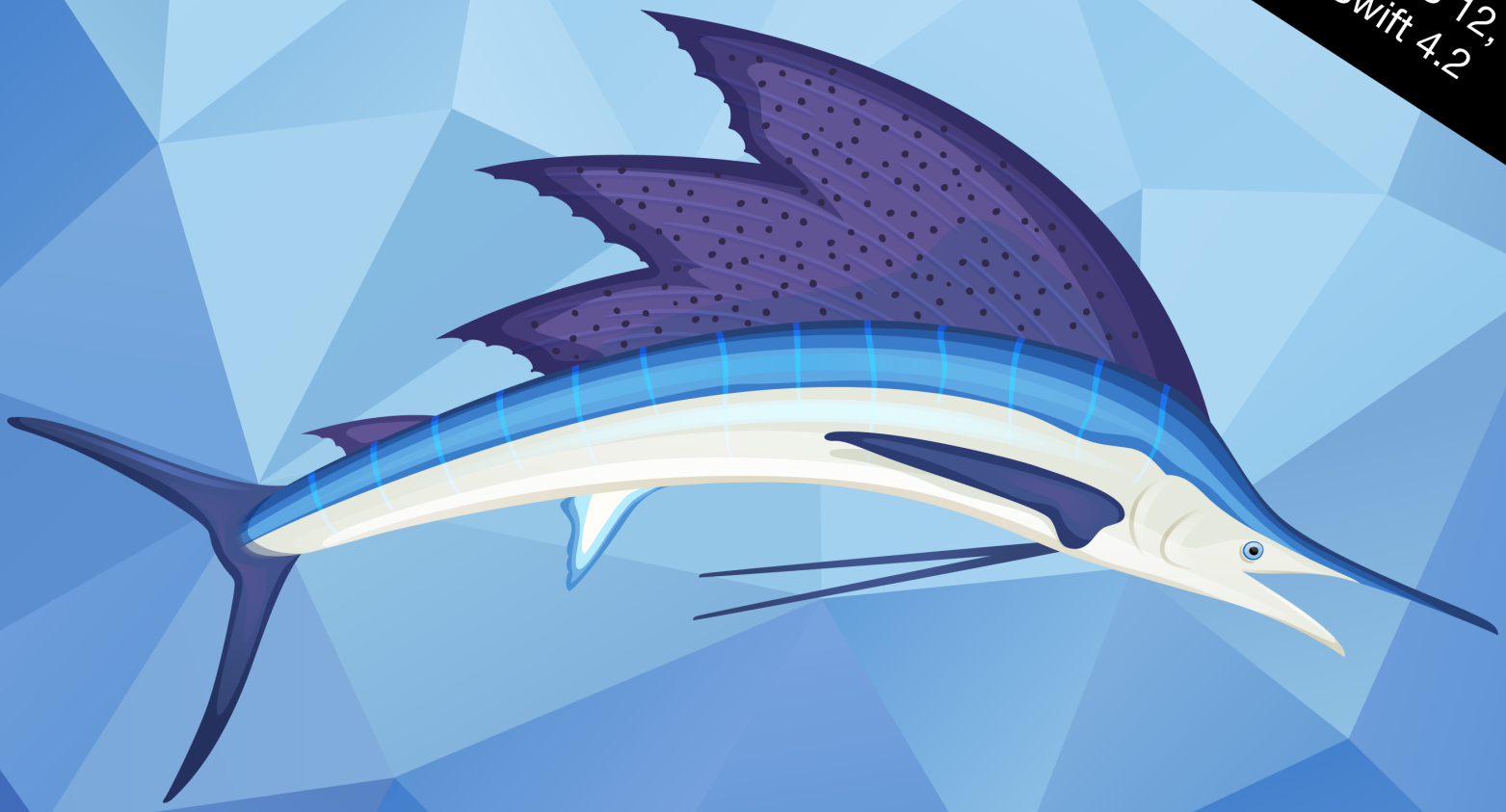
Up to date for iOS 12, Xcode 10 & Swift 4.2



Realm

Building Modern Swift Apps with Realm Database

SECOND EDITION

By Marin **Todorov**

Table of Contents: Overview

Table of Contents: Extended

About This Book Sample

Realm is a database that sports a custom-made engine which gets you started with a single line of code and lets you read and write objects as if you just kept them around in memory. With the RealmSwift API, you use native objects and can push the language to its limits as you please.

The *Realm: Building Modern Swift Apps with Realm Database* book is a thorough introduction to Realm. It goes over each topic in detail and works through some practical examples that will give you enough experience to tackle real-life app development problems.

This book sample contains the first two chapters:

- **Chapter 1: Hello, Realm!** introduces you to the concepts behind Realm and explains how to download and install Realm and some useful tools you'll use throughout the book.
- **Chapter 2: Your First Realm App** shows you how to build a simple To-do app that utilizes the basic Create, Read, Update and Delete (CRUD) operations you'll want to use in most applications.

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 show you why Realm is an exciting proposition for any app that needs to persist data. The full book is available for purchase at:

- https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database

Enjoy!

— Marin and the *Realm: Building Modern Swift Apps with Realm Database* team

Realm: Building Modern Swift Apps with Realm Database

By Marin Todorov

Notice of Rights

Notice of Liability

Trademarks

About the Author

Marin Todorov is the author of this book. Marin is one of the founding members of the raywenderlich.com team and has worked on seven of the team's books. He's an independant contractor and has worked for clients like Roche, Realm, and Apple. Besides crafting code, Marin also enjoys blogging, teaching and speaking at conferences. He happily open-sources code. You can find out more about Marin at www.underplot.com.

About the Editors

Shai Mishali is the technical editor of this book. He's the iOS Lead for the Tim Hortons mobile app and is involved in several open source projects in his spare time — mainly the RxSwiftCommunity and RxSwift projects. As an avid enthusiast of hackathons, Shai took 1st place at BattleHack Tel-Aviv 2014, BattleHack World Finals San Jose 2014 and Ford's Developer Challenge Tel-Aviv 2015. You can find him on GitHub and Twitter @freak4pc.

Tammy Coron is the editor of this book. She is an independent creative professional and the host of Roundabout: Creative Chaos. For more information, visit tammycoron.com.

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 *Realm: Building Modern Swift Apps with Realm Database*, you have the following license:

- You are allowed to use and/or modify the source code in *Realm: Building Modern Swift Apps with Realm Database* 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 *Realm: Building Modern Swift Apps with Realm Database* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Realm: Building Modern Swift Apps with Realm Database*, available at www.raywenderlich.com".
- The source code included in *Realm: Building Modern Swift Apps with Realm Database* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Realm: Building Modern Swift Apps with Realm Database* 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 sample book, you'll need the following:

- **Xcode 10 or later.** Xcode is the main development tool for writing code in Swift. At a minimum, you need Xcode 10 because it includes Swift 4.2. You can download the latest version of Xcode for free from the Mac App Store, here: apple.co/1FLn51R.

If you haven't installed the latest version of Xcode, please do so before continuing with the book. The code covered in this book depends on Swift 4.2 and Xcode 10 — you may get lost if you try to use an older version.

Chapter 1: Hello, Realm!

This book aims to introduce you, the reader, to the Realm database and how to create powerful, reactive iOS apps with Swift and RealmSwift.

Realm finds the sweet spot between the simplicity of storing data as JSON on disk and using heavy, slow ORMs like Core Data or similar that are built on top of SQLite. The Realm Database aims to be fast and performant, and to provide the commodities that mobile developers need such as working with objects, type-safety and native notifications.

In this book, you are going to learn *plenty* about Realm and how to build iOS apps with Realm; you'll also pick up some tips and tricks along the way about getting the most out of the platform.

Before getting to the code though, it's important to know what Realm *is*, and what you can expect from it — and what not to expect.

Currently, Realm offers two products:

- **Realm Database**: A free and open-source object database.
- **Realm Platform**: A syncing server solution, which can be either self-hosted on your servers or used as a cloud service via **Realm Cloud**.

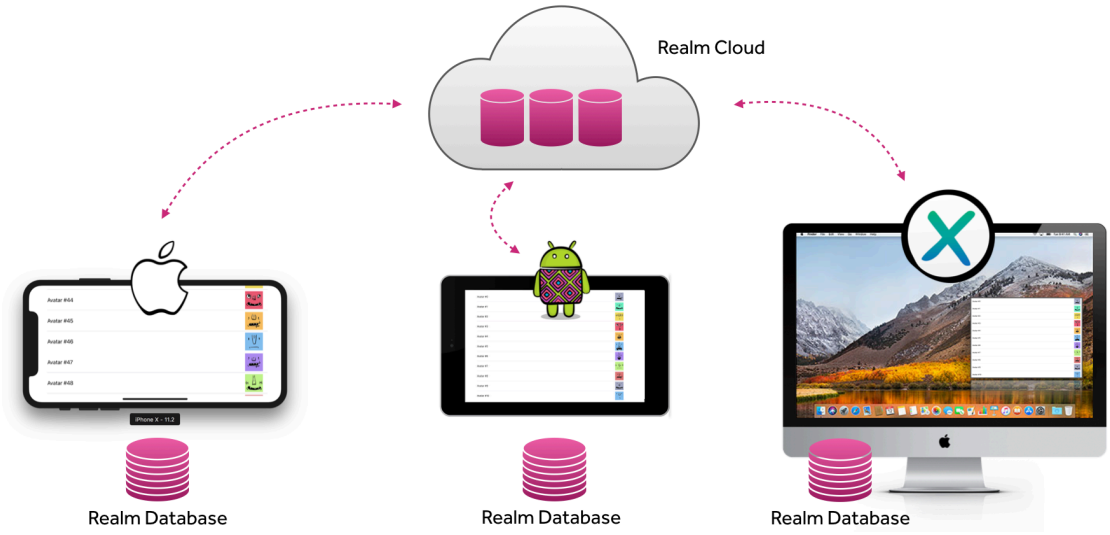
The primary focus of this book is the free database product (the **Realm Database**), which has been under active development for several years. It powers apps by some of the biggest names in the App Store, including Adidas, Amazon, Nike, Starbucks, BBC, GoPro, Virgin, Cisco, Groupon and many more who have chosen to develop their mobile apps with Realm.

