Kyle Richter
Joe Keeley

Second Edition

Mastering iOS Frameworks

Beyond the Basics

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Mastering iOS Frameworks

Developer's Library Series

Visit **developers-library.com** for a complete list of available products

---

The **Developer's Library Series** from Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that's useful for other programmers.

Developer's Library books cover a wide range of topics, from open-source programming languages and databases, Linux programming, Microsoft, and Java, to Web development, social networking platforms, Mac/iPhone programming, and Android programming.

PEARSON

Addison-Wesley | Cisco Press | EXAM/CRAM | IBM Press | QUE | PRENTICE HALL | SAMS | Safari Books Online

Mastering iOS Frameworks

Beyond the Basics, Second Edition

Kyle Richter
Joe Keeley

Addison-Wesley

Hoboken, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Control Number: 2015932706

ISBN-13: 978-0-134-05249-6
ISBN-10: 0-134-05249-8
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing: April 2015

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Sheri Replin

Managing Editor
Kristy Hart

Project Editor
Elaine Wiley

Copy Editor
Cheri Clark

Indexer
Ken Johnson

Proofreader
Kathy Ruiz

Technical Reviewers
Niklas Saers
Justin Williams

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Senior Compositor
Gloria Schurick

❖

I would like to dedicate this book to my co-workers who continually drive me to never accept the first solution.

—Kyle Richter

I dedicate this book to my wife, Irene, and two daughters, Audrey and Scarlett. Your boundless energy and love inspire me daily.

—Joe Keeley

❖

Table of Contents

Foreword

I have been working with the iPhone SDK (now iOS SDK) since the first beta released in 2008. At the time, I was focused on writing desktop apps for the Mac and hadn't thought much about mobile app development.

If you chose to be an early adopter, you were on your own. In typical Apple fashion, the documentation was sparse, and since access to the SDK required an NDA—and, apparently, a secret decoder ring—you were on your own. You couldn't search Google or turn to StackOverflow for help, and there sure as hell weren't any books out yet on the SDK.

In the seven years (yes, it really has been only seven years) since Apple unleashed the original iPhone on the world, we've come a long way. The iPhone SDK is now the iOS SDK. There are dozens of books and blogs and podcasts and conferences on iOS development. And ever since 2009, WWDC has been practically impossible to get into, making it even harder for developers—old and new—to learn about the latest features coming to the platform. For iOS developers, there is so much more to learn.

One of the biggest challenges I have as an iOS developer is keeping on top of all the components and frameworks available in the kit. The iOS HIG should help us with that, but it doesn't go far enough—deep enough. Sure, now I can find some answers by searching Google or combing through StackOverflow; but, more often than not, those answers only explain the how and rarely the why, and they never provide the details you really need.

And this is what Kyle and Joe have done with this book—they're providing the detail needed so you can fully understand the key frameworks that make up the iOS SDK.

I've had the pleasure of knowing Kyle and Joe for a number of years. They are two of the brightest developers I have ever met. They have each written some amazing apps over the years, and they continuously contribute to the iOS development community by sharing their knowledge—speaking at conferences and writing other books on iOS development. If you have a question about how to do something in iOS, chances are good that Kyle and Joe have the answer for you.

But what makes these guys so awesome is not just their encyclopedic knowledge of iOS, but their willingness to share what they know with everyone they meet. Kyle and Joe don't have competitors, they have friends.

Kyle and Joe's in-depth knowledge of the iOS SDK comes through in this book. It's one of the things I like about this book. It dives into the details for each component covered at a level that you won't always find when searching online.

I also like the way the book is structured. This is not something that you'll read cover to cover. Instead, you'll pick up the book because you need to learn how to implement a collection view or sort out some aspect of running a task in a background thread that you can't quite wrangle. You'll pick up the book when you need it, find the solution, implement it in your own code,

and then toss the book back on the floor until you need it again. This is what makes *Mastering iOS Frameworks* an essential resource for any iOS developer—regardless of your experience level. You might think you're a master with Core Location and MapKit, but I reckon you'll find something here that you never knew before.

Kyle and Joe don't come with egos. They don't brag. And they sure don't act like they are better than any other developer in the room. They instill the very spirit that has made the Mac and iOS developer community one of the friendliest, most helpful in our industry, and this book is another example of their eagerness to share their knowledge.

This book, just like the seminal works from Mark and LaMarche or Sadun, will always be within arm's reach of my desk. This is the book I wish I had when I first started developing iOS apps in 2008. Lucky you, it's here now.

—Kirby Turner

Chief Code Monkey at White Peak Software, author of *Learning iPad Programming: A Hands-On Guide to Building iPad Apps, Second Edition* (Addison-Wesley Professional), and Cocoa developer community organizer and conference junkie

Preface

Welcome to *Mastering iOS Frameworks: Beyond the Basics*!

