



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

RestKit for iOS

Link your apps and web services using RestKit

Taras Kalapun

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

RestKit for iOS

Link your apps and web services using RestKit

Taras Kalapun

[PACKT] open source 
PUBLISHING community experience distilled
BIRMINGHAM - MUMBAI

RestKit for iOS

Copyright © 2013 Packt Publishing

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

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

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

First published: September 2013

Production Reference: 1190913

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-370-1

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author

Taras Kalapun

Project Coordinator

Joel Goveya

Reviewers

Angel Garcia Olloqui

Anthony Shoumikhin

Vladimir Pouzanov

Proofreader

Jonathan Todd

Indexer

Monica Ajmera Mehta

Acquisition Editors

Usha Iyer

Julian Ursell

Graphics

Ronak Dhruv

Commissioning Editor

Govindan K.

Production Coordinator

Kirtee Shingan

Technical Editors

Akashdeep Kundu

Krishnaveni Nair

Larissa Pinto

Cover Work

Kirtee Shingan

About the Author

Taras Kalapun has more than 10 years experience as a Software Developer and Consultant in Mobile and Web. His background spans numerous technologies, programming languages, and databases. He was involved in developing more than 100 iOS projects and web services, as well as project management activities. In addition, he managed teams of software developers who all wanted to kill him. Through his mentoring, tech leading, troubleshooting, and code reviewing, he discovered that teaching by example resulted in more effective software development. A method he supplemented with, "Stop trying to reinvent the wheel" – a favorite phrase he used to tell young software developers.

He has worked at a number of IT companies across Europe including Ukrtelecom, a national Ukrainian telecommunication company; Ciklum, a Dutch outstuffing company headquartered in Ukraine; and Xaton, an Amsterdam software development company, in addition to freelance projects.

Occasionally he publishes small how-to articles on solving development problems and impedances on his blog, <http://kalapun.com>, some of which progressed to the development of this book.

I would like to thank Mattt Thompson for creating the AFNetworking library and Blake Watters for writing the RestKit framework. Also, I would like to thank my friends and co-workers who helped me by reviewing the book.

About the Reviewers

Angel G. Olloqui is a computer engineer with a Master's degree in Web technologies. He specializes in mobile application development, with experience in using Agile methodologies, mainly SCRUM and some XP practices.

He started his career at Oracle but he quickly decided to start up his own company (Wixel Solutions). After a couple of years of entrepreneurship, when the first iOS SDK came live, he moved into the mobile sector by joining Mobivery. In Mobivery, he performed as the main developer and SCRUM master of the mobile team in Madrid.

By the end of 2011, after a short period of nine months in San Francisco (USA), he decided to continue his career back in Europe. Nowadays he lives in Amsterdam and works at Xaton as a senior iOS developer, participating in some of the best-known iOS apps in the Netherlands.

Vladimir Pouzanov is systems engineer and mobile development enthusiast.

He spent countless hours hacking on different mobile hardware, porting Linux to Palm® devices, and toying outside the iPhone sandbox. He has been doing professional iOS development and consultancy since the first Apple iPhones were available. Later on, he switched his professional interest to systems management and engineering, but he keeps a close eye on mobile and the embedded world of iPhones, Android devices, and Arduino-based gadgets.

Anthony Shoumikhin is one of those geeks who's jumped from low-level C++ system programming to a new and exciting world of mobile technologies, and has never regretted it.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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

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



<http://PacktLib.PacktPub.com>

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: Getting Started	5
Simple usage example	5
So what is RestKit?	7
Why RestKit?	8
RestKit components	9
How it works?	10
Adding RestKit and libraries	11
MongoHQ – a MongoDB in clouds	15
Trying basic stuff	16
Object mapping fundamentals	19
Data modeling	20
Summary	22
Chapter 2: Modeling and Loading Remote Objects	23
Object Manager	23
Sending requests with object manager	28
Integrating with UI and DRYing the table view	32
Requesting related objects	36
RESTful object manipulation	39
Routing inside out	43
Entering data in forms	46
Summary	53
Chapter 3: Persistence with Core Data	55
Setting up a database	55
Database	56
Collection	56
Configuring	57
Magical Record	58

Table of Contents

Mogenerator	60
Mapping	62
Integrating with UI	64
Database seeding	66
Indexing and searching	68
Summary	71
Chapter 4: Advanced Stuff	73
Reachability	73
Logging	75
Error mapping	78
Metadata mapping	79
Advanced object mapping techniques	81
Batching operations	82
Paginating results	84
Authorization	86
Basic	86
Token-based	86
OAuth 1.0	87
OAuth 2.0	88
SSL and certificates	88
HTTP caching	89
Background processing	92
Custom HTTP client	93
Summary	94
Appendix A	95
Appendix B	97
Index	101

Preface

You can ask yourself the question, "Why would I read the book about this library?" Of course, you can read the RestKit documentation. It's a perfect example of how to write down a good API reference for a framework. On the other hand, you can dive into researching the RestKit test cases, which is also nice and interesting, as the unit tests are covering almost all the functionalities of a framework. However, it won't make you happy.

While reading the API documentation is nice and it is full of simple examples for every class, it won't help you understand how to easily apply the framework in your daily jobs. Therefore, the key goal of this book is to provide guidance and real-life usage examples. We will start from a basic example of how to load a simple list of objects to creating a full-featured app, with advanced mapping techniques, Core Data integration, and so on.

What this book covers

Chapter 1, Getting Started, starts with the key concepts of RestKit, describes how to install it, and shows a short example of basic data loading.

Chapter 2, Modeling and Loading Remote Objects, describes how to configure RestKit and data mapping, how to use one-line methods to load objects and push them back to the server, and how to integrate the code with nice UI.

Chapter 3, Persistence with Core Data, describes how to make RestKit persist and synchronize data with Core Data databases.

Chapter 4, Advanced Stuff, covers some more advanced features of RestKit and AFNetworking libraries that developers might use in everyday app development and will help overcome potential bottlenecks.

Appendix A, Helpful Resources, lists links to nice guides that might be useful to a developer.

Appendix B, Helpful Libraries, lists links and describes some popular libraries that might be useful for developing iOS apps.

What you need for this book

A free Apple iOS Developer account with Xcode 4.5 running in OS X, and an Internet connection for downloading the RestKit and related libraries.