The database has been widely adopted due to its ease of use and the fact that it can be used on both iOS and Android, unlike other database solutions:

Realm Platform, on the other hand, is a relatively new commercial product that allows developers to automatically synchronize data not only across Apple devices but also between any combination of Android, iPhone, Windows or macOS apps.

Realm Platform allows you to run the server software on your own infrastructure and keep your data in-house, which most often suits large enterprises. Alternatively, you can use Realm Cloud, which runs a Platform for you. You start syncing data very quickly and only pay for what you use.

The good news is that, once you know how to use the Realm Database to build iOS apps, which is covered in this book, enabling data sync via a cloud service is a process that takes just a few additional steps on top of what you're already doing:



That being said, you can develop both personal and commercial projects without any restrictions using the Realm Database.

If you decide that you'd like to have the app data synchronized automatically to your cloud or across platforms to Android, watchOS, macOS, Windows or others, you can add that feature later on.

For the time being, you're going to master the Realm Database and then, in the last chapter of this book, you'll look into using the database with Realm Cloud.

Realm Database

Realm Database fills the gap in the field of client-side data persistence. Indeed, there have been a multitude of server products released in recent years, but not much has happened for client-side needs.

Up until a few years ago, the defacto standard for building mobile apps on both iOS and Android was SQLite: a fast, but generic, all-purpose SQL database format. SQLite has a number of virtues, such as being written in C, which makes it fast, portable to almost any platform, and highly conformant to the SQL standard.

Realm, on the other hand, is a modern database solution that focuses on the needs of modern apps. In that sense, it is not an all-purpose database. It is really good at reading and writing data extremely fast, but it requires you to master the Realm API, so you can't simply use your existing SQL database skills.

So, it's precisely *because* Realm is a new type of mobile database that you should not expect to use it the same way as you would an old, generic SQL database.

That being said… **good news, everyone**!

The Realm APIs, built with modern, best-practices code, are arguably *much* easier to use than working with the C API of SQLite from Swift.

In fact, even if you've never used Realm before, you can get started with the basics in mere minutes. After all, the code speaks for itself:

```
class Person: Object {
  dynamic var firstName: String?
  dynamic var lastName: String?
}

let me = Person()
me.firstName = "Marin"
me.lastName = "Todorov"

try realm.write {
  realm.add(me)
}
```

This Swift code defines a new `Person` class, creates an instance and sets its object properties, and finally tries to add the new object to a realm within a write transaction.

Have a look at how you get objects back from the database:

```
realm.objects(Person.self)
  .filter("age > 21")
  .sorted("age")
```

This Swift code asks Realm for all persisted objects of type `Person`, filters those to ones of age `21` or more, and finally sorts the results.

Sometimes Realm code almost looks *too easy* to be used in a solid, robust database API.

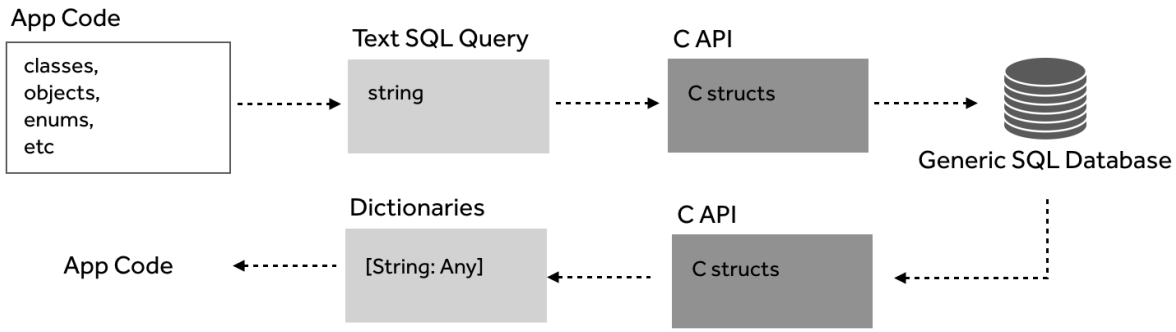
You might be thinking: "I already figured out how to write data… why do I need to work through the whole book then?" That's a fair question to ask!

The Realm basics are really easy to pick up; you can start using Realm in your apps after following a single tutorial. But, just like following a single tutorial on *any* topic, that gives you a very limited view into Realm's API and the specifics of building apps with Realm.

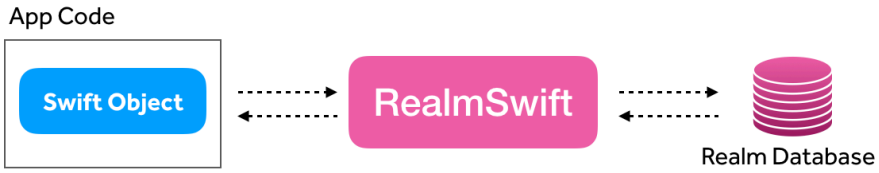
This book aims to give you some detailed knowledge of designing and building object schemas through several projects, and to give you enough experience to make informed decisions when creating your next app with Realm.

Live objects

One of the cornerstones of the Realm Database philosophy is that modern apps work with **objects**. Most persistent solutions will fetch data in the form of generic dictionaries that aren't type-safe and pollute the app code with excessive logic to cast and convert fetched data to a format usable in the app.



Realm, on the other hand, uses classes to define its data schema (e.g., what types of data can be persisted in the database) and the developer always works with objects, both when persisting data to a Realm and when querying the database for results.



That helps to simplify the app code when compared to persisting data other ways. You simply skip over boilerplate data-conversion code and always use native Swift objects.

The fact that your data is always contained by objects allows for a number of other Realm features; it introspects your classes and automatically detects any changes you make to the database schema; it updates fetched data to the latest snapshot from the disk; and much more.

Working with data objects directly and exclusively allows for the speed, robustness and type safety that Realm is known for across platforms. In fact, the Realm Database Core (open sourced on GitHub by Realm) is written in cross-platform C++ so it works exactly the same way on Android, iOS, macOS or any other platform.

But don't worry: You don't ever need to use C++ in your own apps, unless you really want to. Realm provides SDKs in different languages, which perform some lean wrapping of the database core engine to give developers APIs that best fit their app code.

And the best part of all is that since the C++ core is the same across platforms, the native APIs also behave similarly. That *really* helps the iOS and Android developers on a multi-disciplinary team stay in sync and work together.

The code isn't exactly the same across platforms: It's similar, but each language offers different features that Realm uses to make its APIs feel as native as possible. For example, this is how you would declare the interface of an object class in Objective-C:

```
@interface Person : RLMObject
@property NSString *firstName;
@property NSString *lastName;
@end
```

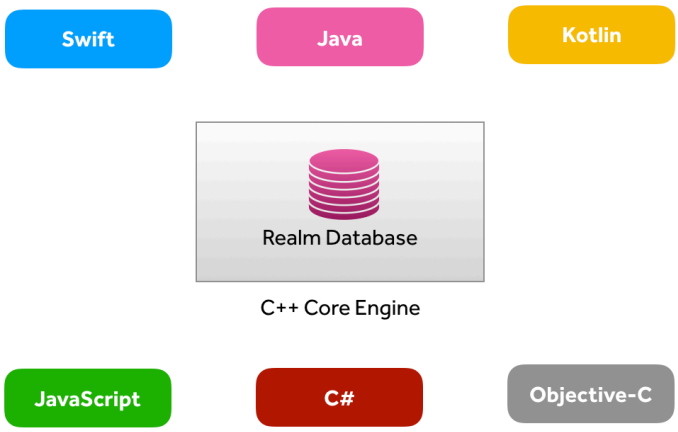
Defining the same class would look like this in Swift:

```
class Person: Object {
  dynamic var firstName: String?
  dynamic var lastName: String?
}
```

And, just for kicks, here's the code in Kotlin:

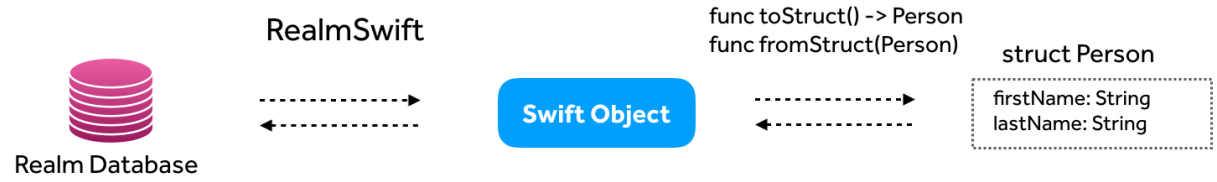
```
open class Person: RealmObject() {
  var firstName: String? = nil
  var lastName: String? = nil
}
```

You get to work with native, custom classes in any of the supported languages that you might be using to build your app, and Realm automatically read and writes the stored data to disk.