There are hundreds of "getting started with iOS" books available to choose from, and there are dozens of advanced books in specific topics, such as Core Data or Security. There was, however, a disturbing lack of books that would bridge the gap between beginner and advanced niche topics.

This publication aims to provide development information on the intermediate-to-advanced topics that are otherwise not worthy of standalone books. It's not that the topics are uninteresting or lackluster; it's that they are not large enough topics. From topics such as working with JSON to accessing photo libraries, these are frameworks that professional iOS developers use every day but are not typically covered elsewhere.

Additionally, several advanced topics are covered to the level that many developers need in order to just get started. Picking up a 500-page Core Data book is intimidating, whereas Chapter 15 of this book provides a very quick and easy way to get started with Core Data. Additional introductory chapters are provided for debugging and instruments, TextKit, HomeKit, HealthKit, and CloudKit.

Topics such as Game Center leaderboards and achievements, AirPrint, music libraries, Address Book, and Passbook are covered in their entirety. Whether you just finished your first iOS project or you are an experienced developer, this book has something for you.

The chapters have all been updated to work with iOS 8. Please let us know if you encounter issues and we will release updates and corrections.

If you have suggestions, bug fixes, corrections, or anything else you'd like to contribute to a future edition, please contact us at mastering.ios.frameworks@gmail.com. We are always interested in hearing what would make this book better and are very excited to continue refining it.

—Kyle Richter and Joe Keeley

Prerequisites

Every effort has been made to keep the examples and explanations simple and easy to digest; however, this is to be considered an intermediate to advanced book. To be successful with it, you should have a basic understanding of iOS development, Objective-C, and C. Familiarity with the tools such as Xcode, Developer Portal, iTunes Connect, and Instruments is also assumed. Refer to *Programming in Objective-C,* by Stephen G. Kochan, and *Learning iOS Development,* by Maurice Sharp, Rod Strougo, and Erica Sadun, for basic Objective-C and iOS skills.

What You'll Need

Although you can develop iOS apps in the iOS simulator, it is recommended that you have at least one iOS device available for testing:

- Apple iOS Developer Account: The latest version of the iOS developer tools including Xcode and the iOS SDKs can be downloaded from Apple's Developer Portal (http://developer.apple.com/ios). To ship an app to the App Store or to install and test on a personal device, you will also need a paid developer account at $99 per year.
- **Macintosh Computer:** To develop for iOS and run Xcode, you will need a modern Mac computer capable of running the latest release of OS X.
- **Internet Connection:** Many features of iOS development require a constant Internet connection for your Mac as well as for the device you are building against.

How This Book Is Organized

With few exceptions (Game Center and Core Data), each chapter stands on its own. The book can be read cover to cover but any topic can be skipped to when you find a need for that technology; we wrote it with the goal of being a quick reference for many common iOS development tasks.

Here is a brief overview of the chapters you will encounter:

- **Chapter 1, "UIKit Dynamics":** iOS 7 introduced UI Kit Dynamics to add physics-like animation and behaviors to UIViews. You will learn how to add dynamic animations, physical properties, and behaviors to standard objects. Seven types of behaviors are demonstrated in increasing difficulty from gravity to item properties.
- **Chapter 2, "Core Location, MapKit, and Geofencing":** iOS 6 introduced new, Apple-provided maps and map data. This chapter covers how to interact with Core Location to determine the device's location, how to display maps in an app, and how to customize the map display with annotations, overlays, and callouts. It also covers how to set up regional monitoring (or geofencing) to notify the app when the device has entered or exited a region.
- **Chapter 3, "Leaderboards":** Game Center leaderboards provide an easy way to add social aspects to your iOS game or app. This chapter introduces a fully featured iPad game called Whack-a-Cac, which walks the reader through adding leaderboard support. Users will learn all the required steps necessary for implementing Game Center leaderboards, as well as get a head start on implementing leaderboards with a custom interface.
- **Chapter 4, "Achievements":** This chapter continues on the Whack-a-Cac game introduced in Chapter 3. You will learn how to implement Game Center achievements in a fully featured iPad game. From working with iTunes Connect to displaying achievement progress, this chapter provides all the information you need to quickly get up and running with achievements.

