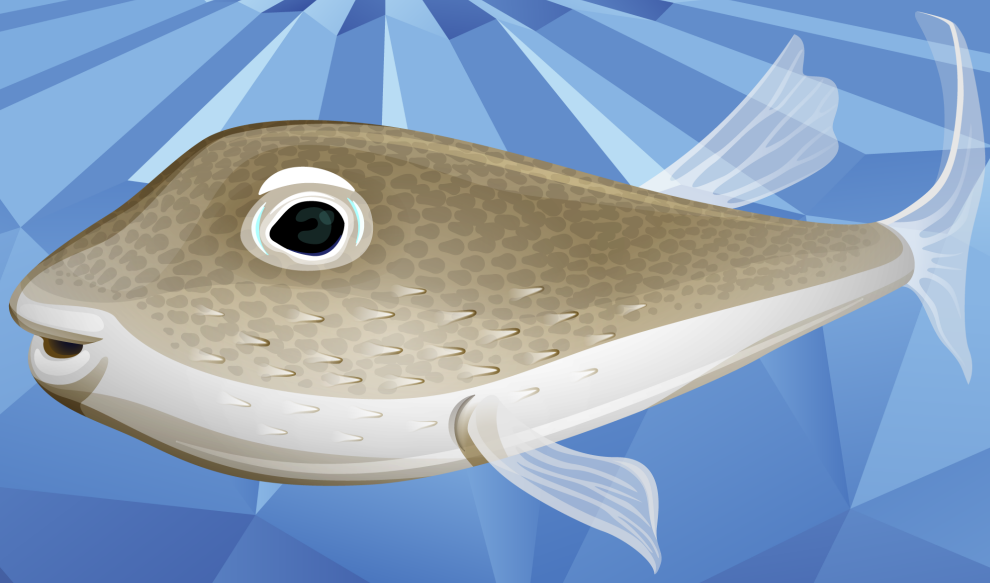
Up to date for iOS 13, Xcode 11 & Swift 5.1



Advanced iOS App Architecture

SECOND EDITION

Real-World App Architecture in Swift

By the raywenderlich Tutorial Team

René Cacheaux & Josh Berlin

Table of Contents: Overview

Table of Contents: Extended

Now, more than ever, it's very important to understand and apply good software architecture practices in our projects as apps are getting more complex and as development teams are pressured to deliver faster results despite constantly changing requirements.

In the book *Advanced iOS App Architecture* you'll learn about how to apply popular iOS app architectures to your apps. The book covers a number of different achitectures, including MVVM and Redux.

This book sample contains an adridged extract of Chapter 6 - Architecture: Redux. This explains the concepts and theory behind Redux and explains its advantages and disadvantages.

We hope that this hands-on look inside the book will give you a good idea of what's available in the full version and that it helps you to choose the best architecture for your apps going forward.

The full book is available for purchase at:

- https://store.raywenderlich.com/products/advanced-ios-app-architecture

Enjoy!

--René, Josh, and the *Advanced iOS App Architecture* team

Advanced iOS App Architecture

By René Cacheaux & Josh Berlin

Notice of Rights

Notice of Liability

Trademarks

About the Authors

Josh Berlin is an author of this book. He loves building thoughtful user experiences on mobile. He's currently an iOS engineer at Cruise Automation making apps for self-driving cars. He's built apps for the iPhone and iPad since 2008. Josh recently finished culinary school in Austin, TX. When he's not coding, he's probably cooking or dreaming of food.

René Cacheaux is an author of this book. He loves to architect and build software. He currently is a Mobile Architect at Atlassian where his mission is to design Atlassian's mobile platform. He especially loves all things mobile and currently architects for both Android and Apple platforms. René has been engineering iOS apps since 2009 and has experience in mobile client and server engineering, mobile user experience design and product management. René has worked on a wide range of apps spanning from industrial sales enablement to world-wide social networking. René enjoys starting his days in true Austin-Texas fashion with a a breakfast taco alongside a freshly brewed cappuccino. In addition to building mobile apps, he loves to travel, snow ski, ocean kayak and root for his alma mater, the Texas Longhorns.

About the Editors

Darren Ferguson is doubling as the tech editor and the final pass editor for this book. He's an experienced software developer and works for M.C. Dean, Inc, a systems integration provider from North Virginia. When he's not coding, you'll find him enjoying EPL Football, traveling as much as possible and spending time with his wife and daughter. Find Darren on Twitter at @darren102.