Not only does Realm allow you to use your own custom classes to persist data, but you can customize your own API in any way you'd like.

For example, if you are a big `struct` fan, you can simply add `toStruct()` and `fromStruct(_)` methods on your Swift data object class to be able to quickly read data as Swift structs from your database:



You'd be wrapping Swift objects into structs in a similar way to how Realm wraps C++ objects into Swift objects.

And since Realm data objects are fully fledged classes, you don't need to limit yourself to only using them for storage. They can include all kinds of additional logic that makes sense in the context of data persistence. Your classes might sport methods to create new instances, fetch instances from a given realm, and to convert data between custom formats that you need to store on disk.

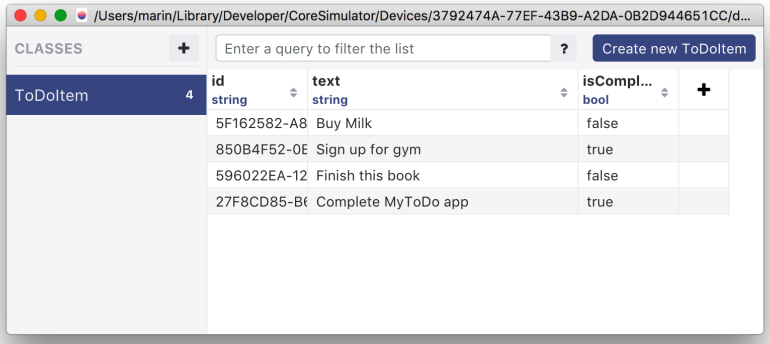
In any case, the Realm SDKs for each language try to give you the tools that you need and provide you with native-feeling APIs to make your life as easy as possible.

Realm Studio

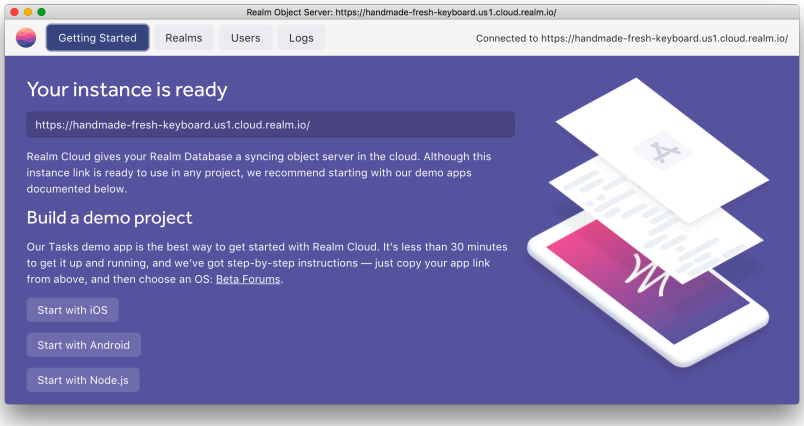
Another of Realm's benefits over other data persistence libraries is that it offers high-fidelity data browser and editor app.

Realm Studio (https://realm.io/products/realm-studio) lets you inspect the contents of your app's realm files and change any of the data manually, if necessary.

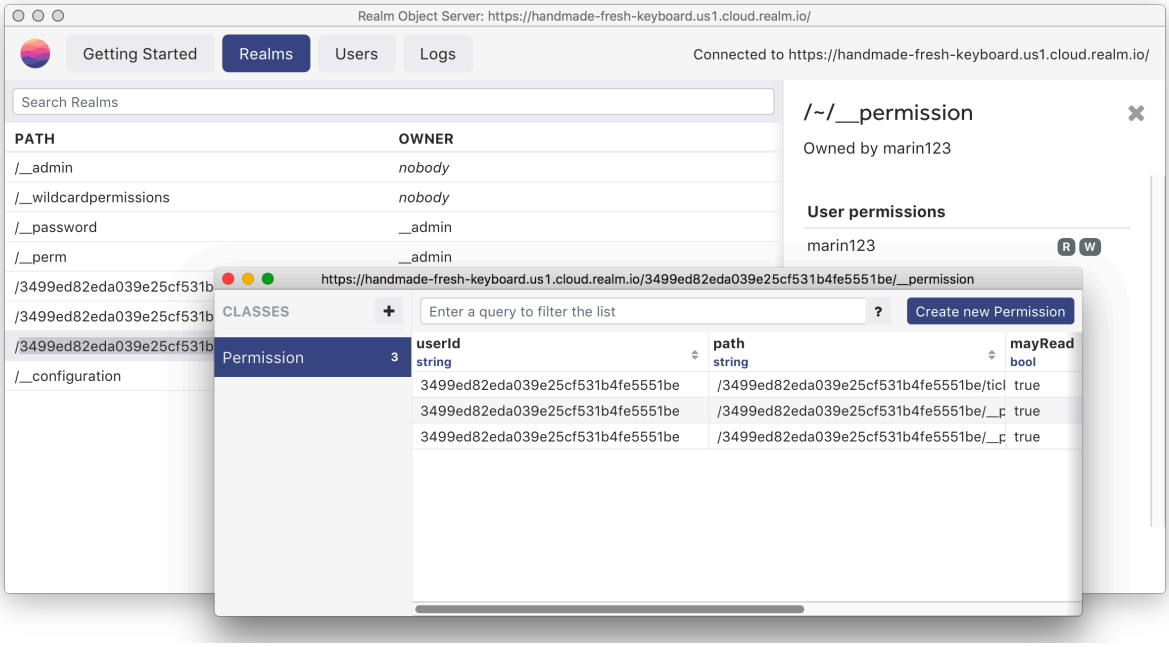
You will use Realm Studio throughout this book to inspect the changes your code commits to your Realm Database files. The app allows you to browse and modify all the objects stored in a file, and, additionally, to browse, query and modify the stored data in a spreadsheet-like manner.



In the final chapter of this book, you will use Realm Studio to connect to Realm Cloud. Realm Studio is a streamlined way to get started with the Realm Platform, including starter code for different platforms. At launch, Realm Studio will prompt you to browse your cloud instances or explore demo apps:



Once you're connected to a server in the cloud, you can browse and modify the data manually as you would with any local file:

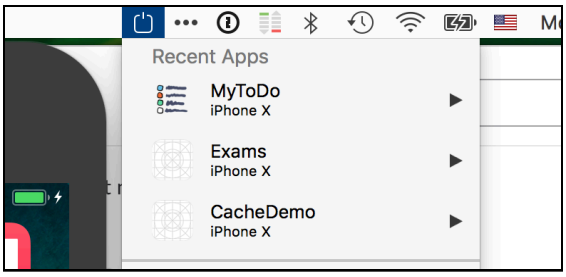


You will be using Realm Studio fairly often, so download and install the app for free from https://realm.io/products/realm-studio.

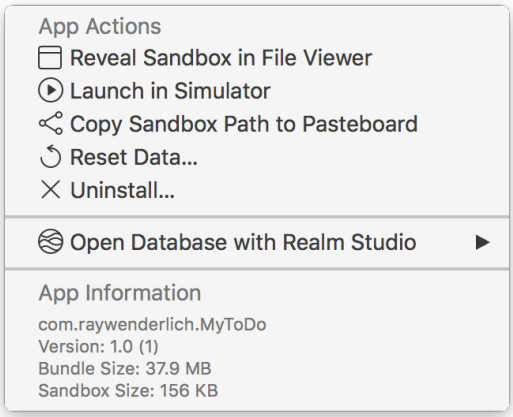
SimPholders

Speaking of amazing tools, you might want to consider purchasing a tool called SimPholders. It integrates very well both with your iPhone Simulator and with Realm Studio, and it allows you to inspect your app's database with a single click.

When you're working on apps in Xcode and testing them in your iPhone Simulator, SimPholders detects the activity and always gives you quick access to recently active apps across all your simulators.



When you select an app from the list above, you can quickly open folders of interest or launch Realm Studio directly:



SimPholders is a useful tool and will make your life developing apps with Realm considerably easier. You can get a 10-day trial or a (very affordable) license from the app's website:

- https://simpholders.com.

Realm or realm?

There are few different things being referred to as "realm" or "Realm" in this book (either lower or upper case) so let's clear the air right now before it becomes source for any confusion.

The name of the company that created the Realm database, the database itself, and a particular database file are all called the same. In this book I tried to make the text clearer by using a specific word casing, depending on what exactly I refer to.

Whenever the noun is uppercased as "Realm" that means I refer to the company or the database name, for example: "Realm is an avid open source software supporter" or "The Realm Database is lean and super-fast", but also "The Realm SDK offers completely safe transactions".

If you see a lowercased "realm" that refers to the data in a given file, for example: "Add two objects to your app's realm" or "You can have two different realm files: default.realm and additional.realm".

Installation

Once you are convinced that you need Realm in your app (and I hope you already are!), you need to add Realm as a dependency of your code. RealmSwift is available for free from https://github.com/realm/realm-cocoa and detailed documentation can be found, here: https://realm.io/docs/swift/latest.

Realm Objective-C and Realm Swift are published under the Apache 2.0 license. Realm Core is separately available and is also published under the Apache 2.0 license.

The easiest way to include Realm and RealmSwift in your projects is via CocoaPods or Carthage.

The projects in this book use **CocoaPods**. Even if you usually use a different dependency manager, please make sure to use CocoaPods while you work through the projects in this book.