Who this book is for

iOS developers of all levels who are interested in boosting their productivity by using third-party libraries with a willingness to learn how to build RESTful apps using the RestKit framework. Basic knowledge of Objective-C is required and a simple understanding of how Core Data works.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "It heavily uses tables (such as `UITableViewController`) to present the loaded data (list of objects) to the user."

A block of code is set as follows:


```
// in MDatabase @implementation
- (NSString *)titleText
{
    return self.name;
}


- (NSString *)subtitleText
{
    return self.plan;
}
```

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

```
[self setupObjectManager];  
  
// Setup CoreData stack after Object Manager  
[self setupCoreData];  
[self setupMappings];
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "A **Save** button is added to the navigation controller, it triggers the `saveAction` method with the following code."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting Started

RestKit is a well-known framework without much documentation. Even with its reference manual and accompanying blog posts, not much has been covered, especially in terms of practical usage. Not every developer has the time to indulge in figuring out RestKit on his own before starting a time-constrained project. Sounds familiar?

Learning a new framework comes with its required steps. Before jumping into RestKit libraries, object mapping fundamentals, and data modeling, we need to make the proper introductions. This chapter will start with a simple usage example to warm up the crowd, before elaborating on the whats, whys, and hows of RestKit, in addition to its components. This compact introduction will already have demonstrated how select real-life examples can provide the required insight into the world of RestKit.

As you know, nothing in this world is perfect, and so are REST APIs. Every single API I worked with has its own glitches and bottlenecks. So, we will discuss some of the possible bottlenecks with APIs, how to overcome them, and we will experience a few in the API that we will use in the example.

Simple usage example

We can show a simple example of using RestKit by loading this kind of JSON:

```
[
  {
    "hostname": "sandbox.mongohq.com",
    "name": "Test",
    "plan": "Sandbox",
    "port": 10097,
    "shared": true
  },
  {
    "hostname": "second.mongohq.com",
```

```
        "name": "Second",
        "plan": "Second",
        "port": 10097,
        "shared": true
    }
]
```

And mapping it to a list of database objects:

```
@interface MDatabase : NSObject
@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *plan;
@property (nonatomic, strong) NSString *hostname;
@property (nonatomic, strong) NSNumber *port;
@end
```

Would be just invoking this piece of code:

```
RKObjectManager *manager = [RKObjectManager sharedManager];
[manager getObjectsAtPath:@"/databases"
    parameters:nil
    success:^(RKObjectRequestOperation *operation,
    RKMappingResult *mappingResult)
    {
        NSLog(@"Loaded databases: %@", [mappingResult array]);
    }
    failure:^(RKObjectRequestOperation *operation,
    NSError *error)
    {
        NSLog(@"Error on loading: %@", [error localizedDescription])
    }
];
```

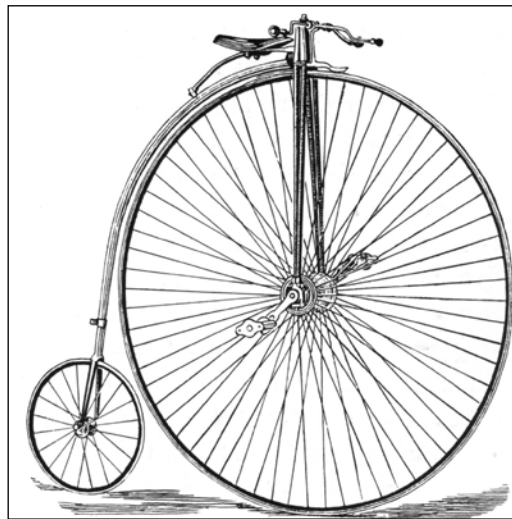
As we see here, not much code. Of course, we will need some additional pre-setup to get this working, which we will cover later in this and the following chapter.

So what is RestKit?

"I once told an Objective-C joke, but nobody got the message."

Many of us are being introduced with the concept of networking in iOS and Objective-C by playing with `NSURLConnection`. It's a base class for making any outgoing HTTP connection. However, after using it several times in a project, you may start building your personal networking library, of course, under the hood; it's still the same `NSURLConnection`, but most likely, you would write your own wrappers for it with additional bells and whistles. In addition, while using it, you may experience different bugs, glitches, and performance issues. Also, while using it in different projects, you will probably modify the code, find and fix bugs in it, but you will experience a lot of hassles in maintaining a "one codebase" of such a homegrown library.

So my personal opinion is "Don't re-invent the wheel. Don't re-invent a bicycle. Unless you really need a very custom one, with a big front wheel (as shown in the next figure), which might happen only in 1 percent of cases."



Try to keep things simple. In our case, the working bicycle can be a library called **AFNetworking**, which is a very useful networking library for iOS and OS X. The framework is built on top of Apple's Foundation technologies such as `NSURLConnection`, `NSOperation`, and others. It has a modular architecture and a well-designed API, which is quite easy to use. While being simple and easy-to-use, it also supports block-based programming, file uploads, reachability, and lots of other useful tasks that any developer might need. It is used by thousands of app developers and is highly maintained.

Now RestKit itself is a framework for implementing clients of RESTful web services, and feels like AFNetworking is on steroids. It provides a simple interface to build network communication while mapping HTTP requests and responses. Additionally, it has a powerful object-mapping engine that seamlessly integrates with the Core Data. It has an elegant, well-designed, fully documented, and fully tested set of APIs magically allowing easy accessing and modeling RESTful resources.

It greatly simplifies the application development by providing a solution by interconnecting your application's data model with JSON or XML documents, provided by a web service you are communicating with. It shifts a lot of logic for making requests and mapping the data to the library, thus giving a developer the ability to keep an application code simpler and less cluttered. It speeds up the development process by giving solutions and patterns for common problems that one can face, such as working with Core Data and doing network-related coding.

When thinking whether to use a RestKit library in your next project, consider a few things:

- Not every API will work with a RestKit library, especially the ones that follow the RPC (Remote Procedure Call) paradigm.
- Your backend web-service API is more or less RESTful. You can describe interactions with your web application with the CRUD (Create, Read, Update, and Delete) operations on resources.
- "Don't use a sledgehammer to crack a nut." If you just need a one-time request to a simple JSON in your app, reconsider using a RestKit library in favor of using something simple, such as AFNetworking.
- While developing a library, think if you can minimize a footprint of it and exclude dependencies on big third-party libraries, such as RestKit.

Why RestKit?

We can compare RestKit to some other popular or similar solutions:

- **AFIncrementalStore** (<https://github.com/AFNetworking/AFIncrementalStore>): A new library from the creator of AFNetworking that works in a tight connection with Core Data. It is not yet very customizable, but still in its early development and has some possible bugs/performance issues.
- **MagicalRecord mappers** (<https://github.com/magicalpanda/MagicalRecord>): It is still in development, not well-documented, and not actively used by developers.
- **Parse** (<https://www.parse.com>): Can be used only with their web services.

- **KeyValueObjectMapping** (<https://github.com/dchohfi/KeyValueObjectMapping>): It is a small new library that is not documented yet, not fully-tested, and possibly has some performance issues.
- **SLRESTfulCoreData** (<https://github.com/OliverLetterer/SLRESTfulCoreData>): It is new and documentation is a bit confusing, and the code is not documented.
- **Mantle** (<https://github.com/github/Mantle/>): An interesting library from GitHub for creating a model layer. It is designed to be used more as in-memory storage, doesn't connect easily with Core Data, and needs some experience to set up and configure every model.

As we can see here, there are numerous libraries that do mapping of objects, or simplify the development by making hidden requests while fetching data from Core Data. Some of them are quite interesting to check and study – depending on a type of project one is doing. The downside of the earlier-mentioned libraries is that these solutions are relatively new, still in active development, or quite complex to set up.

RestKit components

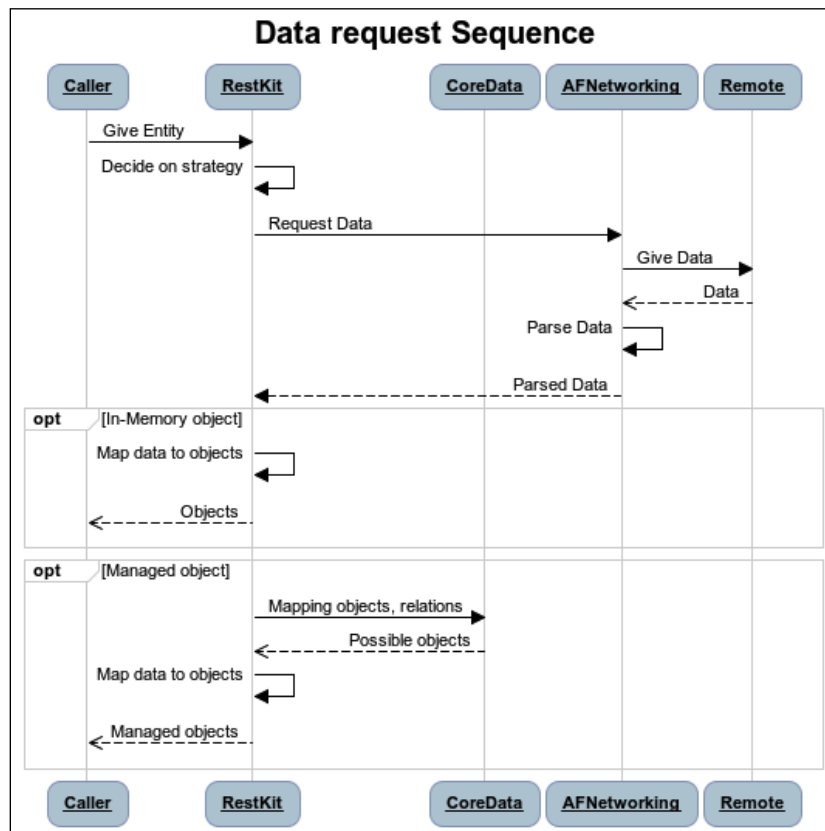
RestKit ships as a single framework to the end user, but internally, it is composed by several interconnected components. We have described the parts of RestKit in the following list:

- **Object manager:** One of the main components of a RestKit library. It works as a bridge between it and other components, and provides one-line methods for "getting the work done".
- **Object mapping:** One of the main functional components of RestKit. The object mapping system enables a mechanism to express the transformation of objects between representations using KVC and the dynamic features of the Objective-C runtime.
- **Networking:** This feature integrates object mapping with the HTTP network layer provided by AFNetworking. It has the ability to make serialization and deserialization of JSON/XML objects, bind mapping descriptions with HTTP requests and responses, generate URLs from path patterns, route, and serialize local objects into HTTP parameters.
- **Core Data:** This component is responsible for the integration between the object mapping and networking components and Apple's Core Data framework. Mostly, it includes a special Core-Data-related implementation of some parts from the object mapping and networking layers, and also specific functionality for the relationship connection of managed objects.

- **Search:** Specific parts related to Core Data components that are responsible for indexing and searching of managed objects. It includes a tokenizer, indexer, and API for generating search predicates to be used while querying indexed objects.
- **Testing:** Helps developing unit tests with ease by providing helpers and mocking abilities for RestKit components. Also includes helpers to build and use test fixtures.

How it works?

We can describe how you interact with RestKit by looking at the following sample sequence diagram for getting data:



RestKit usage sequence diagram

When you, as a **Caller**, wish to get data from a **Remote** web service, you ask for it on RestKit. It then decides on a strategy and gets the paths and mapping information by checking a configuration for the particular type of objects that you want. RestKit uses AFNetworking under the hood to do the actual data retrieving from the Remote and parsing in to `NSDictionary` or `NSArray`. AFNetworking itself checks with `NSURLCache` if it should make a request again, or use the cache. (We'll discuss this in detail in *Chapter 4, Advanced Stuff* in the *HTTP Caching* section). AFNetworking then gives back the response to a RestKit, along with all additional information, which were gathered during the "request-response-parse sequence".

If you're not using Core Data for this type of objects (we can call it "**In-Memory object**"), RestKit creates new instances of the object, and maps a response data to it. It then returns the object(s) back to the **Caller**.

Now if you are using Core Data, and the object is a **Managed object**, RestKit will first check with the Core Data if it already has an object with a similar ID. It will also check if the object in response has a `deleted` flag. It will then do the mapping and update/delete the particular object, and check how to deal with orphan objects. At the end, it will return the resultant objects to the user. Core Data itself will notify all its observers via **Key Value Observing (KVO)** about the changes.

Adding RestKit and libraries

It used to be quite hard to add third-party libraries to Mac or iOS projects. You would have to deal with all sorts of dependencies, configuring special behavior, and spend days on integrating big libraries to your project.

This was not an issue for some other platforms. C# with Visual Studio has had the NuGet package manager for quite a while. And Ruby has its RubyGems with Bundler. Recently, the situation has changed for iOS and Mac app developers. Highly inspired by Ruby's Bundler, a new package manager for us arrived – CocoaPods. It is the best way to manage library discrepancies in Objective-C projects.

Now in comparison to RubyGems' `Gemfile`, CocoaPods uses a so-called `Podfile`, where a developer lists the names of a library he is willing to use and his version. By the way, it uses `Podspec` files to describe how a particular library should be integrated with your project. Actually, CocoaPods is using an Xcode Workspace for the integration between your project and a `Pods` project, which includes all third-party libraries.

If you have never installed a CocoaPods package manager before, let's do so! You start by executing the following commands:

```
$ [sudo] gem install cocoapods
$ pod setup
```

The pod is an executable package, which is installed with CocoaPods.

Next, we want to search the spec repository using the following command to get, which version of RestKit is available as of today:

```
$ pod search RestKit
```

The result might look like this:

```
bash-3.2$ pod search RestKit
```

```
-> RestKit (0.20.3)
  RestKit is a framework for consuming and modeling RESTful web
  resources on iOS and OS X.
  pod 'RestKit', '~> 0.20.3'
  - Homepage: http://www.restkit.org
  - Source:   https://github.com/RestKit/RestKit.git
  - Versions: 0.20.3, 0.20.2, 0.20.1, 0.20.0, 0.20.0-rc1, 0.20.0-
pre6, 0.20.0pre5, 0.20.0-pre4, 0.20.0-pre3, 0.20.0-pre2, 0.20.0-pre1,
0.10.3, 0.10.2, 0.10.1, 0.10.0 [master repo]
  - Sub specs:
    - RestKit/Core (0.20.3)
    - RestKit/ObjectMapping (0.20.3)
    - RestKit/Network (0.20.3)
    - RestKit/CoreData (0.20.3)
    - RestKit/Testing (0.20.3)
    - RestKit/Search (0.20.3)
    - RestKit/Support (0.20.3)
```

In addition, you can also hit your browser to view the official website of CocoaPods (<http://CocoaPods.org>), where you will be able to use the web-based search and get more info on the available packages and CocoaPods news.

Now change to the directory of your Xcode project, and create (or edit) your Podfile with your favorite text editor and add RestKit (or create it using Xcode if you like):

```
$ cd /path/to/MyProject
$ nano Podfile
```

```
# Platform - ios, mac
```

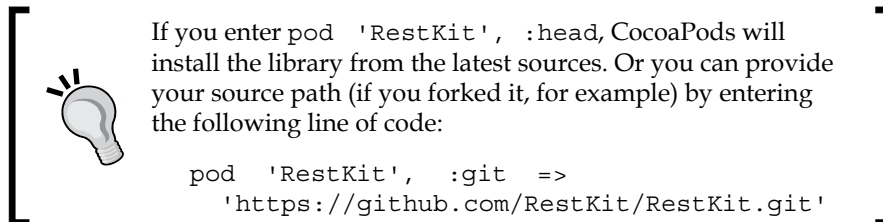
```
platform :ios, '5.1'

# List of libraries to install
pod 'RestKit', '~> 0.20.3'

# Testing and Search are optional components
pod 'RestKit/Testing'
pod 'RestKit/Search'
```

By specifying the platform and its version we want to be sure that all libraries and dependencies we are using will smoothly run on the target.

Now with RestKit, `~> 0.20.3` in the previous command-line snippet, means we want to use at least Version 0.20.3 or advanced in the range of 0.20.X. If you skip specifying the version, CocoaPods will install the latest.



Now it's time to install it in your project. Just run the following command:


```
$ pod install
```

And you will probably see the following output:

```
bash-3.2$ pod install
Analyzing dependencies
Downloading dependencies
Installing AFNetworking (1.3.2)
Installing RestKit (0.20.3)
Installing SOCKit (1.1)
Installing TransitionKit (1.1.1)
Generating Pods project
Integrating client project
```

CocoaPods downloads the third-party code and creates a new workspace—a file named `YourProject.xcworkspace`. From now on, you will use the `.xcworkspace` file to open your project in Xcode.

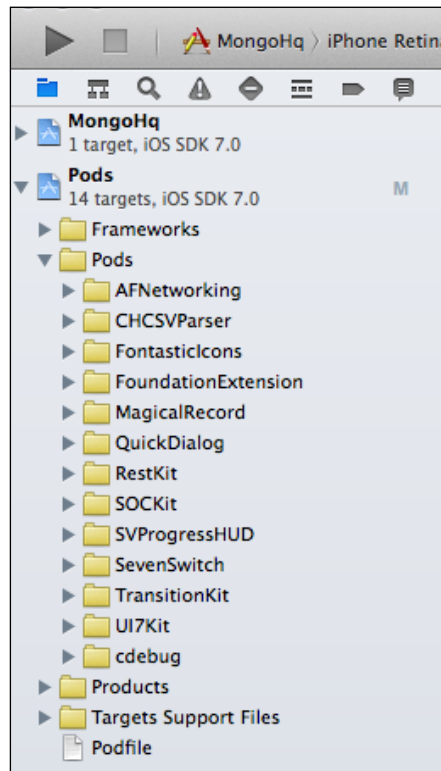
As you can also see, CocoaPods will install some libraries (such as SOCKit) that we did not ask for. They are dependencies of a RestKit library, and are specified in a RestKit's Podspec file.

[ Run `pod update` to fetch and install the latest versions of packages. In addition, running `[sudo] gem update cocoapods` will update the CocoaPods manager to the latest version.]

Once again, do not forget that you need to use workspace from now on. So to open your project in Xcode, one can shoot the following command in the terminal:

```
$ open MyProject.xcworkspace
```

The following screenshot shows the sample project tree in the workspace after installing RestKit and few other libraries.



Project tree in Xcode after installing a few libraries using CocoaPods

Add the following line to your `.pch` file (it is a precompiled header file, which is automatically included in all source files of your project) to be able to use all RestKit components through your code:

```
#import <RestKit/RestKit.h>
```



If you've worked previously with RestKit 0.10, you should know that with the release of Version 0.20, it had major API changes, which are backwards incompatible. Consider checking RestKit's Wiki article *Upgrading from v0.10.x to v0.20.0* in the following link:

<https://github.com/RestKit/RestKit/wiki/Upgrading-from-v0.10.x-to-v0.20.0>

Please note that if your installation fails, it may be because you are installing with a version of Git lower than what CocoaPods is expecting. Please ensure that you are running Git 1.8.0 or higher by executing `git --version`. You can get a full picture of the installation details by executing `pod install --verbose`.

If you want to install RestKit as a Git submodule or from a release package, the best way is to follow the instructions on RestKit's Wiki article *Installing RestKit v0.20.x as a Git Submodule* in the following link:

<https://github.com/RestKit/RestKit/wiki/Installing-RestKit-v0.20.x-as-a-Git-Submodule>

MongoHQ – a MongoDB in clouds

"Three DBAs walk into a NoSQL bar. A little while later they walk out because they couldn't find a table."

For our examples in this book, we will use a service in a cloud called **MongoHQ**. It's the most powerful platform for MongoDB hosting. Apart from providing one of the best MongoDB hosting solutions, they recently released a beta REST API for accessing their services. This is quite interesting for using in mobile clients, as accessing directly a MongoDB server is not the easiest of tasks.



MongoDB (from "humongous") is an open source document database and the leading NoSQL database.

<http://en.wikipedia.org/wiki/MongoDB>

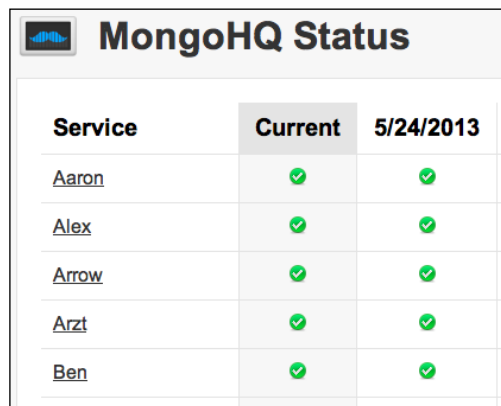
MongoDB's main difference from "classical" relational databases is that instead of storing data in tables, MongoDB stores structured data as JSON-like documents with dynamic schemas (MongoDB calls the format **BSON**), making the integration of data in certain types of applications easier and faster.

The nice part about it is if you are not sure beforehand on what your data will look like, the document-type databases are a weapon of choice. You can change the structure with ease almost on the fly; you don't need to run any migration scripts. This greatly simplifies development in the early stages and/or the startup phase.

Trying basic stuff

For our basic example, we will check the status of MongoHQ servers. For showing statuses, MongoHQ uses the Stashboard web app, which you can access at <http://status.mongohq.com>.

The following screenshot (cropped) shows the status page of the MongoHQ servers:



Service	Current	5/24/2013
Aaron	✓	✓
Alex	✓	✓
Arrow	✓	✓
Arzt	✓	✓
Ben	✓	✓

An example of Stashboard App usage is **Twilio**, with the status page at <http://status.twilio.com>, and the Status API endpoint at <http://status.twilio.com/api/v1/>.

The documentation for a Stashboard API can be found at <https://stashboard.readthedocs.org/en/latest/restapi.html>.

To keep going, first let's define our `StatusItem` object, looking at a possible data we will need:

```
// StatusItem.h
@interface StatusItem : NSObject
```

```

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *itemDescription;
@property (nonatomic, strong) NSDate *timestamp;
@property (nonatomic, strong) NSString *eventMessage;
@property (nonatomic, strong) NSString *statusName;
@property (nonatomic, strong) NSString *imageUrl;

@end

// StatusItem.m
@implementation StatusItem

// for better debug information output
- (NSString *)description
{
    return [NSString stringWithFormat:@"%@" - %@",
            self.name, self.eventMessage];
}
@end

```

Now let's try and load the current statuses:

```

// Method to load the status items
- (void)refresh
{
    // Setup the object mapping
    RKObjectMapping *mapping = [RKObjectMapping
mappingForClass:[StatusItem class]];
    // From JSON -> To property
    [mapping addAttributeMappingsFromDictionary:@{
        @"name" : @"name",
        @"description" : @"itemDescription",
        @"current-event.status.name" : @"statusName",
        @"current-event.status.image" : @"imageUrl",
        @"current-event.timestamp" : @"timestamp",
        @"current-event.message" : @"eventMessage",
    }];

    // Define the response mapping
    // Map response with any status code in 2xx
    NSIndexSet *statusCodes = RKStatusCodeIndexSetForClass(RKStatusCodeClassSuccessful);
}

```

```
RKResponseDescriptor *responseDescriptor = [RKResponseDescriptor
responseDescriptorWithMapping:mapping
method:RKRequestMethodAny
pathPattern:@"/api/v1/services"
keyPath:@"services"
statusCodes:statusCodes];

// Prepare the request operation
NSURLRequest *request = [NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://status.twilio.com/api/v1/services"]];
RKObjectRequestOperation *operation = [[RKObjectRequestOperation
alloc] initWithRequest:request responseDescriptors:@
[responseDescriptor]];

// Set on completion and on error blocks
[operation
setCompletionBlockWithSuccess:^(RKObjectRequestOperation
*operation, RKMappingResult *result)
{
NSLog(@"Loaded items: %@", [result array]);
}
failure:^(RKObjectRequestOperation
*operation, NSError *error)
{
NSLog(@"Failed with error: %@", [error localizedDescription]);
}];