About the Artist

Vicki Wenderlich is the designer and artist of the cover of this book. She is Ray's wife and business partner. She is a digital artist who creates illustrations, game art and a lot of other art or design work for the tutorials and books on raywenderlich.com. When she's not making art, she loves hiking, a good glass of wine and attempting to create the perfect cheese plate.

Book License

By purchasing *Advanced iOS App Architecture*, you have the following license:

- You are allowed to use and/or modify the source code in *Advanced iOS App Architecture* in as many apps as you want, with no attribution required.
- You are allowed to use and/or modify all art, images and designs that are included in *Advanced iOS App Architecture* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Advanced iOS App Architecture*, available at www.raywenderlich.com".
- The source code included in *Advanced iOS App Architecturer* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Advanced iOS App Architecture* without prior authorization.
- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

What You Need

To follow along with this book, you'll need the following:

- A Mac running **macOS Catalina** or later.
- **Swift 5.2**: all projects have been written to work with Swift 5.2 in Xcode.
- **Xcode 11.4 or later.** You'll need Xcode 11.4 or later to open and run the example apps included in this book.

If you haven't installed the latest version of macOS or Xcode, be sure to do that before continuing with the book. The code covered in this book depends on Swift 5.2 and Xcode 11.4.

This book provides the building blocks for developers who wish to broaden their horizons and learn how architectures can help them build robust and maintainable applications and SDKs.

The only prerequisites for this book are an intermediate understanding of Swift and iOS development. If you've worked through our classic beginner books — *Swift Apprentice* https://store.raywenderlich.com/products/swift-apprentice and *iOS Apprentice* https://store.raywenderlich.com/products/ios-apprentice — or have similar development experience, you're ready to read this book.

To get the most out of this book an understanding of Apple's new **Combine** framework would also be helpful. If you've worked through our - *Combine Asynchronous Programming with Swift* https://store.raywenderlich.com/products/combine-asynchronous-programming-with-swift book you'll understand more regarding the reactive programming used within this book.

As you work through the book, you'll be taken through a deep dive into different

architectures for a fictional app named **Koober**. Each chapter will explain the theory behind each of the architectures first. The second half of the chapters will guide you through how the **Koober** application utilized the architecture and show you how the architecture was used within the application.

Chapter 6: Architecture: Redux

By Josh Berlin & René Cacheaux

History

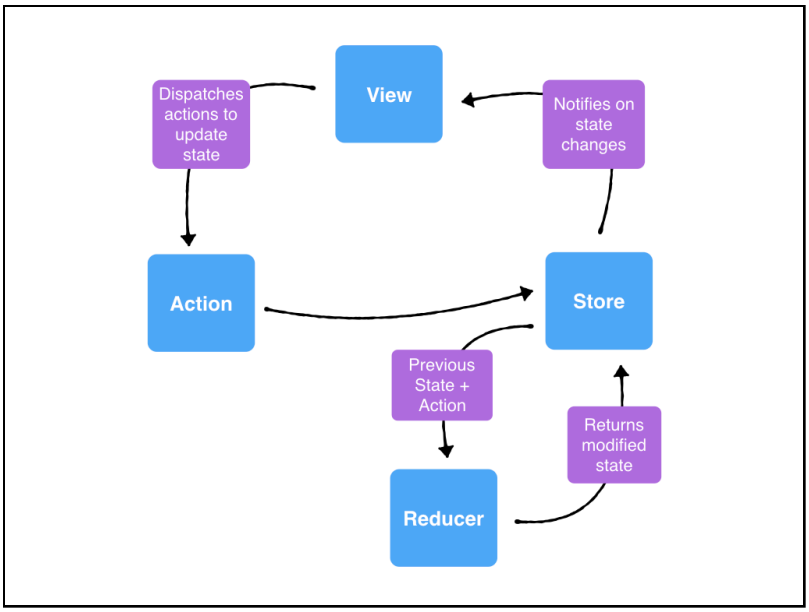
At Facebook, some years ago, a bug in the desktop web app sparked a new architecture. The app presented the unread count of messages from Messenger in several views at once, not always presenting the same amount of unread messages. This could get out of sync and report different numbers, so the app looked broken. Facebook needed a way to guarantee data consistency and, out of this problem, a new unidirectional architecture was born — Flux.

After Facebook moved to a Flux based architecture, views that showed the unread message count got data from the same container. This new architecture fixed a lot of these kinds of bugs.

Flux is a pattern, though, not a framework. In 2015, Dan Abramov and Andrew Clark created Redux as a JavaScript implementation of a Flux inspired architecture. Since then, others have created Redux implementations in languages such as Swift and Kotlin.