Installing via CocoaPods

You install the Realm database in your app much like any other pod:

```
use_frameworks!

target "AppTarget" do
  pod "RealmSwift"
end
```

This will pull the Realm Core engine and the Swift language SDK and make them available to your app's code.

Installing RealmSwift in the book projects

As for the projects in this book, they all come with a prepared Podfile, but without the dependency files included. We looked into this option, but it didn't make sense to include all the files for RealmSwift in every single project for each chapter in the book download.

Before you start working on the book, make sure you have the latest version of CocoaPods installed. You need to do that just once before starting to work on the book's projects. Usually, executing this in Terminal will suffice:

```
sudo gem install cocoapods
```

If you want to know more, visit the CocoaPods website:

- https://guides.cocoapods.org/using/getting-started.html.

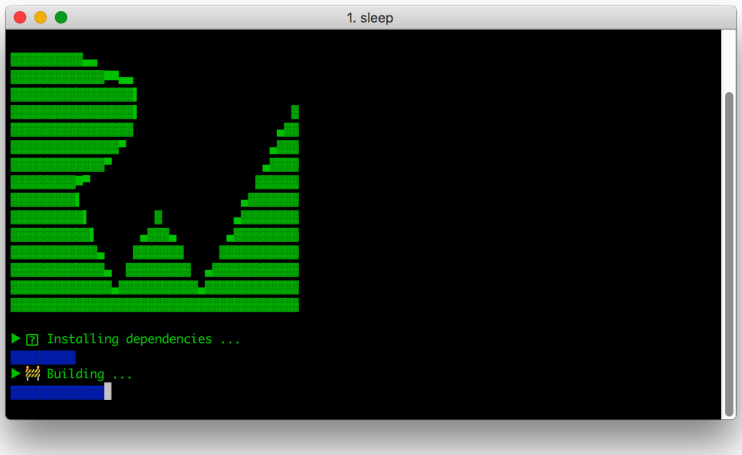
In this book, you will work on two types of projects depending on the complexity of the APIs they cover.

Xcode playgrounds

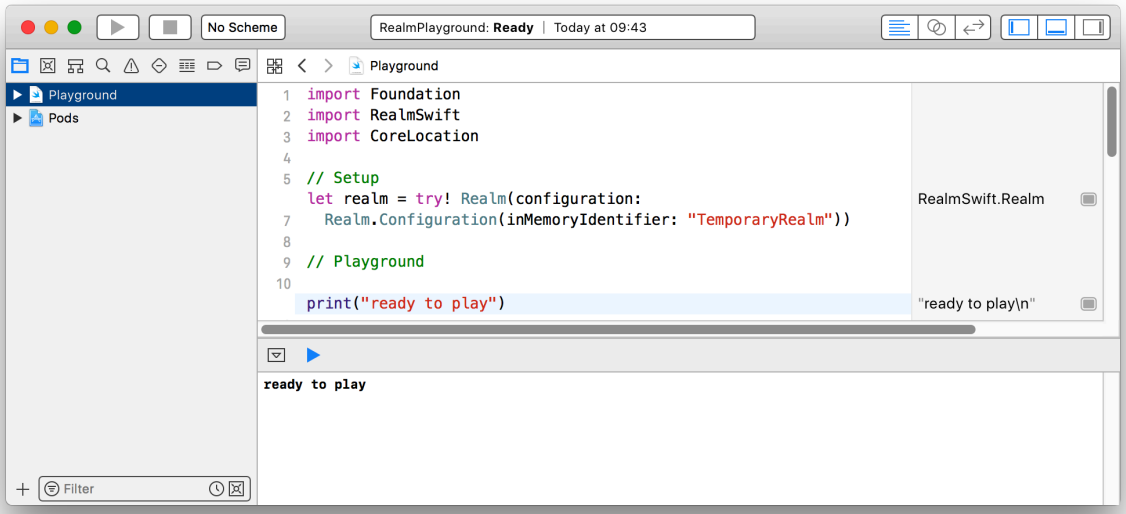
For simpler code experimentation, or examples that need a lot of iterations in which you observe the output and change the code accordingly, you will use **Xcode playgrounds**. For these projects, we've prepared a script that installs Realm and pre-builds the code to save you from waiting around while in Xcode.

For these chapters, open the book's source code folder and find the relevant chapter folder. Copy the **starter** subfolder to a convenient location and use the Terminal app to navigate to the newly copied folder.

Run the script `./bootstrap.sh` to install Realm in the playground project and build the source code. This might take a little while so you will be presented with a neat text-based UI to track the progress:



Once the script has finished, it will open the playground in your default Xcode installation, and you'll be ready to start working through the chapter.



It might look a bit off at first but the generated project does not have any build targets or an active scheme. To get started, click on the playground and start adding the chapter's code as instructed.

Should you experience errors getting the playground to run that's probably Xcode acting up, try to rebuild the playground by running the bootstrap script again like so: `./bootstrap.sh clean`. This should solve any issues related to compiling the sources and get you started working through the relevant book chapter.

Xcode projects

To give you more context, some of the chapters' projects are in the form of complete Xcode projects, which you can run in the iPhone Simulator or on your device.

For these chapters, open the book's source code folder and find the relevant chapter folder. Copy the **starter** subfolder to a convenient location and use the Terminal app to navigate to the newly copied folder.

Install the project dependencies by executing:

```
pod install
```

When the process has finished, open the resulting Xcode workspace file.

> **Note**: If your CocoaPods is not up-to-date and fails to install Realm, run `pod update` once to fetch the meta-information about the latest pods published.

Installing via Carthage

The Carthage installation instructions as described in the Realm documentation are as follows:

- Add Realm to your `Carthage` file: `github "realm/realm-cocoa"`.
- Run `carthage update` to fetch and build Realm.
- Drag **RealmSwift.framework** and **Realm.framework** from the `Carthage/Build/` sub-folder into the **Linked Frameworks and Libraries** section of your project's General section.

This will make RealmSwift available in your app. There is, however, an extra step you need to take before submitting to the App Store, which works around a bug in app submission. You can read more online in Realm's own docs:

- https://realm.io/docs/swift/latest#installation.

Installing like a dynamic framework

For those that don't use dependency managers, Realm provides pre-built frameworks. Instructions on how to integrate those in your project can be found at:

- https://realm.io/docs/swift/latest/#installation

Curious to know more?

Realm is known to be very active in the developer community. The company has launched a number of important iOS open-source projects such as:

- **Realm Database**: A forever-free and open-source database for both personal and commercial projects.
- **SwiftLint**: A tool that enforces strict style guidelines in your Swift code.
- **Jazzy**: A modern tool for generating Swift documentation, which is the defacto standard for building API docs from Swift source code.

Besides these popular projects, Realm's GitHub account https://github.com/realm is packed with demo apps showcasing how to use Realm, additional Realm-based libraries and more.

Additionally, the Realm Academy offers an almost endless amount of recorded conference talks, slides, articles and more:

- https://academy.realm.io.

There, you will find the specialized Realm Academy path. You can also explore other interesting Realm-related talks, video series and blog posts, here:

- https://academy.realm.io/section/realm

Where to go from here?

Even though there is plenty of information out there about learning Realm, the most detailed and in-depth resources on learning how to use the Realm Database in your apps could only be found in a book. Namely, this book! I hope you will enjoy learning from this light-hearted but thorough "missing" Realm manual.

In the next chapter, you'll learn how to quickly build a complete iOS app powered by the Realm database.

Chapter 2: Your First Realm App

In the previous chapter, you learned about the Realm Database, how it works and what problems it can solve for you as a developer.

Now you'll take a leap of faith and dive right into creating an iOS app that uses Realm to persist data on disk while following this tutorial-style chapter.

The idea of this chapter is to get you started with Realm without delving too much into the details of the APIs you're going to use. This will hopefully inspire you to try and find the right APIs, figure out what they do, and perhaps even browse through RealmSwift's source code.

Have no fear though, as the rest of this book will teach you just about everything there is to learn about Realm in detail. You're going to learn the mechanics behind everything you're about to do in this chapter and much, much more.

I invite you to work through this chapter's exercises with an open mind. This chapter is simply about getting a feeling for using Realm in your Swift code.

Getting started

The theme of to-do apps as an educational tool might be getting old by now, but the simple truth is that a to-do app does a great job of demonstrating how to build an app based on data persistence. A to-do app includes features like fetching a list of to-do items from disk, adding, modifying and deleting items, and using the data with some of the common UIKit components such as a table view, an alert view, buttons and more.

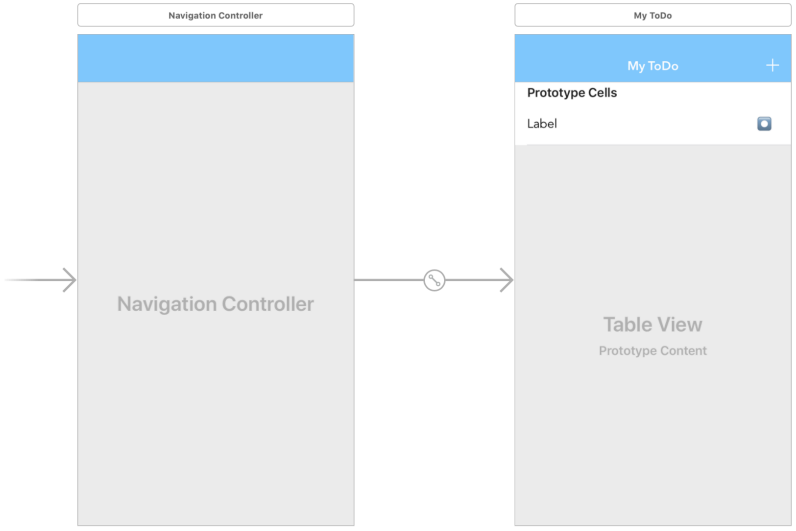
In this chapter, you're going to work on a to-do app that's built on the Realm Database. This will be very useful as you learn the basics of CRUD operations with Realm.