- **Chapter 5, "Getting Started with Address Book":** Integrating a user's contact information is a critical step for many modern projects. Address Book framework is one of the oldest available on iOS; in this chapter you'll learn how to interact with that framework. You will learn how to use the people picker, how to access the raw address book data, and how to modify and save that data.
- **Chapter 6, "Working with Music Libraries":** This chapter covers how to access the user's music collection from a custom app, including how to see informational data about the music in the collection, and how to select and play music from the collection.
- **Chapter 7, "Implementing HealthKit":** HealthKit provides a centralized location for health information that can be shared among apps. This chapter explains how to get started with HealthKit, how to access information available in HealthKit, and how to read and write various types of health data.
- **Chapter 8, "Implementing HomeKit":** This chapter explains how to get started using HomeKit, which enables iOS devices to communicate with home automation technology. It explains how to set up a home in HomeKit, and how to discover, set up, and interact with home automation devices such as lights, locks, and garage door openers.
- **Chapter 9, "Working with and Parsing JSON":** JSON, or JavaScript Object Notation, is a lightweight way to pass data back and forth between different computing platforms and architectures. As such, it has become the preferred way for iOS client apps to communicate complex sets of data with servers. This chapter describes how to create JSON from existing objects, and how to parse JSON into iOS objects.
- **Chapter 10, "Notifications":** Two types of notifications are supported by iOS: local notifications, which function on the device with no network required, and remote notifications, which require a server to send a push notification through Apple's Push Notification Service to the device over the network. This chapter explains the differences between the two types of notifications, and demonstrates how to set them up and get notifications working in an app.
- **Chapter 11, "Cloud Persistence with CloudKit":** CloudKit offers public and private remote data storage, with notifications for changes in data. This chapter explains the basic CloudKit concepts, and illustrates how to build an app that uses CloudKit for storing and syncing both private and public data remotely.
- **Chapter 12, "Extensions":** Extensions provide a way to access an app's functionality outside the app's sandbox. This chapter explains the different types of extensions that are available, and illustrates how to create a Today extension and an Apple Watch extension.
- **Chapter 13, "Handoff":** Handoff is one of the Continuity features introduced with iOS 8 and Yosemite. It enables the user to switch between devices and have an activity seamlessly move from one device to another. This chapter explains the basic Handoff mechanisms, and how to implement Handoff for developer-defined activities and document-based activities.

- **Chapter 14, "AirPrint":** An often-underappreciated feature of the iOS, AirPrint enables the user to print documents and media to any wireless-enabled AirPrint-compatible printer. Learn how to quickly and effortlessly add AirPrint support to your apps. By the end of this chapter you will be fully equipped to enable users to print views, images, PDFs, and even rendered HTML.
- **Chapter 15, "Getting Up and Running with Core Data":** This chapter demonstrates how to set up an app to use Core Data, how to set up a Core Data data model, and how to implement many of the most commonly used Core Data tools in an app. If you want to start using Core Data without digging through a 500-page book, this chapter is for you.
- **Chapter 16, "Integrating Twitter and Facebook Using Social Framework":** Social integration is the future of computing, and it is accepted that all apps have social features built in. This chapter walks you through adding support for Facebook and Twitter to your app using the Social Framework. You will learn how to use the built-in composer to create new Twitter and Facebook posts. You will also learn how to pull down feed information from both services and how to parse and interact with that data. Finally, using the frameworks to send messages from custom user interfaces is covered. By the end of this chapter, you will have a strong background in Social Framework as well as working with Twitter and Facebook to add social aspects to your apps.
- **Chapter 17, "Working with Background Tasks":** Being able to perform tasks when the app is not the foreground app was a big new feature introduced in iOS 4, and more capabilities have been added since. This chapter explains how to perform tasks in the background after an app has moved from the foreground, and how to perform specific background activities allowed by iOS.
- **Chapter 18, "Grand Central Dispatch for Performance":** Performing resource-intensive activities on the main thread can make an app's performance suffer with stutters and lags. This chapter explains several techniques provided by Grand Central Dispatch for doing the heavy lifting concurrently without affecting the performance of the main thread.
- **Chapter 19, "Using Keychain and TouchID to Secure and Access Data":** Securing user data is important and an often-overlooked stage of app development. Even large public companies have been called out in the news over the past few years for storing user credit card info and passwords in plain text. This chapter provides an introduction to not only using the Keychain to secure user data but developmental security as a whole. By the end of the chapter, you will be able to use Keychain to secure any type of small data on users' devices and provide them with peace of mind.
- **Chapter 20, "Working with Images and Filters":** This chapter covers some basic image-handling techniques, and then dives into some advanced Core Image techniques to apply filters to images. The sample app provides a way to explore all the options that Core Image provides and build filter chains interactively in real time.