What is Redux?

Redux is an architecture in which all of your app's state lives in one container. The only way to change state is to create a new state based on the current state and a requested change.



The **Store** holds all of your app's state.

An **Action** is immutable data that describes a state change.

A **Reducer** changes the app's state using the current state and an action.

Let's go through each component in depth.

Store

The Redux store contains all state that drives the app's user interface. Think of the store as a living snapshot of your app. Anytime its state changes, the user interface updates to reflect the new state.

You might think storing everything in one place is insane — that's a valid thought. Instead of creating one massive file for the state, split it up into different sub-states. Each screen cares about a part of the entire apps state, anyway. We'll talk more about keeping the store organized in the example code section of this chapter.

Types of state

A Store contains data that represents an app's user interface (UI). Here are some examples:

- **View state** determines which user elements to show, hide, enable, disable or

whether a spinner is animating.

- **Navigation state** determines which view to present to the user and which views are currently presented.
- **High-level state** determines whether the user is signed in or signed out. Current user profile metadata and authentication tokens could be contained in the high-level state.
- **Data from web services** include things like responses from a REST API. The response gets parsed into models and placed in the store. In Koober, the available ride options displayed on the map live in the store.
- **Formatted strings** are strings that get transformed for display from raw model data from an API.

The store is the source of truth for your app. All views get data from the same store, so there's no chance of two views displaying different data, as was happening during Facebook's bug.

Derived values

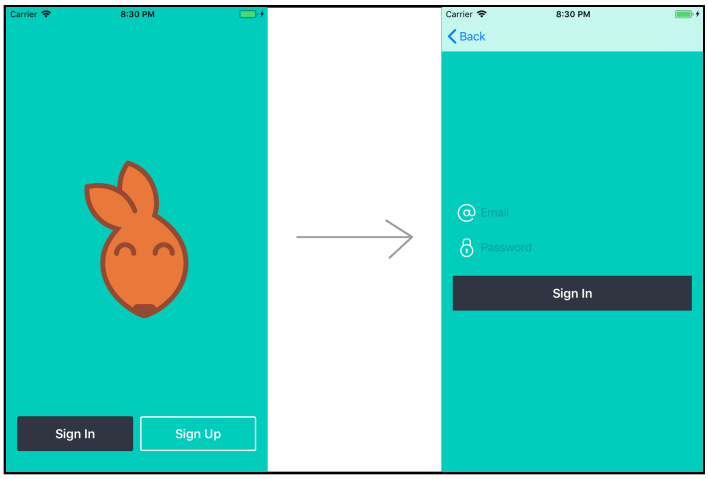
The store doesn't contain larger files, such as images or videos. Instead, it contains file URLs pointing to media on disk.

The entire store is in memory at all times. If your app has tons of video files or images in the store instead of file references, iOS may crash your app to free up memory.

Modeling view state

In a Redux architecture, views store no state. They only listen and react to state changes. So, any state that changes how the view behaves lives in the store. Stores consist of immutable value types. When data in the store changes, views get a new, immutable state to re-render the user interface.

To help illustrate how this works, let's take a look at an example for a fictional ride-sharing app for kangaroos, called Koober:



The onboarding for Koober contains a welcome screen where you can navigate to the sign-in or sign-up screens. The app state determines which screen is currently shown to the user. When the app state changes, the app presents a new screen to the user.

The `Onboarding` state shows the unauthenticated screens before the user logs in. The `Signed In` state shows the authenticated screens after the user logs in.

The Onboarding state contains three states:

1. Welcoming
2. Signing In
3. Signing Up

`Welcoming` displays the welcome screen, which has Sign In and Sign Up buttons. When you tap the Sign In button, you set the app state to `Signing In`. When you tap the Sign Up button, you set the app state to `Signing Up`.

At any moment, you can look at the state of the Redux store to determine what screen the user interface is presenting.

Loading and rendering initial state

Koober has two high-level app states:

- **Launching** loads any data that the app needs to function, like a previous user session.
- **Running** displays either the onboarding flow or the map screen.

The **Running** state has two sub-states:

- **Onboarding** displays the sign-in or sign-up screen so that the user can authenticate.
- **Signed In** displays the map and needs a valid user session.