> **Note**: **C**reate, **R**ead, **U**pdate and **D**elete are the basic persistence operations you can perform on mostly any data entity. Often times, this set of operations will be referred by the acronym CRUD. Just don't mix up a CRUD application with a *crude* application!

To get started, open the macOS Terminal app (or another similar app of your choice), navigate to the current chapter's starter project folder, run `pod install` (as described in Chapter 1) and open the newly created Xcode workspace.

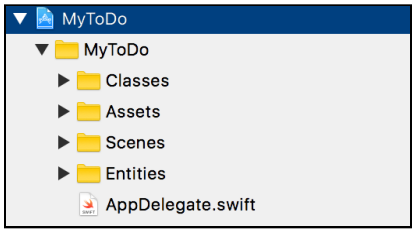
> **Note**: The starter projects in this book contain some UI boilerplate and other non-database related code. Sometimes Xcode might show code warnings since the starter code "misses" some parts that you will add while working through the tasks in the respective chapter.

Open **Main.storyboard** to get an idea of the app's basic structure:



The project consists of a single table view controller with a custom to-do item cell.

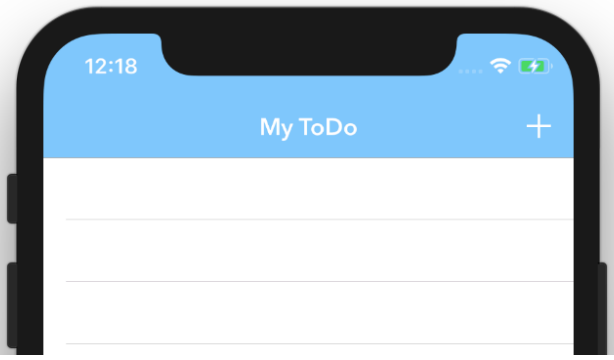
The source code structure follows a general pattern for all of the projects in this book:



- The **Classes** folder is a catch-all location for code that doesn't fall under one of the other folders mentioned below. In this project, it includes an extension on `UIViewController` to add a simple API to present alerts on screen, as well as a handy extension on `UITableView`.
- **Assets** is where you'll find the app's launch screen, meta information .plist file and the asset catalog.
- **Scenes** contains all of the app's scenes — including their view controller, view and view-model code, when available. In this project, you have a single view controller and a custom table cell class.
- **Entities** contains the Realm object classes you'll persist to disk. This is practically your data models but backed by Realm. You have a single class called `ToDoItem` in this project. In later chapters, you'll work on more complex database schemas, but this simple schema will suffice for now.

The projects in this book all follow a similar code structure, but you aren't forced to use this structure in your own work. We've provided it as a guideline so you'll know where to look for files as later projects in this book become more complicated.

If you run the starter app right now, you'll see that it compiles and displays an empty to-do list on screen:



Realm objects

Realm objects are basically a standard data model, much like any other standard data model you've defined in your apps. The only difference is they're backed by Realm persistence and abilities.

You won't dive into Realm objects at this stage, as you'll be going into more detail on how to define Realm models and persist them in the next section of this book. In this chapter, you're going to waltz through these model definitions and get straight into action.

In fact, the MyToDo starter project already includes a class that will serve as the data model for storing to-do items. Open **Entities/ToDoItem.swift** and notice the `ToDoItem` class.

There are a few interesting points to note, here, but you'll learn more about these subjects in the next chapters.

Dynamic properties

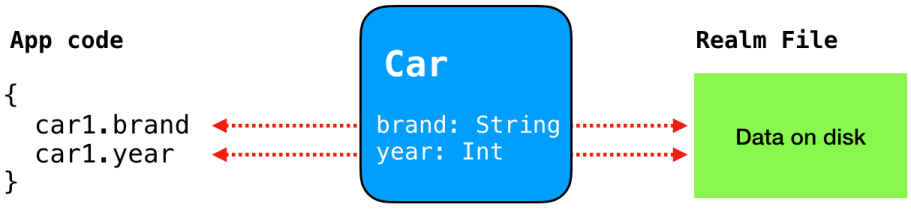
First and foremost, you should recognize that the `ToDoItem` class subclasses `Object`. `Object` is a base class all of your Realm-backed data entities must inherit from. This allows Realm to introspect them and persist them to disk.

Another interesting oddity in this class is that all of its properties are defined as `dynamic`:

```
dynamic var id = UUID().uuidString
dynamic var text = ""
dynamic var isCompleted = false
```

This allows Realm to implement some custom, behind-the-scenes logic, to automatically map these properties to the data persisted to disk.

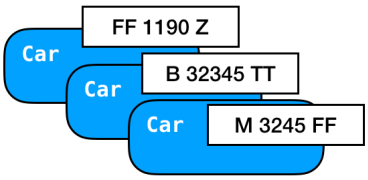
With the help of the `dynamic` keyword, a model class serves as a loose proxy between your app's code and the data stored on disk, as seen in this simple schema:



Primary key

`ToDoItem` also features a convenience `init(_)` and overrides a static method called `primaryKey()` from its parent class. Realm will call `primaryKey()` to decide which property will be used as the object's primary key.

The primary key stores unique values that are used to identify objects in the database. For example, if you're storing `Car` objects in the database, their uniquely identifying primary key can be their registration plate:



To make things easy, the `id` property, which is the primary key of `ToDoItem`, is automatically given a unique string UUID value. The `UUID` Foundation class lets you easily generate unique identifiers like these:

```
DB4722D0-FF33-408D-B79F-6F5194EF018E
409BC9B9-3BD2-42F0-B59D-4A5318EB3195
D9B541AF-16BF-41AC-A9CF-F5F43E5B1D9B
```

Ordinary code

You'll find that the class looks very much like a common Swift class. This is one of the greatest things about Realm! You don't have to go out of your way to adapt your code to work with the persistence layer. You always work with native classes like `ToDoItem`, while Realm does the heavy-lifting behind the scenes automatically.

To recap: `ToDoItem` is a class you can persist on disk because it inherits from Realm's base `Object` class. The class is pretty much ready to go, so you can start adding the code to read and write to-do items from and to disk.

Reading objects from disk

In this section, you're going to write code to retrieve any persisted `ToDoItem` objects and display their data in the main scene of your app.

Start off by adding a method to fetch all to-do items from the Realm file on disk. Open **Entities/ToDoItem.swift** and insert in the extension at the bottom:

```
static func all(in realm: Realm = try! Realm()) -> Results<ToDoItem> {
  return realm.objects(ToDoItem.self)
    .sorted(byKeyPath: ToDoItem.Property.isCompleted.rawValue)
}
```

You just added a static `all(in:)` method, which, by default, fetches all to-do items from your default Realm file. Having a `realm` parameter with a default value allows you to easily work with the default Realm, but also leaves room for using an arbitrary Realm, if needed.

You can actually have more than a single Realm file in your app, as you might want to separate out the data your app uses into different "buckets." You'll learn more about this in Chapter 7, "Multiple Realms/Shared Realms."

> **Note**: You may be outraged by the use of `try!` and of course you will have right to be. For brevity's sake, the book code will only focus on Realm's APIs but if you'd like to learn more about error handling or other Swift-related topics do check the *Swift Apprentice* book, available on raywenderlich.com where we cover Swift itself in great detail.

If you look further down the code, you'll spot the `objects(_)` and `sorted(byKeyPath:)` methods, which are some of the APIs you're going to use throughout this book. `objects(_)` fetches objects of a certain type from disk, and `sorted(byKeyPath:)` sorts them by the value of a given property or key path.

In your code, you ask Realm to return all persisted objects of type `ToDoItem` and sort them by their `isCompleted` property. This will sort incomplete items to the start of the list and completed ones to the end. The method returns a `Results<ToDoItem>`, a generic results type, which gives you dynamic access to the result set.

Next up, you will update your view controller to use this new method to fetch and display the to-do items.

Open **Scenes/ToDoListController.swift** and spot the `items` property towards the top of the file. It is an `Optional` type where you'll store the result fetched from Realm.

Next, append to `viewDidLoad()`:

```
items = ToDoItem.all()
```

This code uses your new `all(in:)` method to ask Realm for all persisted `ToDoItems`.

Currently, the app doesn't display any items when launched, since you don't have any data stored. You'll fix that by adding some default to-do items in case the user hasn't created any.