[operation start]; //Fire the request
}
```

Downloading the example code



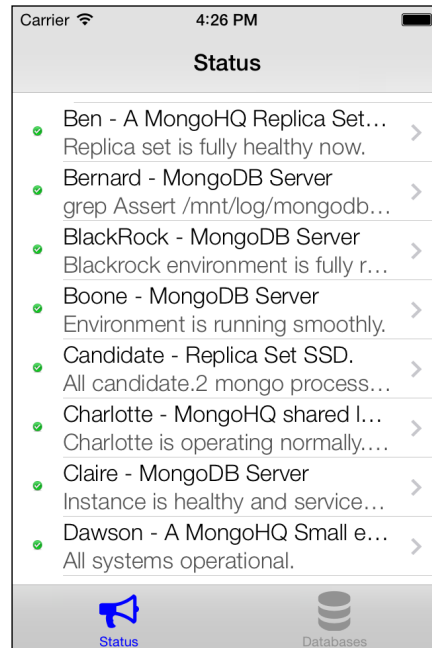
You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

In this example, we are using a `RKObjectRequestOperation` object. It performs a base operation in all RestKit requests. Even for more high-end user-friendly methods, it will be run under the hood.

When we run the example, we see the debugger output:

```
Loaded items: (
  "Aaron - All systems operational.",
  "Alex - Alex is operational.",
  "Arrow - All databases accessible and operating normally."
)
```

If we put it in a table view (we will discuss about integrating RestKit and user interface best practices in *Chapter 2, Modeling and Loading Remote Objects*), it will look like the next screenshot:



Status of DBs

Object mapping fundamentals

The object mapping engine of a RestKit is built on the KVC informal protocol, which is foundational to numerous Cocoa technologies, such as Key-Value observing, bindings, and Core Data. After the response body was parsed, RestKit relies on KVC to identify the content that can be mapped and dynamically updates the attributes and relationships of your local domain objects with the appropriate content. Before diving into the details of RestKit's object mapping system, be sure to get familiar with Apple's Key-Value Coding and go through its programming guide at the following link:

<http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/KeyValueCoding/Articles/KeyValueCoding.html>

Using a highly dynamic Objective-C runtime, RestKit examines the type of source and destination properties of object and performs appropriate type transformations. For example, when a JSON is parsed and a source key path `created_at` (with a string content) is configured to be mapped to a destination key path, `creationDate` (this is an `NSDate` property on a target object), RestKit will transform the date from a string into an `NSDate` property using an `NSDateFormatter`. The other transformations can be string to number and vice versa, or a developer can build his own transformation strategy, if needed.

The mapper also fully supports relationship mappings, where nested to-one or to-many child objects are mapped recursively.

Data modeling

Let's discuss our basic example. We were loading data from the Status API endpoint, <http://status.twilio.com/api/v1/services>. If we visit this URL with our web browser, we see a JSON output similar to the following code:

```
{
  "services": [
    {
      "current-event": {
        "informational": false,
        "message": "All systems operational.",
        "sid": "ag5tb25nb2hxLXN0YXR1c3INCxIFRXZlbnQYi-gUDA",
        "status": {
          "description": "The service is up",
          "id": "up",
          "image":
            "http://status.mongohq.com/images/status/tick-circle.png",
          "level": "NORMAL",
          "name": "Up",
          "url":
            "http://status.mongohq.com/api/v1/statuses/up"
        },
        "timestamp": "Wed, 13 Mar 2013 18:10:21 GMT",
        "url":
          "http://status.mongohq.com/api/v1/services/aaron/
            events/ag5tb25nb2hxLXN0YXR1c3INCxIFRXZlbnQYi-gUDA"
      },
      "description": "MongoDB Server",
      "id": "aaron",
      "name": "Aaron",
      "url":
        "http://status.mongohq.com/api/v1/services/aaron"
```

```

    },
    {
      "current-event": {
        "informational": false,
        "message": "Alex is operational.",
        "sid": "ag5tb25nb2hxLXN0YXR1c3INCxIFRXZlbnQY0coWDA",
        "status": {
          "description": "The service is up",
          "id": "up",
          "image":
            "http://status.mongohq.com/images/status/tick-
            circle.png",
          "level": "NORMAL",
          "name": "Up",
          "url":
            "http://status.mongohq.com/api/v1/statuses/up"
        },
        "timestamp": "Sat, 20 Apr 2013 15:23:20 GMT",
        "url":
          "http://status.mongohq.com/api/v1/services/alex\
          /events/ag5tb25nb2hxLXN0YXR1c3INCxIFRXZlbnQY0coWDA"
      },
      "description": "A sandbox environment for MongoHQ.",
      "id": "alex",
      "name": "Alex",
      "url":
        "http://status.mongohq.com/api/v1/services/alex"
    }
  ]
}

```

When RestKit loads this JSON, first of all it parses it to the `NSDictionary` or `NSArray` response object. Then, looking at the mapping we provided, it will make a KVC query on the response object. If the value is found, it will check its type as well as the type of our target property. If the types don't match, RestKit will try to transform it. If one of such transformations is a timestamp mapping, which in JSON is a string, and on the target property it is `NSDate`, RestKit will parse the JSON string in to `NSDate` with default (or custom provided) `NSDate` formatter(s). If the parsing is successful, it will update our destination property.

Let's check our mapping again line by line:

We create the mapping for the `StatusItem` class:

```

RKObjectMapping *mapping = [RKObjectMapping
mappingForClass:[StatusItem class]];

```

We tell how to map JSON objects to properties by providing an `NSDictionary` to the `addAttributeMappingsFromDictionary` method:

```
[mapping addAttributeMappingsFromDictionary:@{
```

Map name from JSON to the property name:

```
@"name": @"name",
```

We can't use some names for properties, such as `id` or `description`. So, we named `description` from JSON as `itemDescription` property:

```
@"description": @"itemDescription",
```

Here, RestKit will use KVC to get the value (asking name from `status` from `current-event`):

```
@"current-event.status.name" : @"statusName",  
@"current-event.status.image": @"imageUrl",
```

It will parse the `timestamp` string in JSON and store it as an `NSDate` property:

```
@"current-event.timestamp": @"timestamp",
```

And here again it will use KVC to get `message` and store it in the `eventMessage` property:

```
@"current-event.message": @"eventMessage",  
}];
```

Now that wasn't hard, was it?

Summary

In this chapter, we discovered what is RestKit, its components, and why it's good to keep things simple. We discovered a CocoaPods library manager and installed RestKit through it. We covered basic data modeling techniques, and tried a simple example by sending a request to a status page and getting parsed and mapped objects in response.

The next chapter will cover more about how to configure the RestKit, do the RESTful object manipulation, and integrate the code with our user interface, in detail. So let's move on!

2

Modeling and Loading Remote Objects

We can use the `RKObjectRequestOperation` class for every single request, as seen in the basic example of the previous chapter. But such code blocks will tend to be pretty big, and against the Don't Repeat Yourself (DRY) principle.

The most preferable way would be to configure all the API endpoints and their mapping configuration in one place, and then just use one-line request methods.

This chapter will describe how to configure our project to leverage such an approach using `RKObjectManager` and configured object mapping. Apart from discussing how to load the remote objects, we will get some knowledge on pushing data and objects back to the server. Additionally, the chapter will describe how to make our lives easier by using routing and integrate the code with Apps UI.

Object manager

The `RKObjectManager` class provides a centralized interface to configure the object mapping for request and response operations and perform such operations. It also provides helpers for the creation of `NSURLRequest` and `RKObjectRequestOperation` objects, and one-line methods to enqueue object request operations for basic HTTP request methods (GET, POST, PUT, DELETE, and so on).

Each object manager is configured with a base URL. It defines the relative URL for all requests that will be sent through the manager. One way to set the base URL is via the `managerWithBaseURL` method. This will initialize a new instance of the `RKObjectManager` class. The other ways are by configuring an `AFHTTPClient` class or creating a custom subclass—from the `AFNetworking` library. In this case, the base URL will be inherited by the object manager from an HTTP client class. The base URL can point to a root URL, or it can contain a path.

While performing a request operation, the object manager will use the base URL and the provided request path, to construct the NSURL object with `[NSURL URLWithString:relativeToURL:]`. The way this method evaluates the relativity of the URL can sometimes be confusing and surprising, and one can experience a lot of errors regarding this. For example, a small part of the `AFNetworking` documentation is provided, so one can better understand how the base URL and different paths interact.

Let's take a base URL:

```
NSURL *baseURL = [NSURL URLWithString:@"http://example.com/v1/"];
```

Now:

```
[NSURL URLWithString:@"foo" relativeToURL:baseURL];  
// Will give us http://example.com/v1/foo
```

```
[NSURL URLWithString:@"foo?bar=baz" relativeToURL:baseURL];  
// -> http://example.com/v1/foo?bar=baz
```

```
[NSURL URLWithString:@"/foo" relativeToURL:baseURL];  
// -> http://example.com/foo
```

```
[NSURL URLWithString:@"foo/" relativeToURL:baseURL];  
// -> http://example.com/v1/foo
```

```
[NSURL URLWithString:@"/foo/" relativeToURL:baseURL];  
// -> http://example.com/foo/
```

```
[NSURL URLWithString:@"http://example2.com/" relativeToURL:baseURL];  
// -> http://example2.com/
```

Having the knowledge of what an object manager is, let's try to apply it in a real-life example.

Before proceeding, it is highly recommend that we check the actual documentation on REST API of MongoHQ. The current one is at the following link:

```
http://support.mongohq.com/mongohq-api/introduction.html
```

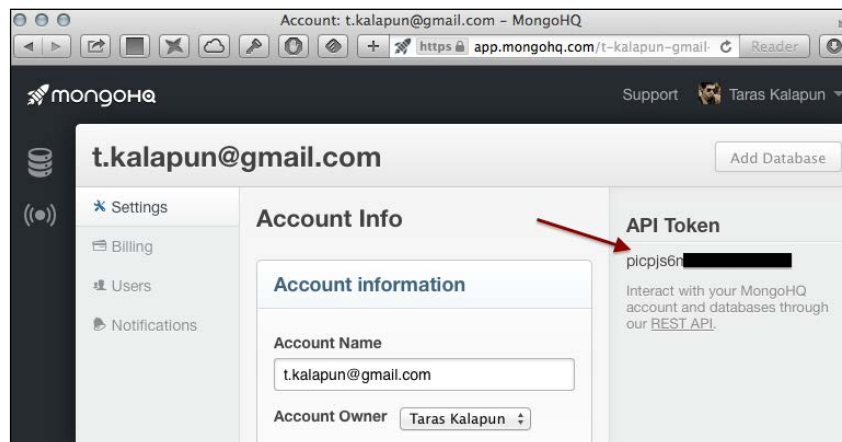
As there are no strict rules on REST API, every API is different and does a number of things in its own way. MongoHQ API is not an exception. In addition, it is currently in "beta" stage.

Some of the non-standard things one can find in it are as follows:

- The API key should be provided as a parameter with every request. There is an undocumented way of how to provide it in Headers, which is a more common approach. The way to deal with providing it with every single request is described in *Chapter 4, Advanced Stuff* in the *Custom HTTP Client* section.
- Sometimes, if you get an error with the status code returned as 200 (OK), which is not according to REST standards, the normal way would be to return something in 4xx, which is stated as a **client error**.
- Sometimes, while the output of an error message is a JSON string, the HTTP response Content-type header is set as text/plain.

To use the API, one will need a valid API Key. You can easily get one for free following a simple guideline recommended by the MongoHQ team:

1. Sign up for an account at <http://MongoHQ.com>.
2. Once logged in, click on the **My Account** drop-down menu at the top-right corner and select **Account Settings**.
3. Look for the section labeled **API Token**. From there, take your token.
4. We will put the API key into the `MongoHQ-API-Token` HTTP header. The following screenshot shows where one can find the API token key:



API Token on Account Info page

So let's set up our configuration using the following steps:

 You can use the AppDelegate class for putting the code, while I recommend using a separate `MongoHqApi` class for such App/API logic separation.

First, let's set up our object manager with the following code:

```
- (void)setupObjectManager
{
    NSString *baseUrl = @"https://api.mongohq.com";

    AFHTTPClient *httpClient = [[AFHTTPClient alloc]
        initWithBaseURL:[NSURL URLWithString:baseUrl]];

    NSString *apiKey = @"MY_API_KEY";
    [httpClient setDefaultHeader:@"MongoHQ-API-Token"
        value:apiKey];

    RKObjectManager *manager = [[RKObjectManager alloc]
        initWithHTTPClient:httpClient];

    [RKMIMETypeserialization
        registerClass:[RKNSJSONSerialization class]
        forMIMETypes:@"text/plain"];

    [manager.HTTPClient
        registerHTTPOperationClass:[AFJSONRequestOperation
            class]];