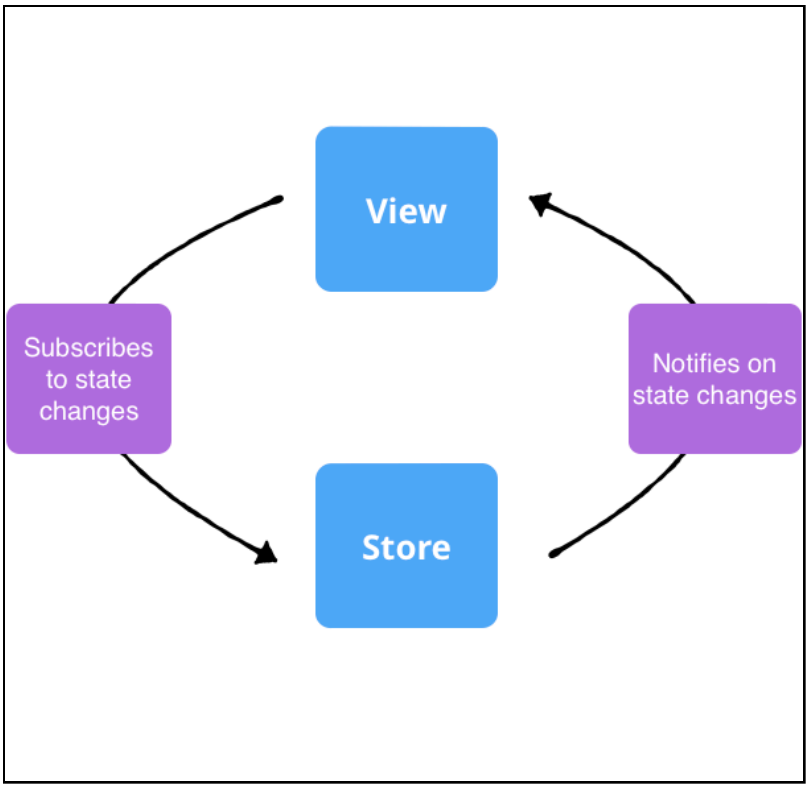
Koober starts in the “launching” state.

Once the “launching” state reads the user session, the app transitions to the ”running” state. Then, the user interface displays either the onboarding flow or the signed-in screens.

The Redux store always has an initial state and never has an invalid state. Redux forces you to declare every possible state for your app.

If you don‘t persist data between launches, that data must have an initial default in the store. For example, Koober has ride options you can choose before requesting a ride: Wallaby, Wallaroo and Kangaroo. These values can change, so they come from the server. Before you download them, the initial state in the Redux store is an empty array. The user interface should be able to gracefully handle this empty state.

Subscription



For a view to render, it subscribes to changes in the store. Each time state changes, the view gets wholesale changes containing the entire state — there is no middle ground. This is unlike MVVM, where you manipulate one property at a time.

Using **Focused Observation**, the view can subscribe to pieces of state that it's interested in, avoiding updates when any app state changes occur. The view still gets the piece of state in one update.

Views need the current state from the store each time the view loads. On load, they always have an empty state. After subscribing to the store, it fires an update and the view re-renders.

There's a short delay between when the app presents the view on screen and when the subscription fires its first update. The duration is usually short enough where you don't notice the first update. But make sure all views can gracefully display an empty state.

Responding to user interactions

Actions are immutable data that describe a state change. You create actions when the user interacts with the user interface.

Dispatching an action is the only way to change state in the Redux store. No sneaky view can grab the store and make changes without the rest of the app finding out. Redux works because actions change the store, and it notifies subscribers across the app.

For example, in the welcome screen, there are two buttons, Sign In and Sign Up. When the Sign Up button gets tapped, you create and dispatch a Go to Sign Up action. The store updates its state, and it notifies the `OnboardingViewController`. Then, the `OnboardingViewController` pushes the sign-up screen onto the navigation stack.

Reducers describe possible state changes. Reducers are the step between dispatching an action and changing the store's state. After an action is dispatched, it travels through a reducer. The *only* place the store's state can mutate is in a reducer. Reducers are free functions that take in the current store's state along with an action describing a state. They mutate a copy of the current state based on the action, and return the new state. Reducer functions should not introduce side effects. They should not make API calls or modify objects outside of their scope.

In addition to updating state based on actions, reducers can run business logic to transform state. Date formatting logic lives in reducers to transform data for display. For example, a reducer can transform a `Date` object to a presentable `String`.

Koober contains a lot of logic in reducers. It's already enough trouble keeping view controllers small. The last thing you need is a massive reducer file. Redux recommends to split your reducers into sub-reducers. Sub-reducers help keep your reducer logic focused and readable. Koober has sub-reducers for the onboarding flow, the sign-up screen, the sign-in screen and so on.