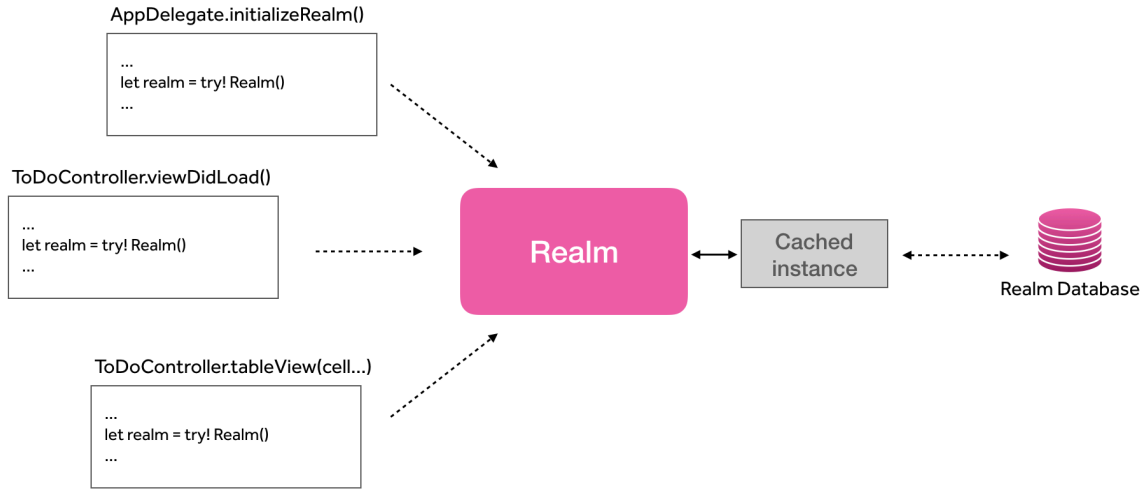
Creating some test data

Open **AppDelegate.swift** and add inside the empty `initializeRealm()` method:

```
let realm = try! Realm()
guard realm.isEmpty else { return }
```

You start by getting an instance of the default Realm by initializing it without any arguments. Then, you check if the Realm is empty using the handy `isEmpty` property. If the Realm isn't empty, you simply return since there's no need to add test data.

Don't worry about the creation of a new Realm in the first line. Initializing a `Realm` object simply creates a handle to the file on disk. Furthermore, this handle is shared across your app and returned each time you use `Realm()` on the same thread. Therefore, you're not duplicating your data, or consuming any extra memory — all pieces of code in this app work with the same Realm instance and the same file on disk.



Next, add code to create some test data, right after your `guard` statement:

```
try! realm.write {
  realm.add(ToDoItem("Buy Milk"))
  realm.add(ToDoItem("Finish Book"))
}
```

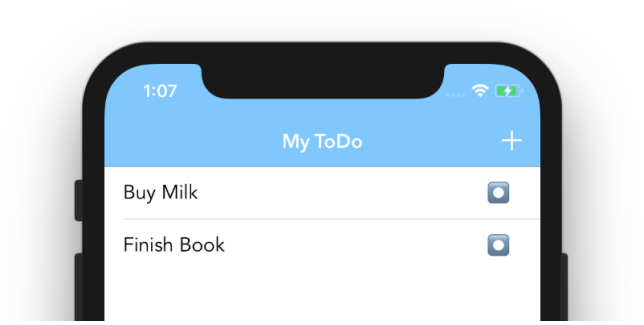
This quick piece of code persists two objects to disk. And by quick, I mean that it's literally only four lines of code!

You start a write transaction by using `realm.write` and add two new `ToDoItem` objects from within the transaction body. To persist objects, you simply create them as you would any other class, and hand them off to `Realm.add(_)` which adds them to your Realm.

Last but not least, call `initializeRealm` from `application(_:didFinishLaunchingWithOptions:)`. Just before `return true`, add the following:

```
initializeRealm()
```

Build and run the project to see your new code in action:



That was easier than expected, wasn't it? To be fair, the starter project *did* include some code to make your life a tad less complicated, but it's clear how easy it is to fetch items from disk and persist new ones when needed.

Open **Scenes/ToDoListController.swift** one more time and look for the `UITableViewDataSource` implementations in the bottom of the file:

- `tableView(_:numberOfRowsInSection:)` uses `items?.count` to return the number of objects fetched in the `items` result set.
- `tableView(_:cellForRowAt:)` uses an index subscript to get the object at the required index path `items?[indexPath.row]` and use its properties to configure the cell.

Additionally, the next class extension defines some delegate methods from the `UITableViewDelegate` protocol that enable swipe-to-delete on table cells. You'll write the code to actually delete items a bit later in this chapter.

Adding an item

Next, you'll add code to allow the user to add new to-do items to their list.

Since this is one of the CRUD operations, add the relevant method to the `ToDoItem` class. Open **Entities/ToDoItem.swift** and add right below your `all(in:)` method:

```
@discardableResult
static func add(text: String, in realm: Realm = try! Realm())
  -> ToDoItem {
  let item = ToDoItem(text)
  try! realm.write {
    realm.add(item)
  }
  return item
}
```

`add(text:in:)` lets you create a new `ToDoItem` instance *and* persist it to a realm of your choice. This is a useful shortcut when you don't intend to use an object outside of the context of the database.

You're already familiar with the type of code above. You create a new `ToDoItem` instance, open a write transaction, and use `Realm.add(_)` to persist the object.

You can now add some UI code to your view controller to let the user input new to-do items and add them to the app's realm.

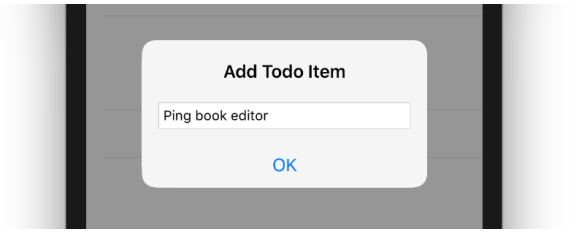
Back in **Scenes/ToDoListController.swift**, scroll to `addItem()`, which is a method already connected to the + button in your navigation bar. Add inside `addItem()`:

```
userInputAlert("Add Todo Item") { text in
  ToDoItem.add(text: text)
}
```

`userInputAlert(_)` is a `UIViewController` extension method (found in the **Classes** folder of the starter project) that presents a new alert controller on the screen and asks the user to enter some text. Once the user taps **OK**, you'll receive the user-provided text in a closure.

In the callback closure, you use the new method you just created to create and persist a new to-do item to disk: `ToDoItem.add(text: text)`.

Run the project one more time and tap on the + button. `userInputAlert(_:_:)` will display an alert on screen and let you enter the text for a new to-do.

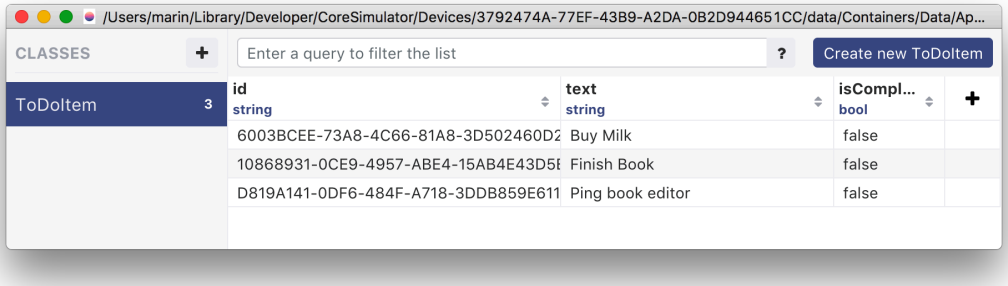


Tapping **OK** will execute your callback and save the new to-do item to disk.

As you might have noticed, the table view still only displays the two items you fetched when initially loading the view controller.

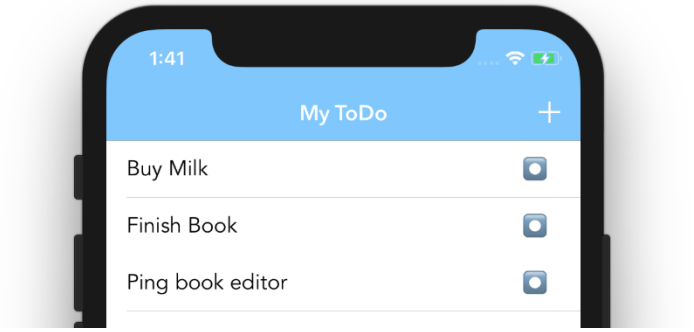
Use Realm Studio to open the app database **default.realm** from the Simulator folders and check its contents. For this, you can use the SimPholders tool as mentioned in Chapter 1, “Hello Realm”:

Realm Studio displays a list of all classes stored in your file on the left-hand side and a spreadsheet-like UI on the right side letting you browse all the data persisted in the file:



Hey, that new to-do item has been successfully added — you can find it at the bottom of the list!

In fact, if you re-run the project, you'll see it appear in your app as well:



It seems like you need a way to refresh the table view whenever the database changes.

Reacting to data changes

One sub-optimal way to do this is to refresh the table view from `addItem()`. But going down this path means you'll need to refresh the table view from *every* method that modifies your data, such as when you delete an item, or set its completion status, and so on. Leaving that aside, the real issue is how to refresh the table if you commit a change from another class, which runs somewhere in the background, and is completely decoupled from the view controller?

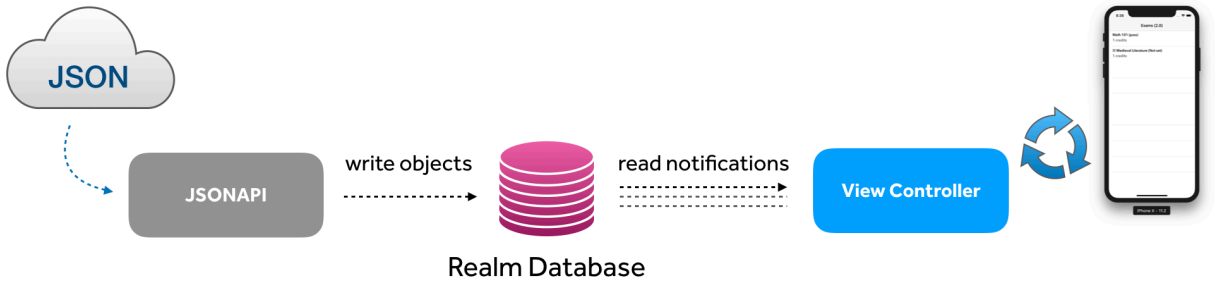
Fortunately, Realm provides a powerful solution to this. Realm's own change notifications mechanism lets your classes read and write data independently and be notified, in real-time, about any changes that occurred.