- **Chapter 21, "Collection Views":** Collection views, a powerful new API introduced in iOS 6, give the developer flexible tools for laying out scrollable, cell-based content. In addition to new content layout options, collection views provide exciting new animation capabilities, both for animating content in and out of a collection view and for switching between collection view layouts. The sample app demonstrates setting up a basic collection view, a customized flow layout collection view, and a highly custom, nonlinear collection view layout.
- **Chapter 22, "Introduction to TextKit":** iOS 7 introduced TextKit as an easier-to-use and greatly expanded update to Core Text. TextKit enables developers to provide rich and interactive text formatting to their apps. Although TextKit is a very large subject, this chapter provides the basic groundwork to accomplish several common tasks, from adding text wrapping around an image to inline custom font attributes. By the end of this chapter, you will have a strong background in TextKit and have the groundwork laid to explore it more in depth.
- **Chapter 23, "Gesture Recognizers":** This chapter explains how to make use of gesture recognizers in an app. Rather than dealing with and interpreting touch data directly, gesture recognizers provide a simple and clean way to recognize common gestures and respond to them. In addition, custom gestures can be defined and recognized using gesture recognizers.
- **Chapter 24, "Accessing the Photo Library":** The iPhone has actually become a very popular camera, as evidenced by the number of photos that people upload to sites such as Flickr. This chapter explains how to access the user's photo library, and handle photos and videos in a custom app. The sample app demonstrates building some of the concepts from the iOS 8 version of Photos.app.
- **Chapter 25, "Passbook and PassKit":** With iOS 6, Apple introduced Passbook, a standalone app that can store "passes," or such things as plane tickets, coupons, loyalty cards, or concert tickets. This chapter explains how to set up passes, how to create and distribute them, and how to interact with them in an app.
- **Chapter 26, "Debugging and Instruments":** One of the most important aspects of development is to be able to debug and profile your software. Rarely is this topic covered even in a cursory fashion. This chapter introduces you to debugging in Xcode and performance analysis using Instruments. Starting with a brief history of computer bugs, the chapter walks you through common debugging tips and tricks. Topics of breakpoints and debugger commands are briefly covered, and the chapter concludes with a look into profiling apps using the Time Profiler and memory analysis using Leaks. By the end of this chapter, you will have a clear foundation on how to troubleshoot and debug iOS apps on both the simulator and the device.

About the Sample Code

Each chapter of this book is designed to stand by itself; therefore, each chapter with the exception of Chapter 26, "Debugging and Instruments," has its own sample project. Chapter 3, "Leaderboards," and Chapter 4, "Achievements," share a base sample project, but each expands

on that base project in unique ways. Each chapter provides a brief introduction to the sample project and walks the reader through any complex sections of the sample project not relating directly to the material in the chapter.

Every effort has been made to create simple-to-understand sample code, which often results in code that is otherwise not well optimized or not specifically the best way of approaching a problem. In these circumstances the chapter denotes where things are being done inappropriately for a real-world app. The sample projects are not designed to be standalone or finished apps; they are designed to demonstrate the functionality being discussed in the chapter. The sample projects are generic with intention; the reader should be able to focus on the material in the chapter and not the unrelated sample code materials. A considerable amount of work has been put into removing unnecessary components from the sample code and condensing subjects into as few lines as possible.

Many readers will be surprised to see that the sample code in the projects is built with Objective-C instead of Swift; this is by design as well. Since all the APIs illustrated are built with Objective-C, it is easier to interact with them using Objective-C, rather than add an additional layer of complexity by using Swift. The concepts illustrated are easily portable to Swift after the reader is comfortable with developing in Swift. The sample code is prefixed with "ICF" and most, but not all, sample projects are named after the chapter title.

When you're working with the Game Center chapters, the bundle ID is linked to a real app, which is in our personal Apple account; this ensures that examples continue to work. It also has the small additional benefit of populating multiple users' data as developers interact with the sample project. For chapters dealing with iCloud, Push Notifications, and Passbook, the setup required for the apps is thoroughly described in the chapter, and must be completed using a new App ID in the reader's developer account in order to work.

Getting the Sample Code

You will be able to find the most up-to-date version of the source code at any moment at https://github.com/dfsw/icf, in the Mastering folder. The code is publicly available and open source. Each chapter is broken down into its own zip file containing an Xcode project; there are no chapters with multiple projects. We encourage readers to provide feedback on the source code and make recommendations so that we can continue to refine and improve it long after this book has gone to print.

Installing Git and Working with GitHub

Git is a version control system that has been growing in popularity for several years. To clone and work with the code on GitHub, you will want to first install Git on your Mac. A command-line version Git is included in the Xcode command-line tool installation, or a current installer for Git can be found at http://git-scm.com/downloads. Additionally, there are several GUI front ends for Git, even one written by GitHub, which might be more appealing to developers who avoid command-line interfaces. If you do not want to install Git, GitHub also allows for downloading the source files as a zip.

GitHub enables users to sign up for a free account at https://github.com/signup/free. After Git has been installed, from the terminal's command line `$git clone git@github.com:dfsw/icf.git` will download a copy of the source code into the current working directory. The sample code for this version of the book is in the Mastering folder. You are welcome to fork and open pull requests with the sample code projects.

Contacting the Authors

If you have any comments or questions about this book, please drop us an e-mail message at mastering.ios.frameworks@gmail.com, or on Twitter at @kylerichter and @jwkeeley.