Threading

In Redux, it's important to run all the reducers on the same thread. It doesn't have to be the main thread, but the same serial queue.

If you run the reducers on multiple threads, the input state of the reducer could change while it's running on another thread. Redux is a synchronization point by design.

ReSwift, a Redux implementation of unidirectional data flow architecture in Swift, lets you run reducers on any serial queue, but defaults to the main queue. The simplest approach is to run on the main queue because the user interface and store

are completely in sync. Then, there's no need to hop on main queue when observing the store.

> **Note**: In a complex app, reducers might take some time. In this case, it can be a good idea to run reducers on another serial queue that's not the main queue. Most of the time, the main queue is fine.

Performing side effects

Side effects are any non-pure functions. Any time you call a function that can return a different value given the same inputs is a side effect. Pure functions are deterministic. Given the same inputs, the function always has the same outputs.

Reducers should be pure functions, free of side effects. In Redux, you handle side effects before dispatching actions and after the store updates.

For example, apps commonly make asynchronous API calls to a server and wait for a response. In Redux, you never make these asynchronous API calls in reducer functions. Instead, create multiple actions for different stages of your network request.

Stages of a network request:

1. Network request is in progress.
2. Network request completed successfully.
3. Network request failed.

Before starting the network request, dispatch an In-progress action. The reducer updates the state in the store to indicate the network request is in-progress. The view updates its user interface to reflect the change by showing a spinner and disabling UI elements as needed.

Next, make the network request. Once the API call completes, dispatch a Network Request Succeeded or Network Request Failed action. The store updates its state, and the view updates to show a success or failed message, and enables its UI elements. You can also dispatch actions during the network requests to update percentage complete state in the store.

A network request is one example of an asynchronous operation, and any asynchronous task can follow the same process: dispatch actions before, during and after the task completes.

Rendering updates

Redux is a “reactive” architecture. The word “reactive” is thrown around a lot these days. In Redux, “reactive” means the view receives updated state via subscriptions, and it “reacts” to the updates. Views never ask the store for the current state; they only update when the store fires an update that data changed.

Diffing

Each time a view receives new state via subscription, it gets the whole state. The view needs to figure out what changed and then properly render the update.

The simple solution is to reload the entire UI, although this might look clunky. Another solution is to diff the new state with the current state of the UI and render necessary updates. Diffing helps avoid unnecessary changes. It also allows views to animate changes, since you know exactly which user interface element changed.

UIKit sometimes won‘t render unnecessary changes. You can test this by subclassing a `UIView`, set a property to some test value, and check if the system calls draw rect or needs display.

That covers the main points of how the Redux architecture works - but before we continue, let's review the key points and discuss the pros and cons of using this architecture in the first place.

Key points

- Redux architecture keeps all your app‘s state in a single **store**.
- An **action** describes a state change. The only way to change state is to dispatch an **action** to the store.
- **Reducers** are pure functions that take an action and the current state, and they return a modified state. The only place the state can change is in a reducer function.

Advantages and Disadvantages

Advantages

1. Redux scales well as your application grows — if you follow best practices. Separate your Redux store state into sub-states and only observe partial state in your view controllers.
2. Descriptive state changes are all contained in reducers. Any developer can read through your reducer functions to understand all state changes in the app.
3. The store is the single source of truth for your entire app. If data changes in the store, the change propagates to all subscribers.
4. Data consistency across screens is good for iPad apps and other apps that display the same data in multiple places at the same time.
5. Reducers are pure functions — they are easy to test.
6. Redux architecture, overall, is easy to test. You can create a test case by putting the app in any app state you want, dispatch an action and test that the state changed correctly.
7. Redux can help with state restoration by initializing the store with persisted state.
8. It's easy to observe what's going on in your app because all the state is centralized to the store. You can easily record state changes for debugging.
9. Redux is lightweight and a relatively simple high-level concept.
10. Redux helps separate side effects from business logic.
11. Redux embraces value types. State can't change from underneath you.

Disadvantages

1. You need to touch multiple files to add new functionality.
2. Requires a third-party library, but the library is very small.
3. Model layer knows about the view hierarchy and is sensitive to user-interface changes.
4. Redux can use more memory than other architectures since the store is always in memory.
5. You need to be careful with performance because of possible frequent deep copies of the app state struct.
6. Dispatching actions can result in infinite loops if you dispatch actions in

response to state changes.

