Amazon Web Services, and JSON with iOS and macOS

Jesse Feiler
Plattsburgh, New York, USA

ISBN-13 (pbk): 978-1-4842-3620-8 ISBN-13 (electronic): 978-1-4842-3621-5
https://doi.org/10.1007/978-1-4842-3621-5

Library of Congress Control Number: 2018955902

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

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

For information on translations, please email rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-3620-8. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Jesse Feiler is a developer, consultant, trainer, and author specializing in database technologies and location-based apps. He is the creator of Minutes Machine, the meeting-management app, as well as the Saranac River Trail app, a guide to the trail that includes location-based updates as well as social media tools. His apps are available in the App Store and are published by Champlain Arts Corp. Jesse is heard regularly on WAMC Public Radio for the Northeast's *The Roundtable.* He is founder of Friends of Saranac River Trail, Inc. A native of Washington, D.C., he has lived in New York City and currently lives in Plattsburgh, NY.

About the Technical Reviewer

A passionate developer and experience enthusiast, **Aaron Crabtree** has been involved in mobile development since the dawn of the mobile device. He has written and provided technical editing for a variety of books on the topic, as well as taken the lead on some very cool cutting-edge projects over the years. His latest endeavor, building apps for augmented reality devices, has flung him back where he wants to be: an early adopter in an environment that changes day-by-day as new innovation hits the market. Hit him up on Twitter, where he tweets about all things mobile and AR: `@aaron_crabtree`.

Introduction

As technologies change, we see how basic patterns recur over time. In many ways, there aren't that many new things to learn — just new variations and combinations of existing technologies and concepts. (See my book "Learn Computer Science with Swift" for more on the patterns that recur).

As always, there are many people to thank for helping on this book. Most important are the people who have contributed to the technologies. When it comes to the many open source technologies (including ReactiveX and its projects), there are more and more people working on the technologies, and that makes it easier for everyone.

Closer to home, Aaron Crabtree has provided very helpful and watchful comments on the manuscript. And, as always, Carole Jelen at Waterside Productions has helped make this book possible.

PART I

Building Composite Apps with Swift

CHAPTER 1

Building Blocks: Projects, Workspaces, Extensions, Delegates, and Frameworks

Building apps today isn't really about writing code. You may have learned how to write code in school or at a bootcamp intensive workshop, and those experiences are valuable ways to learn about the principles of coding. However, when you start your first coding job, you may find that you're asked to correct a typo in the title of a report that an existing app produces. It's a simple job that you can divide into two parts.

First, find where the typo is (a basic app can easily have many thousands of lines of code—Windows is estimated to have 50 million lines). It might not take long to find a typo in a single line of code, but how long does it take to find the line of code in the first place?

Second, fix the typo.

A month later, after you have finished the task of changing the title typo, you may find yourself actually building an app. That job, too, can be divided into several component parts.

J. Feiler, *Beginning Reactive Programming with Swift*,
https://doi.org/10.1007/978-1-4842-3621-5_1

First, implement a user authentication process. You can do this using the Facebook API or using some open source code from a trusted web source. You just have to find the code or API and then put it into your app.

Second, you need to implement your app's functionality that comes into play after the authentication process is complete. Depending on what the app is, you may have to write it from scratch, but chances are that you'll find yourself revising existing code from a similar project.

Third, you may take your new app and port it to a different platform.

Coding today is often about reading and understanding existing code and then reusing it in new apps and new combinations. Yes, there is a lot of from-scratch coding going on, but there's also a lot of reuse of existing code happening in the development world.

A number of factors have come together to create and support this world of reusable and repurposed code, which, after all, represents many, many hours of effort by many, many people. Reusing analysis and code is just as important as reusing and recycling natural resources. In the case of code, reuse means not reinventing the wheel. By not starting from scratch each time an app is created, the entire world of software development can move forward.

This chapter will provide an overview of how this world of reusable code functions—particularly from the vantage point of iOS, tvOS, macOS, and watchOS. You will find an overview of the reusable code building blocks, along with an overview of how you can put them together using Xcode and other tools that are part of the Apple developer's standard toolkit. There are three parts to the chapter:

- **Component Architecture Overview** gives you an idea of what it's like to build apps from components.
- **Looking at the iOS and macOS Building Blocks** provides an overview of what those blocks are.
- **Building with the Building Blocks** provides an overview of how to put them together.

Component Architecture Overview

Since the beginning of the computer age in the 1940s, there has been a development backlog of projects waiting to be done. (A companion backlog accompanied the rise of the web.) The need for software seemed unstoppable. Various strategies emerged, and *components* were a key part of many of them, both for the web and for software in general.

The idea was that building complete websites, programs, and apps from scratch was an unsustainable model. There had to be some way of speeding things up by reusing code that had already been written and debugged. The problem with this simple idea was that it wasn't possible to easily reuse code—changes always needed to be made.