Acknowledgments

This book could not have existed without a great deal of effort from far too many behind-the-scenes people; although there are only two authors on the cover, dozens of people were responsible for bringing this book to completion. We would like to thank Trina MacDonald first and foremost; without her leadership and her driving us to meet deadlines, we would never have been able to finish. The editors at Pearson have been exceptionally helpful; their continual efforts show on every page, from catching our typos to pointing out technical concerns. The dedicated work of Niklas Saers, Olivia Basegio, Justin Williams, Sheri Replin, Elaine Wiley, Cheri Clark, Chuti Prasertsith, and Gloria Shurick made the following pages possible.

We would also like to thank Jordan Langille of Langille Design (http://jordanlangille.com) for providing the designs for the Whack-a-Cac game featured in Chapters 3 and 4. His efforts have made the Game Center sample projects much more compelling.

The considerable amount of time spent working on this book was shouldered not only by us but also by our families and co-workers. We would like to thank everyone who surrounds us in our daily lives for taking a considerable amount of work off of our plates, as well as understanding the demands that a project like this brings.

Finally, we would like to thank the community at large. All too often we consulted developer forums, blog posts, and associates to ask questions or provide feedback. Without the hard efforts of everyone involved in the iOS community, this book would not be nearly as complete.

About the Authors

Kyle Richter is the Chief Executive Officer at MartianCraft, an award-winning Mobile Development Studio. Kyle began developing software in the early 1990s and has always been dedicated to the Apple ecosystem. He has authored and coauthored several books on iOS development, including *Beginning iOS Game Center Development, Beginning Social Game Development,* and *iOS Components and Frameworks*. Between running day-to-day operations at MartianCraft, Kyle travels the world speaking on development and entrepreneurship. He currently calls the Florida Keys home, where he spends his time with his border collie. He can be found on Twitter at @kylerichter.

Joe Keeley is a Partner and Lead Engineer at MartianCraft. Joe provides technical leadership on iOS projects for clients, and has led a number of successful client projects to completion. He has liked writing code since first keying on an Apple II, and has worked on a wide variety of technology and systems projects in his career. Joe has presented several technical topics at iOS and Mac conferences around the U.S. Joe lives in Denver, Colorado, with his wife and two daughters, and hopes to get back into competitive fencing again in his spare time. He can be reached on Twitter at @jwkeeley.

This page intentionally left blank

9

Working with and Parsing JSON

JSON is a great way to send data back and forth between servers, Web sites, and iOS apps. It is lighter and easier to handle than XML, and with iOS's built-in support for JSON, it is easy to integrate into an iOS project. Many popular Web sites, including Flickr, Twitter, and Google, offer APIs that provide results in JSON format, and many languages offer JSON support. This chapter demonstrates how to parse and present JSON from a sample message-board server in an app, and encode a new message entry in JSON to send to the server.

JSON

JavaScript Object Notation (JSON) is a lightweight format for sharing data. It is technically a part of the language JavaScript and provides a way to serialize JavaScript objects; however, practically, it is supported in a wide variety of programming languages, making it a great candidate for sharing data between different platforms. JSON also has the benefit of being human-readable.

JSON has a simple and intuitive syntax. At its most basic level, a JSON document can contain *objects,* which are essentially key-value dictionaries like what Objective-C programmers are familiar with, or arrays. JSON can contain arrays of objects and arrays of values, and can nest arrays and objects. Values stored in JSON, either in arrays or associated with a key, can be other JSON objects, strings, numbers, or arrays, or `true`, `false`, or `null`.

Benefits of Using JSON

There are many reasons to use JSON in an iOS app:

- **Server Support:** Communicating information to and from a remote server is a common use case for iOS apps. Since so many server languages have built-in support for JSON, it is a natural choice as a data format.

- **Lightweight:** JSON has little formatting overhead when compared to XML and can present a significant savings in the amount of bandwidth needed to transmit data between a server and a device.
- **iOS Support:** JSON is now fully supported as of iOS 5 with the addition of the `NSJSONSerialization` class. This class can conveniently provide an `NSDictionary` or `NSArray` (or even mutable varieties) from JSON data or can encode an `NSDictionary` or `NSArray` into JSON.
- **Presentation and Native Handling:** The simplest method to get data from a server to an iOS device is just to use a `UIWebView` and display a Web page; however, this approach has drawbacks in terms of performance and presentation. In many cases it is much better to just pull the data from the server, and present it on the device using native tools like `UITableView`. Performance can be much better, and presentation can be optimized to work on iOS screen sizes and take advantage of available retina displays.

JSON Resources

For more information on JSON, visit http://json.org. That site has a formal definition of JSON, with specific information on format and syntax.

The Sample App

The sample app for this chapter is Message Board, including a Ruby on Rails server and an iOS app.