7. Data modeling is hard. Benefits of Redux depend on having a good data model.
8. It is designed to work with a declarative user interface framework like React. This can be awkward to apply to UIKit because UIKit is imperative. This isn't a blocker, just that it's not a natural fit.
9. Since the entire app state is centralized, it's possible to have reducers that depend on each other. That removes modularity and encapsulation of a model / screen / component's state. So refactoring a component's state type could cause compiler issue elsewhere and this is not good. You won't run into this if you organize your reducers to only know about a module's state and no more. This is not constrained by the architecture, though, so it depends on everyone being aware.

Where to go from here?

This sample chapter from *Advanced iOS App Architecture* ends here.

However, in the full version of the book, the chapter continues by:

- Reviewing this theory by walking through several examples.
- Applying the theory to a production-level sample app and walks through the code in detail.
- Explains how to use ReSwift, a popular Swift Redux implementaiton.
- Explains how to deal with real-life situations such as dealing with app state, user inateraction, equatable state models, RxSwift, user session persistence, communication across views, and more.

Where to Go From Here?

We hope you enjoyed this sample of *Advanced iOS App Architecture*!

If you enjoyed this sample, be sure to check out the full book, which contains the following chapters:

1. **Welcome**: A quick introduction over what you'll be learning this book, what the goals are of this work, and what you'll need to get started.
2. **Navigating Architecture Topics**: There are a lot of aspects to consider when selecting an architecture for you and your team. This chapter provides a high-level overview of these aspects, such as qualities to look for in an architecture and more.
3. **Example App**: In this book you'll use a fully fledged real world example application each written in the specific architectures this book will cover. The app used through this book is known as **Koober**. A sample of the application for each architecture accompanies each architecture chapter so you can see the differences between the chapters.
4. **Objects & Their Dependencies**: We all depend on one thing or another in the real world and when architecting your applications this is no different. You'll learn how objects depend on other object to avoid having large objects doing a lot of things instead of comparmentializing to make them more reusable and testable.
5. **Architecture: MVVM**: In this chapter, you'll be taken through the history of **MMVM** and its concepts. You'll then walk through the **Koober** application and explore how its been architected using the **MVVM** architecture approach.

6. **Architecture: Redux**: In this chapter, you'll be taken through the history of Redux and its concepts. You'll then walk through the Koober application and explore how its been architected using the Redux architecture approach.
7. **Architecture: Elements, Part 1**: Elements is an architecture meant to make iOS development fun and flexible. Elements organizes your codebase and makes your project easy for anyone to navigate. This organization allows you to make changes to layers of your application without affecting stability. A set of "Elements" make up the architecture.
8. **Architecture: Elements, Part 2**: In Part 1 you learned about Elements and how to design User Interface and Interaction Responder elements. In this chapter, you'll take a deep dive into two more elements: **Observer** and **Use Case**.
9. **Getting Ready for SwiftUI**: In this chapter, you'll be given an introduction to SwiftUI which Apple have released at WWDC 2019. You'll then walk through how to get Koober ready for SwiftUI while keeping your current architectural approaches.

You can find the book on the raywenderlich.com store here: https://store.raywenderlich.com/products/advanced-ios-app-architecture

We hope you enjoy the book!

--René, Josh, and the *Advanced iOS App Architecture* team

Apply Different Architectures to Your Codebase!

Advanced iOS App Architecture guides you through building one real-world app written in different architectures to give you hands-on and practical experience working in different architectures. This book will also guide you through the theory you need to gain a solid foundation of architecture concepts so that you can make your own informed decisions on how to use them in your codebase.

Who This Book Is For

This book is for intermediate iOS developers who already know the basics of iOS and are looking to build apps using defined architectures, making apps cleaner and easier to maintain.

Topics Covered in Advanced iOS App Architecture:

- **Navigating Architecture Topics:** Learn the theory behind various architectures to help inform which works best for you in different situations you may face.
- **Managing Dependencies:** Learn how to manage dependencies both internally and externally within your app.
- **MVVM Architecture:** Explore the history of the MVVM architecture and begin building KOOBER — the book's project app — using MVVM principles.
- **Redux Architecture:** Explore the history of the Redux architecture and continue building KOOBER using Redux principles.
- **Elements Architecture:** Explore the history of the Elements architecture and continue building KOOBER using Elements principles.

After reading this book, you'll have the knowledge to decide which types of architecture components suit your apps and you'll have a deep understanding of the covered architectures.

About the iOS Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website **raywenderlich.com.** We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.