Realm's own notification system is incredibly useful because it lets you cleanly separate your data persistence code. Take a look at an example that uses two classes:

- A networking class, which persists JSON as Realm objects on a background queue.
- A view controller displaying the fetched objects in a collection view.

Without Realm and change notifications, you'll need to make one class a delegate of the other (in a way that inevitably couples them) or use `NotificationCenter` to broadcast update notifications.

With Realm, the two classes can cleanly separate their concerns. One will only *write* to the database the other only *reads* and *observes changes*:



With this setup, it would be trivial to do something like test the class that writes objects to the database without creating a view controller. In addition, in the event the app goes offline, the view controller won't care at all about the fact the class responsible for writing objects isn't doing any work at the moment.

Relying on Realm's built-in change notifications lets you separate concerns extremely well and keeps your app's architecture simple and clean.

See how that looks in practice.

Open **Scenes/ToDoListController.swift** and add a new property at the top of the class:

```
private var itemsToken: NotificationToken?
```

A notification token keeps a reference to a subscription for change notifications. You'll use notifications throughout the book so you'll get to learn all about them, starting in Chapter 6, "Notifications and Reactive Apps."

Continue in `viewWillAppear(_)` where you'll set your notification token:

```
itemsToken = items?.observe { [weak tableView] changes in
  guard let tableView = tableView else { return }

  switch changes {
  case .initial:
    tableView.reloadData()
  case .update(_, let deletions, let insertions, let updates):
    tableView.applyChanges(deletions: deletions, insertions: insertions,
updates: updates)
  case .error: break
  }
}
```

You call `observe(_)` on the to-do items result set, which lets Realm know that you want to receive updates any time the result set changes.

For example, if you add a to-do item, Realm will call your `observe` callback. If you remove a to-do item, Realm will call your callback. If you change a property on one of your to-do items… yes, you guessed right — Realm will call your callback.

The `observe(_)` callback closure is the place to implement any UI code that will reflect the latest data changes in your app's UI. In the code above, you receive detailed information about what items have been inserted, modified or deleted. If any changes occurred in your result set, you call the `applyChanges(_)` extension method to apply them on screen, and you also take care of simply reloading the table view with the initial data at the time of observing changes.

> **Note**: The callback is called on the same thread you create the subscription on. In your code above you create the subscription in `viewWillAppear(_)` and is, therefore, safe to update the app's UI without any extra checks.

That's as far as you'll take this right now. Later on, you'll learn about the notification data in greater detail.

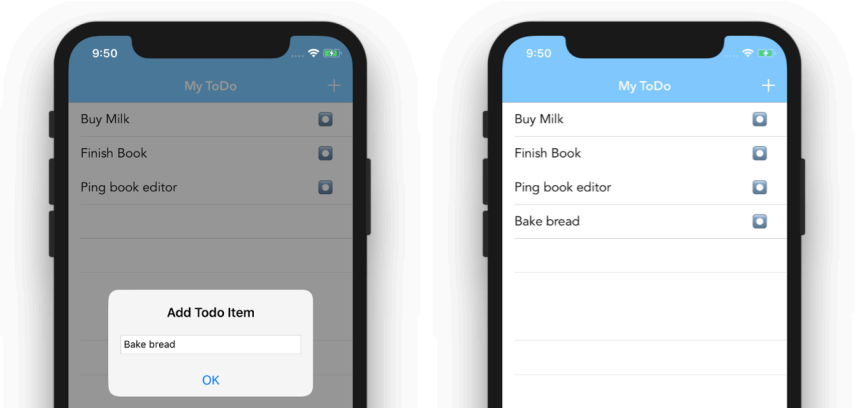
Next, since you start observing `items` in `viewWillAppear(_)`, it makes sense to stop the observation in `viewWillDisappear(_)`.

Add the following to `viewWillDisappear(_)`:

```
itemsToken?.invalidate()
```

`invalidate()` invalidates the token and cancels the data observation.

Run the project again. This time, as soon as you enter a new to-do item, it'll appear in your list with a nice accompanying animation:



Now that you have the table all reactive and animated, you can add the remaining CRUD operations. Thanks to Realm's change notifications, the table will reflect any changes automatically. Isn't that great?

Modifying a persisted object

You just learned how to add new objects to your app's Realm, but how would you modify an object that has already been persisted?

Obviously, you first need to fetch the object from the database and then modify it somehow. In this section of the chapter, you're going to add code to complete (and *un-complete*?) a to-do task.

Open **Entities/ToDoItem.swift** and add a new method below `add(text:in:)`:

```
func toggleCompleted() {
  guard let realm = realm else { return }
  try! realm.write {
    isCompleted = !isCompleted
  }
}
```

`toggleCompleted()` is a new method that allows you to easily toggle the status of a to-do item from incomplete to completed and vice-versa.

Every object persisted to a realm has a `realm` property, which provides you with quick access to the Realm where the object is currently persisted on.

You start by unwrapping the `ToDoItems`' `realm` and start a new write transaction, just like you did before.

From within the transaction, you toggle `isCompleted`. As soon as the transaction has been successfully committed, that change is persisted on disk and propagated throughout your observation to change notifications. You can now add the code to toggle the item whenever the user taps the right button on each to-do item cell.

Switch back to **Scenes/ToDoListController.swift** and add the following method below `addItem()`:

```
func toggleItem(_ item: ToDoItem) {
  item.toggleCompleted()
}
```

This method calls your newly created `toggleCompleted()` on a given to-do item object. You can use `toggleItem(_)` in the code that configures each individual to-do cell.

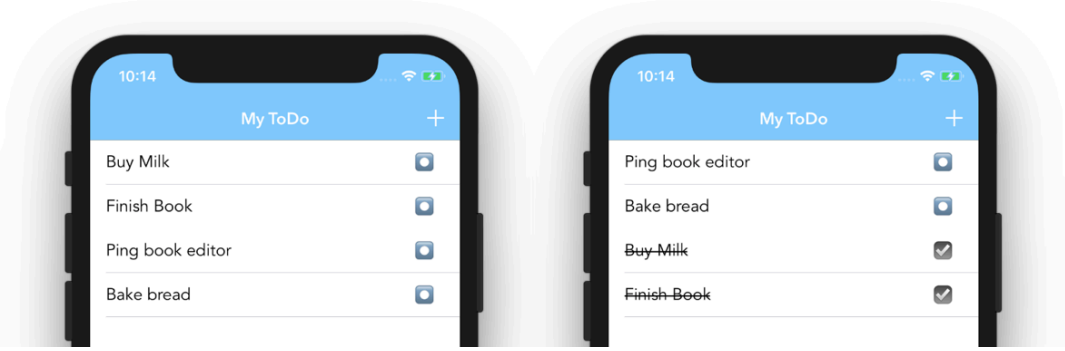
Scroll down to `tableView(_:cellForRowAt:)` and insert inside the callback closure for `cell.configureWith`:

```
self?.toggleItem(item)
```

When the user taps the cell button, it will call your code back and invoke `toggleItem(item)`, which will toggle that item's status.

That should take care of toggling to-do items in your project.

Run the app one more time and try tapping the status button of some of those to-do items:



As soon as you modify any of the to-do objects, the view controller is being notified about the change and the table view reflects the latest persisted data.

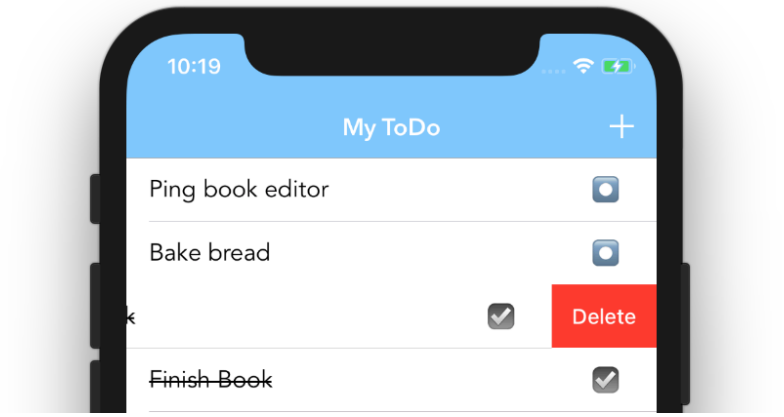
You'll also notice that items you mark as completed will animate towards the bottom of the list. This is because Realm's `Results` class reorders the objects in the collection according to the sorting you applied when you initially started observing changes. If you look back to `ToDoItem.all(in:)`, you'll see you're sorting the results by their `isCompleted` property — incomplete tasks first and completed tasks last.

Deleting items

Last but not least, you're going to let the user delete items from their list.

This is quite similar to adding and modifying items: You're going to add a new method on `ToDoItem` and then add the relevant code in the view controller to react to user events.

Thanks to the two `UITableViewDelegate` methods already included in the starter code of `ToDoListController`, the table view already reacts to left-swipes and displays a red **Delete** button:



This provides a good starting point for this chapter's last task. Let's get down to business!