The Ruby on Rails server consists of just one object: the message. It has been set up to support sending a list of messages in JSON, and to accept new messages in JSON format. The server also supports Web-based interactions.

The iOS app will pull messages from the server and display them in a standard table view and will be able to post new messages to the server in JSON format.

Accessing the Server

To view the Message Board Ruby on Rails server, visit http://freezing-cloud-6077.herokuapp.com/. The Messages home screen will be visible, as shown in Figure 9.1.

The messages server has been set up to handle creating and displaying messages on the Web and with JSON.

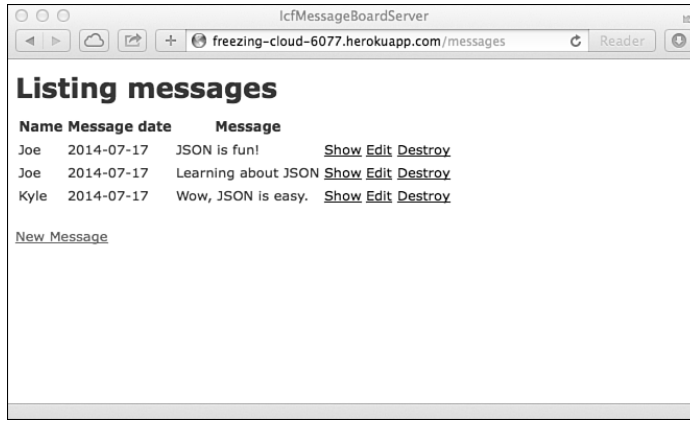

Figure 9.1 Messages home screen.

Getting JSON from the Server

To update the sample iOS app to handle JSON, the first thing to address is pulling the message list from the server and displaying it.

Building the Request

First, set up the URL so that the app can make calls to the right location:

```
NSString *const kMessageBoardURLString =
➥@"http://freezing-cloud-6077.herokuapp.com/messages.json";
```

In the `ICFViewController.m` implementation, look at the `viewWillAppear:` method. This code will initiate the request to the server:

```
NSURL *msgURL = [NSURL URLWithString:kMessageBoardURLString];
NSURLSession *session = [NSURLSession sharedSession];

NSURLSessionTask *messageTask = [session dataTaskWithURL:msgURL
 ➥completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {
 ...
}];
[messageTask resume];
```

This creates and initiates a network request to the `messages.json` resource at the server URL. The network request will run asynchronously, and when data comes back the completion handler block will be called. The important thing to note is that nothing special is required here for JSON; this is a standard network call. The only difference is that the `.json` extension

used in the URL tells the server that the response should be in JSON format. Other servers might use a `Content-Type` and/or `Accept` HTTP header that specifies `application/json` as the mime-type to indicate that a JSON response is desired.

> **Note**
>
> Using the `.json` extension is not required for servers to return JSON format data; that is just how the sample server was set up. It is a common approach but is not required.

Inspecting the Response

When the network request has returned, the completion handler will be called. In the sample app, the data is converted into a UTF-8 string so that it can be logged to the console. This should not be done for every request in a production app; it is done here to demonstrate how to see the response for debugging when a problem parsing JSON is encountered.

```
NSString *retString =
➥ [NSString stringWithUTF8String:[data bytes]];

NSLog(@"json returned: %@", retString);
```

The log message will display on the console the data received:

```
json returned: [{"message":{"created_at":"2012-04-29T21:59:28Z",
"id":3, "message":"JSON is fun!", "message_date":"2012-04-29",
"name":"Joe","updated_at":"2012-04-29T21:59:28Z"}},
{"message":{"created_at":"2012-04-29T21:58:50Z","id":2,
"message":"Learning about JSON", "message_date":"2012-04-
29","name":"Joe", "updated_at":"2012-04-29T21:59:38Z"}},
{"message":{"created_at":"2012-04-29T22:00:00Z","id":4,
"message":"Wow, JSON is easy.", "message_date":"2012-04-
29","name":"Kyle", "updated_at":"2012-04-29T22:00:00Z"}},
{"message":{"created_at":"2012-04-29T22:46:18Z","id":5,
"message":"Trying a new message.", "message_date":"2012-04-
29","name":"Joe", "updated_at":"2012-04-29T22:46:18Z"}}]
```

Parsing JSON

Now that JSON has been received from the server, it is just a simple step to parse it. In the case of the sample app, an array of messages is expected, so parse the JSON into an `NSArray`:

```
NSError *parseError = nil;
NSArray *jsonArray =
➥ [NSJSONSerialization JSONObjectWithData:data
                                   options:0
                                     error:&parseError];
```

```
if (!parseError) {
    [self setMessageArray:jsonArray];
    NSLog(@"json array is %@", jsonArray);
} else {
    NSString *err = [parseError localizedDescription];
    NSLog(@"Encountered error parsing: %@", err);
}
```

`NSJSONSerialization`'s method `JSONObjectWithData:options:error:` expects as parameters the data to be serialized, any desired options (for example, returning a mutable array instead of a regular array), and a reference to an `NSError` in case there are any parsing errors.