One way of reusing code to speed up the development process was to take existing code and extract its key functions and features. These elements could be reused more easily than an entire code base. This was the beginning of component software development.

As time passed, these reusable extracts began to be used in two different ways:

- **Use a framework or shell.** In this model, there is a framework into which you can plug components.

  The framework model was popular in the 1990s; IBM's Software Object Model (SOM) was one of the first. Microsoft entered the component software world with Object Linking and Embedding (OLE) and Component Object Model (COM). A consortium of Apple, IBM, and WordPerfect worked on OpenDoc. All of these were frameworks into which you could plug specialized components (from the user's point of view, most were documents into which you could plug components).

- **Build a product from components.** In this model, you combine a number of reusable components (off-the-shelf or written specifically for the project) to make a single product. There usually isn't a framework or container as a shell; in some projects, there is indeed such an overarching container or shell, but it may be created specially for each project.

Regardless of the component model you're working with, there is a critical issue that crops up as soon as you start thinking about components: What language will you use? In today's world, the languages for iOS (and macOS) are Swift and Objective-C. However, one of the features of component architecture is that in some cases you can mix different languages, as you will see in the "Command-Line Integration" section later in this chapter.

Looking at the iOS and macOS Building Blocks

The building blocks in this section are all built in the Swift and Objective-C languages for iOS and macOS, and with APIs such as UIKit for iOS and AppKit for macOS, as well as their companions. This section will provide a brief overview of the building blocks; for more information, look on `developer.apple.com`.

Extensions

Extensions in Swift let you add functionality to an existing class, structure, enumeration, or protocol type. You can find an example in the *Adopting Drag and Drop in a Custom View* sample code on `developer.apple.com`. The drag-and-drop functionality is defined in protocols (see the following

section for more on protocols and delegates) and implemented in extensions.

In Figure 1-1, you can see an app that uses the code from the Adopting Drag and Drop in a Custom View sample. In this case, the basic class is a custom view controller (`PersonnelViewController`). There is an extension defined as follows:

```
extension PersonnelViewController: UIDragInteractionDelegate {
```

Each extension is in a file that references the base class (the class that is to be extended). As you can see in Figure 1-1, the names of those files are

```
PersonnelViewController+Drag
PersonnelViewController+Drop
```

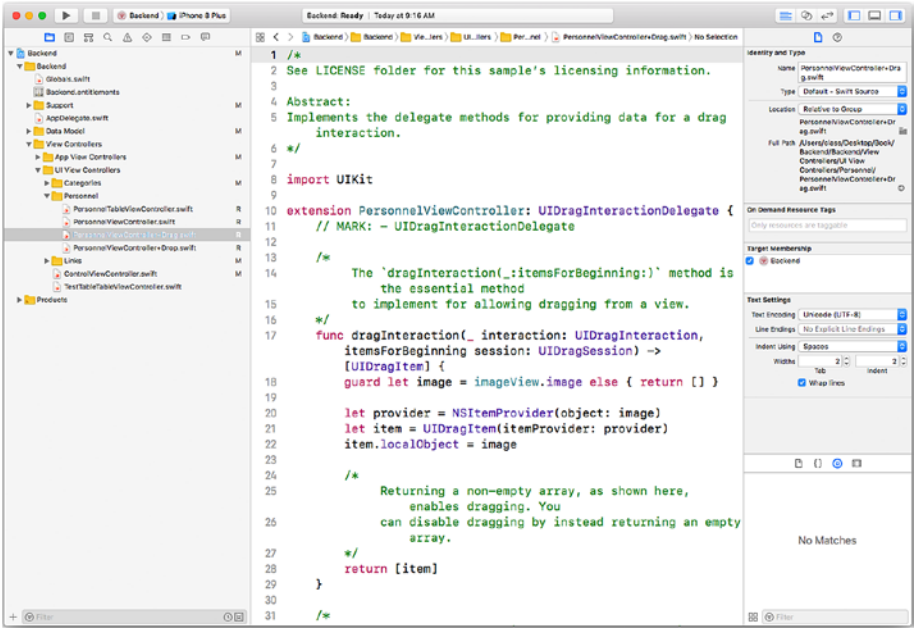

Figure 1-1. *Using extensions in a Swift class*

At runtime, you can reference the functions and other members of the extension just as you would reference elements of the class.

Extensions can be added to base classes and other structures for which you only have the API and not the source code.

Delegates and Protocols

Delegates and protocols work together. In the declaration of a class or other structure, you see the superclass (if any) in the declaration, as in the following declaration for a subclass of `UIDocument` in iOS:

```
class MyDocument: UIDocument {
```

A protocol can define functions that will be implemented in any class that conforms to the protocol. Whereas with an extension the extensions are added to the base class, with a protocol the protocol defines the code that *you* will add to the base class.