Open **Entities/ToDoItem.swift** and add one last method to the extension, below `toggleCompleted()`:

```
func delete() {
  guard let realm = realm else { return }
  try! realm.write {
    realm.delete(self)
  }
}
```

Just like before, you get a reference to the object's Realm and then start a write transaction to perform your updates.

> **Note**: As you'll learn later, if you try modifying a persisted object without starting a write transaction, your code will throw an exception. You can only modify managed objects inside a Realm write transaction.

Since the class is a Realm object, you can simply call `realm.delete(self)` to delete the current object from the realm.

Finally, you need to add a few more lines in your view controller to call your new `delete()` method. Back in **Scenes/ToDoListController.swift** add below `toggleItem(_:)`:

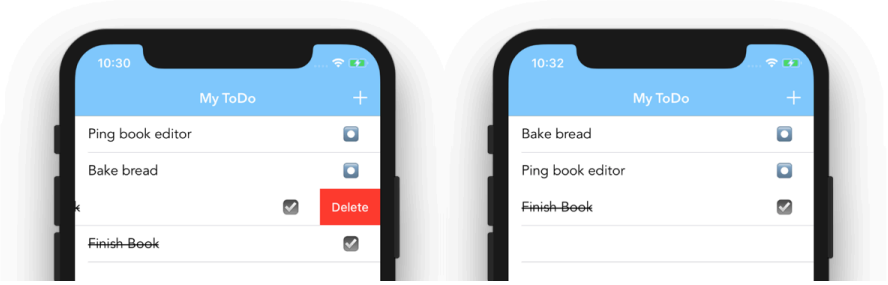
```
func deleteItem(_ item: ToDoItem) {
  item.delete()
}
```

Then, scroll down to `tableView(_:commit:forRowAt:)` and append at the bottom:

```
deleteItem(item)
```

This will call your new `deleteItem(_)` method, which in turn invokes the `delete()` method on the to-do item.

Run the app one last time, swipe left on a to-do item, and tap **Delete**:



Just like the previous features you added, the deletion of the object from the Realm is reflected in the table, accompanied by a pleasant animation.

With that last piece of code, your simple CRUD application is complete. You've learned a bit about fetching objects from a Realm file, adding and modifying existing objects, and how to react to data changes and keeping your read and write code separate.

In fact, you already possess the knowledge to create simple Realm apps! However, since working with Realm has so many advantages, you'll want to expand your knowledge as soon as possible. Worry not, we've got you covered. The rest of this book provides everything you'll need to learn about Realm in detail.

Challenge

Challenge: Enhance your to-do app with more features

To warm up for the next chapter, work through a few small tasks to polish your to-do app.

Start by modifying the existing code so it only allows the deletion of completed tasks. Way to simulate ticking items off the list!

Finally, add a feature that allows the user to tap a cell and be presented with an alert where they can edit the current to-do item's text. Once they close the alert, the text change will be persisted, and the UI should be updated accordingly.

This chapter didn't go into much detail in regards to the various available APIs, so don't worry too much if you can't figure out how to complete this challenge. You can open the challenge folder of this chapter and peek at the completed solution code. At this point, it's not expected you can figure out everything on your own.

These challenges might not be very complex, but they'll get you writing some simple Realm code to warm up for the grand tour of Realm's object features in the next chapter.

Where to Go From Here?

We hope you enjoyed this sample of *Realm: Building Modern Swift Apps with Realm Database*!

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

1. **Hello Realm!**: Realm finds the sweet spot between the simplicity of storing data as JSON on disk and using heavy, slow ORMs like Core Data or similar that are built on top of SQLite. In this book you are going to learn plenty about Realm, how to build iOS apps with Realm, and you'll pick up some tips and tricks along the way about getting the most out of the platform.

2. **Your First Realm App**: In the previous chapter, you learned about the Realm Database, how it works and what problems it can solve for you as a developer. Now you'll take a leap of faith and dive right into creating an iOS app that uses Realm to persist data on disk while following this tutorial-style chapter.

3. **Object Basics and Data types**: You now have a grasp of the easy and clean way of writing Realm-related code. In this chapter, you'll dive deeper into Realm's Swift API and go over many of the available classes and their methods in order to get a solid understanding of Realm's superpowers, as well as some of its limitations.

4. **Schema relationships**: You've mastered your object-type properties and your primitives, learned how to add a primary key and indices, and other important details. Your data, however, isn't isolated when in a Realm. You can have many different objects connected to each other in all sorts of useful and meaningful ways. This chapter will teach you all about building powerful and efficient relationships between objects.

5. **Reading and Writing Objects**: You've had a sneak peek into persisting objects and reading them back, but you barely scratched the surface of dealing with stored objects, as you'll realize in this chapter. In the first half of this chapter you'll look into fetching, querying, and sorting persisted data from Realm. In the second half, you will cover the Realm APIs that let you add, update, and delete objects in your Realm.

6. **Notifications and Reactive Apps**: You are probably eager to use the in-depth knowledge you soaked up working through the last few chapters in practice, and rightfully so. In this chapter, you'll learn about Realm's built-in notification APIs. Realm features a rather clever system of detecting any changes, regardless of the thread or process responsible for those changes, and deliver notifications to any observers.

7. **Realm Configurations**: In this chapter that covers Realm configurations, you'll take a step back (or is it one forward?), dig into working with the Realm class itself, and learn about how to configure Realm using Realm.Configuration. You'll learn how to work with different Realm files on disk and in-memory, as well as how to use advanced features such as data encryption.

8. **Multiple Realms / Shared Realms**: In this chapter, you're going to make use of your existing Realm skills while learning some new ones. You're going to use multiple configurations, read and write data, use notifications to build reactive UI, and explore new topics like sharing data between your app and a Today extension.

9. **Dependency Injection and Testing**: In this chapter, you're going to touch on two important topics: how to use dependency injection to improve the architecture of your Flash Cards apps, and how to write both synchronous and asynchronous tests powered by Realm. This chapter won't delve into topics such as test driven development, but will instead focus specifically on tips and tricks for testing classes that use Realm objects and depend on Realm-specific functionality such as change notifications.

10. **Effective Multi-threading**: Many of the issues you would experience with database ORMs and/or other databases are rooted in asynchronous, multi-threaded code. Accessing your data from concurrent threads in an efficient and safe manner is, unsurprisingly, not very straightforward and quite error-prone. Realm saves you from the hassles of over-thinking concurrent threads, since it's been planned with a deeply integrated multi-threading strategy which makes concurrent access to the database a walk in the park.

11. **Beginning Realm Migrations**: Nothing in life (and code) remains static forever. So what happens, then, when the app you're working on becomes wildly successful and you need release a new version, and then another one, and another one? In this chapter you'll learn how you can migrate the schema of a Realm file as it evolves alongside your app.
12. **Advanced Schema Migrations**: In this chapter, you're going to build a more complex migration in an Xcode project. The demo is designed in such a way that you can build three different versions of the same app. This will allow you to write code and simulate real-life app upgrades, all from the comfort of a single Xcode project.
13. **Extending Realm with Custom Features**: In this chapter, you'll work on adding a new database feature to Realm: hard and soft cascading deletes. While this won't be a universal drop-in feature, it will give you direction on how and when to extend the database.
14. **Real-Time Sync with Realm Cloud**: In this chapter, you'll learn more about Realm Platform and take Realm Cloud for a test drive. You'll develop a working chat app using Realm Cloud as your backend.

You can find the book on the raywenderlich.com store here: https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database

We hope you enjoy the book! :]

— Marin and the *Realm: Building Modern Swift Apps with Realm Database* team

Learn Realm Database and Realm Cloud!

Realm finds the sweet spot between the simplicity of storing data as JSON on disk and using heavy, slow ORMs like Core Data or similar that are built on top of SQLite. And although the Realm documentation is pretty complete, you need a more detailed approach to help you learn how to leverage Realm properly in your app.

Realm: Building Modern Swift Apps with Realm Database is here to help! This book is the easiest and fastest way to get hands-on experience with using Realm Database in your apps.

Who This Book Is For

This book is for anyone who would like to leverage the power of Realm Database or Realm Cloud in their apps. The book starts with a gentle introduction, then moves on to more complicated scenarios, including migrations, real-time sync and more.

Topics Covered in Realm:

- **Getting Started:** Dive right into creating an iOS app that uses Realm to persist data on disk while following this tutorial-style chapter.
- **Object Basics and Data Types:** Go deeper into Realm's Swift API and discover the available classes and their methods to understand Realm's superpowers.
- **Schema Relationships:** Learn all about building powerful and efficient relationships between objects.
- **Notifications and Reactive Apps:** See how to leverage Realm's built-in notification APIs to deliver notifications to any observers.
- **Multiple Realms / Shared Realms:** Use multiple configurations, read and write data, and explore new topics like sharing data in your app.
- **Dependency Injection and Testing:** Learn how to use dependency injection, and how to write both synchronous and asynchronous tests in your app.
- **Schema Migrations:** See how to migrate the schema of a Realm file as it evolves alongside your app.
- **Real-Time Sync:** Get started with Realm Cloud and learn how to apply your existing Realm Database skills to Realm Cloud.

One thing you can count on: after reading this book, you'll be well-prepared to use Realm in your own apps!

About Marin Todorov

Marin Todorov is one of the founding members of the **raywenderlich.com tutorial team** and has worked on seven of the team's books. Besides crafting code, Marin also enjoys blogging, teaching, and speaking at conferences. He happily open-sources code.