In this example, a local instance variable has been updated to the just-parsed array, the table view has been told to reload data now that there is data to display, and the activity view has been hidden. Note that the completion handler will most likely be called on a background queue, so if the user interface will be updated, it will be necessary to switch to the main queue.

```
dispatch_async(dispatch_get_main_queue(), ^{
    [self.messageTable reloadData];
    [self.activityView setHidden:YES];
    [self.activityIndicator stopAnimating];
});
```

Displaying the Data

Now that the JSON has been parsed into an `NSArray`, it can be displayed in a `UITableView`. The magic here is that there is no magic; the JSON received from the server is now just an array of `NSDictionary` instances. Each `NSDictionary` contains information for a message from the server, with attribute names and values. To display this in a table, just access the array and dictionaries as if they had been created locally.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
    ➥ [tableView dequeueReusableCellWithIdentifier:@"MsgCell"];

    if (cell == nil) {
        cell = [[UITableViewCell alloc]
                ➥initWithStyle:UITableViewCellStyleSubtitle
                ➥reuseIdentifier:@"MsgCell"];

        cell.selectionStyle = UITableViewCellSelectionStyleNone;
    }
    NSDictionary *message =
    ➥ (NSDictionary *)[[self.messageArray
                        ➥objectAtIndex:indexPath.row]
                        ➥objectForKey:@"message"];
```

```
    NSString *byLabel =
    ➥ [NSString stringWithFormat:@"by %@ on %@",
    ➥ [message objectForKey:@"name"],
    ➥ [message objectForKey:@"message_date"]];

    cell.textLabel.text = [message objectForKey:@"message"];
    cell.detailTextLabel.text = byLabel;
    return cell;
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [[self messageArray] count];
}
```

The parsed JSON data will be visible in a standard table view, as shown in Figure 9.2.

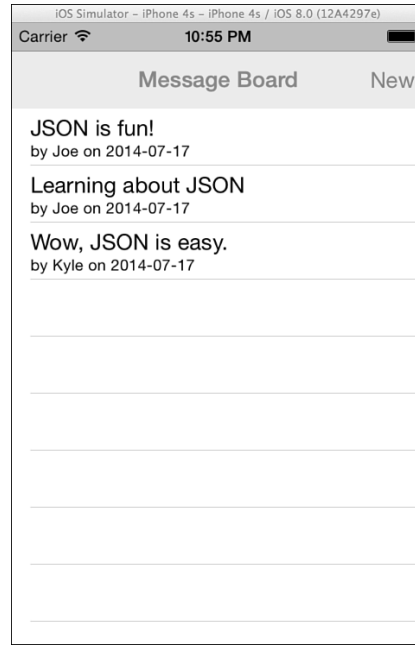

Figure 9.2 Sample app message table view.

> **Tip**
>
> When a `null` value is in the JSON source data, it will be parsed into an `[NSNull null]`. This can be a problem if `nil` is expected in a check or comparison, because `[NSNull null]` will return `YES` whereas `nil` will return `NO`. It is wise to specifically handle `[NSNull null]` when converting to a model object or presenting parsed JSON.

Posting a Message

The sample app includes `ICFNewMessageViewController` to post new messages to the server. There are two fields on that controller: one for a name and one for a message (see Figure 9.3). After the user enters that information and hits Save, it will be encoded in JSON and sent to the server.

Figure 9.3 Sample app new message view.

Encoding JSON

An important detail for sending JSON to a Ruby on Rails server is to encode the data so that it mirrors what the Rails server provides. When a new message is sent to the server, it should have the same structure as an individual message received in the message list. To do this, a dictionary with the attribute names and values for the message is needed, and then a wrapper

dictionary with the key "message" pointing to the attribute dictionary. This will exactly mirror what the server sends for a message. In the `saveButtonTouched:` method, set up this dictionary, like so:

```
NSMutableDictionary *messageDictionary =
➥ [NSMutableDictionary dictionaryWithCapacity:1];

[messageDictionary setObject:[nameTextField text]
                      forKey:@"name"];

[messageDictionary setObject:[messageTextView text]
                      forKey:@"message"];

NSDate *today = [NSDate date];

NSDateFormatter *dateFormatter =
➥ [[NSDateFormatter alloc] init];

NSString *dateFmt = @"yyyy'-'MM'-'dd'T'HH':'mm':'ss'Z'";
[dateFormatter setDateFormat:dateFmt];
[messageDictionary setObject:[dateFormatter stringFromDate:today]
                      forKey:@"message_date"];