Delegates often work together with protocols so that the implementation of the protocol code is not placed into the base class; rather, it is placed in a separate file called a *delegate*. The specific file that implements the protocol is typically assigned to a field called *delegate* in the base class.

Frameworks

If you work with iOS or macOS a lot, you are probably familiar with the basic frameworks, such as AppKit (macOS) and UIKit (iOS), along with smaller frameworks such as AddressBook. Frameworks can contain functions and properties. You add them to a Swift project with an `import` statement; with Objective-C, you can use an #import or `#include` statement. (In Objective-C, the `#import` directive imports the framework once; `#include` may import it several times.)

Building with the Building Blocks

You can use delegates and protocols, extensions, and frameworks within an Xcode project. You can use a workspace to combine several projects, and you can use other tools to combine multiple components. Both workspaces and the combinations of building blocks will be described in this section.

Using a Workspace

With a workspace, Xcode takes care of managing the building of whichever target within the workspace you want to build. Targets may share elements from the workspace and will use them as needed to build various targets (such as for iOS and watchOS with the same workspace).

Building with Combinations of Building Blocks

The building blocks from Apple (frameworks, protocols, and delegates, as well as extensions) often provide a neat and elegant way to extend and expand your code. However, there are cases in which a single feature requires the use of multiple building blocks—for example, a feature might need one very big framework to be installed, along with a dozen or more smaller (but related) frameworks. Protocols and delegates are now commonplace in many structures, and extensions, likewise, may be added to the mix. Thus, implementing a new feature using shared code may require many additions to your code base.

Situations in which multiple building blocks need to be added to an app are common, and they can be difficult to manage. There are several tools available to help you manage such combinations. These tools use a structure that organizes the changes to your app so that a script or other tool can apply the changes in the right places and in the right order.

One of the most widely used of those tools is CocoaPods, which is the topic of Chapter 2.

GitHub has become the most widely used code-sharing tool and site today, and it is integrated with most package managers. Thus, the download of the latest GitHub version of the complex building blocks is done for you automatically as you run the package manager.

Package managers like CocoaPods use their own code and scripts to perform the integration. To do so, they—and you—must use some command-line code. If you are used to macOS and the Finder, you may not use the command line very often. Don't worry—the products hide most of that syntax from you. However, for the cases in which you do need to access a file or folder from the command line, the following section will provide some tips.

Command-Line Integration

Terminal, which is automatically installed as part of macOS, is the app that gives you access to the command line. When you launch it, you will see the basic screen shown in Figure 1-2.

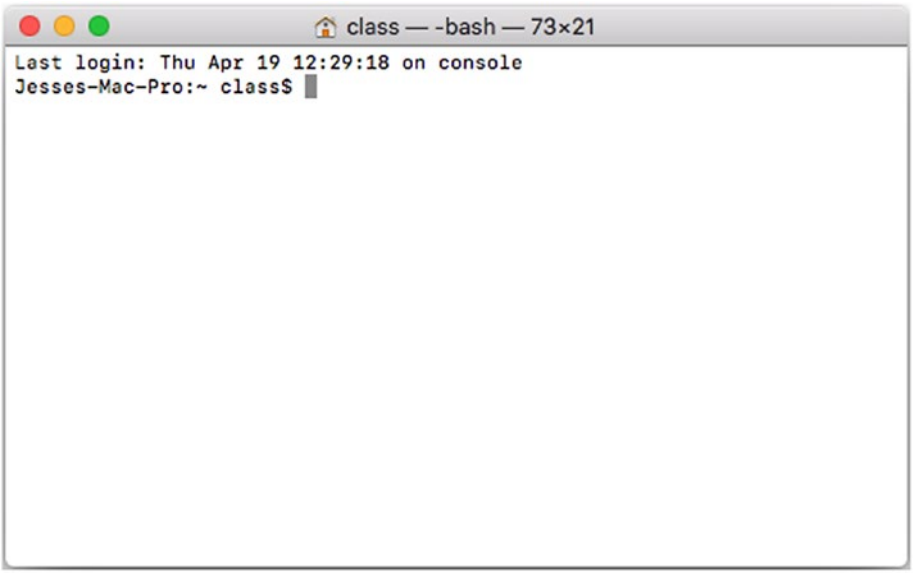

Figure 1-2. *Use Terminal to access the command line*

The first line shows you the date and time of the last login. On the second line, you can see the name of the computer you are using. You then see the identifier of the user you are running, and a symbol such as $ marks the end of the automatically generated text. You type your command after that.

Note You can customize the formatting of lines in Terminal.

In Figure 1-3, you can see the first command entered into Terminal. It is the list command (the code is `ls`).