NSDictionary *postDictionary = @{@"message" : messageDictionary};
```

Note that `NSJSONSerialization` accepts only instances of `NSDictionary`, `NSArray`, `NSString`, `NSNumber`, or `NSNull`. For dates or other data types not directly supported by `NSJSONSerialization`, they will need to be converted to a supported format. For example, in this example the date was converted to a string in a format expected by the server. Now that there is a dictionary, it is a simple step to encode it in JSON:

```
NSError *jsonSerializationError = nil;
NSData *jsonData = [NSJSONSerialization
                    ➥dataWithJSONObject:postDictionary
                    ➥options:NSJSONWritingPrettyPrinted
                    ➥error:&jsonSerializationError];

if (!jsonSerializationError)
{
    NSString *serJSON =
    [[NSString alloc] initWithData:jsonData
                          encoding:NSUTF8StringEncoding];

    NSLog(@"serialized json: %@", serJSON);
        ...
} else
{
```

```
    NSLog(@"JSON Encoding failed: %@",
          ➥ [jsonSerializationError localizedDescription]);
}
```

`NSJSONSerialization` expects three parameters:

1. An `NSDictionary` or `NSArray` with the data to be encoded.
2. Serialization options (in our case, we specified `NSJSONWritingPrettyPrinted` so that it's easy to read; otherwise, the JSON is produced with no whitespace for compactness).
3. A reference to an `NSError`.

If there are no errors encoding the JSON, it will look like this:

```
serialized json: {
  "message" : {
    "message" : "Six Test Messages",
    "name" : "Joe",
    "message_date" : "2012-04-01T14:31:11Z"
  }
}
```

Sending JSON to the Server

After the JSON is encoded, it is ready to be sent to the server. First, an instance of `NSMutableURLRequest` is needed. The request will be created with the URL for the server, and then will be customized with the HTTP method (`"POST"`) and HTTP headers to indicate that the uploaded content data is in JSON format.

```
NSURL *messageBoardURL =
➥ [NSURL URLWithString:kMessageBoardURLString];

NSMutableURLRequest *request = [NSMutableURLRequest
                                requestWithURL:messageBoardURL
                            cachePolicy:NSURLRequestUseProtocolCachePolicy
                        timeoutInterval:30.0];

[request setHTTPMethod:@"POST"];

[request setValue:@"application/json"
forHTTPHeaderField:@"Accept"];

[request setValue:@"application/json"
forHTTPHeaderField:@"Content-Type"];
```

When the request is completed, an `NSURLSessionUploadTask` can be created. The task requires the request, the JSON data, and a completion handler. The completion handler will be called on a background thread, so any user interface updates must be dispatched to the main queue.

```
NSURLSession *session = [NSURLSession sharedSession];

NSURLSessionUploadTask *uploadTask =
[session uploadTaskWithRequest:uploadRequest fromData:jsonData
 ➥ completionHandler:^(NSData *data, NSURLResponse *response, NSError *error) {

    NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
    BOOL displayError = (error || httpResponse.statusCode != 200);

    dispatch_async(dispatch_get_main_queue(), ^{
        [self.activityView setHidden:YES];
        [self.activityIndicator stopAnimating];
        if (displayError) {
            NSString *errorMessage = error.localizedDescription;
            if (!errorMessage) {
                errorMessage =
                ➥ [NSString stringWithFormat:@"Error uploading - http status: %i",
                ➥ httpResponse.statusCode];
            }

            UIAlertController *postErrorAlertController =
            ➥ [UIAlertController alertControllerWithTitle:@"Post Error"
                                                   message:errorMessage
                                            preferredStyle:UIAlertControllerStyleAlert];

            [postErrorAlertController addAction:
             ➥ [UIAlertAction actionWithTitle:@"Cancel"
                                         style:UIAlertActionStyleCancel
                                       handler:nil]];

            [self presentViewController:postErrorAlertController
                               animated:YES
                             completion:nil];
        } else {
            [self.presentingViewController dismissViewControllerAnimated:YES
                                                                completion:nil];
        }
    });

}];
[uploadTask resume];
```

When `resume` is called on the `uploadTask`, the request will be made to the server, and the completion handler will be called when it is complete. Both the `error` returned in the completion handler and the `response` should be checked for errors; an `error` will be returned if there is a problem connecting (for example, the device is in airplane mode), or an HTTP status code might indicate a different problem if there is an issue on the server (for example, if the URL is

not found, or if the server cannot process the data sent to it). If the request completes with no errors, the view controller will be dismissed and message board will be refreshed.

Summary

This chapter introduced JavaScript Object Notation (JSON). It explained how to request JSON data from a server in an iOS app, parse it, and display it in a table. The chapter also described how to encode an `NSDictionary` or `NSArray` into JSON, and send it over the network to a server.

This page intentionally left blank

Index

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V - W

X

Y

Z