    [manager setAcceptHeaderWithMIMETypes:RKMIMETypesJSON];
    manager.requestSerializationMIMETypes = RKMIMETypesJSON;

    [RKObjectManager setSharedManager:manager];
}
```

1. Let's look at the code line by line and set the base URL. Remember not to put a slash (/) at the end, otherwise, you might have a problem with response mapping:


```
NSString *baseUrl = @"https://api.mongohq.com";
```

2. Initialize the HTTP client with baseUrl:

```
AFHTTPClient *httpClient = [[AFHTTPClient alloc]
    initWithBaseURL:[NSURL URLWithString:baseUrl]];
```

3. Set a few properties for our HTTP client, such as the API key in the header:

```
NSString *apiKey = @"MY_API_KEY";
[httpClient setDefaultHeader:@"MongoHQ-API-Token"
value:apiKey];
```

 For the real-world app, one can show an **Enter Api Key** view controller to the user, and use a `NSUserDefaults` or a keychain to store and retrieve it.

4. And initialize the `RKObjectManager` with our HTTP client:

```
RKObjectManager *manager = [[RKObjectManager alloc]
initWithHTTPClient:httpClient];
```

5. MongoHQ APIs sometimes return errors in `text/plain`, thus we explicitly will add `text/plain` as a JSON content type to properly parse errors:

```
[RKMIMETypeserialization
registerClass:[RKNSJSONSerialization class]
forMIMEType:@"text/plain"];
```

6. Register `JSONRequestOperation` to parse JSON in requests:

```
[manager.HTTPClient
registerHTTPOperationClass:[AFJSONRequestOperation class]];
```

7. State that we are accepting JSON content type:

```
[manager setAcceptHeaderWithMIMEType:RKMIMETypesJSON];
```

8. Configure so that we want the outgoing objects to be serialized into JSON:

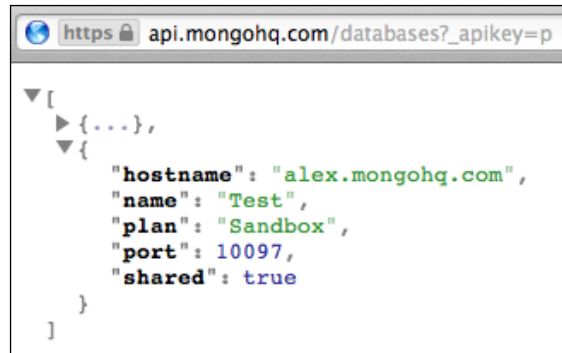
```
manager.requestSerializationMIMEType = RKMIMETypesJSON;
```

9. Finally, set the shared instance of the object manager, so that we can easily re-use it later:

```
[RKObjectManager setSharedManager:manager];
```

Sending requests with object manager

Next, we want to query our databases. Let's first see how a database request will show us the output in JSON. To check this, go to http://api.mongohq.com/databases?_apikey=YOUR_API_KEY in your web browser YOUR_API_KEY. If a JSON-formatter extension (<https://github.com/rfletcher/safari-json-formatter>) is installed in your Safari browser, you will probably see the output shown in the following screenshot.



JSON response from API

As we see, the JSON representation of one database is:

```
[
  {
    "hostname": "sandbox.mongohq.com",
    "name": "Test",
    "plan": "Sandbox",
    "port": 10097,
    "shared": true
  }
]
```

Therefore, our possible MDatabase class could look like:

```
@interface MDatabase : NSObject

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *plan;
@property (nonatomic, strong) NSString *hostname;
@property (nonatomic, strong) NSNumber *port;

@end
```

We can also modify the `@implementation` section to override the description method, which will help us while debugging the application and printing the object:

```
// in @implementation MDatabase
- (NSString *)description
{
    return [NSString stringWithFormat:@"%@" on %@ @ %@:%@",
           self.name, self.plan, self.hostname, self.port];
}
```

Now let's set up a mapping for it:

```
- (void)setupDatabaseMappings
{
    RKObjectManager *manager = [RKObjectManager sharedManager];
    Class itemClass = [MDatabase class];
    NSString *itemsPath = @"/databases";

    RKObjectMapping *mapping = [RKObjectMapping
                               mappingForClass:itemClass];

    [mapping addAttributeMappingsFromArray:@[@"name", @"plan",
                                           @"hostname", @"port"]];

    NSString *keyPath = nil;
    NSIndexSet *statusCodes =
        RKStatusCodeIndexSetForClass(RKStatusCodeClassSuccessful);

    RKResponseDescriptor *responseDescriptor =
        [RKResponseDescriptor
         responseDescriptorWithMapping:mapping
                             method:RKRequestMethodGET
                             pathPattern:itemsPath
                             keyPath:keyPath
                             statusCodes:statusCodes];

    [manager addResponseDescriptor:responseDescriptor];
}
```

Let's look at the mapping setup line by line:

1. First, we define a class, which we will use to map to:

```
Class itemClass = [MDatabase class];
```

2. And the endpoint we plan to request for getting a list of objects:

```
NSString *itemsPath = @"/databases";
```

3. Then we create the `RKObjectMapping` mapping for our object class:

```
RKObjectMapping *mapping = [RKObjectMapping  
mappingForClass:itemClass];
```

4. If the names of JSON fields and class properties are the same, we will use an `addAttributeMappingsFromArray` method and provide the array of properties:

```
[mapping addAttributeMappingsFromArray:@[@"name", @"plan",  
@"hostname", @"port"]];
```

5. The root JSON key path in our case is `nil`. It means that there won't be one.

```
NSString *keyPath = nil;
```

6. The mapping will be triggered if a response status code is anything in `2xx`:


```
NSIndexSet *statusCodes =  
RKStatusCodeIndexSetForClass (RKStatusCodeClassSuccessful);
```

7. Putting it all together in response descriptor (for a GET request method):

```
RKResponseDescriptor *responseDescriptor =  
[RKResponseDescriptor  
responseDescriptorWithMapping:mapping  
method:RKRequestMethodGET  
pathPattern:itemsPath  
keyPath:keyPath  
statusCodes:statusCodes];
```

8. Add response descriptor to our shared manager:

```
RKObjectManager *manager = [RKObjectManager sharedManager];  
[manager addResponseDescriptor:responseDescriptor];
```

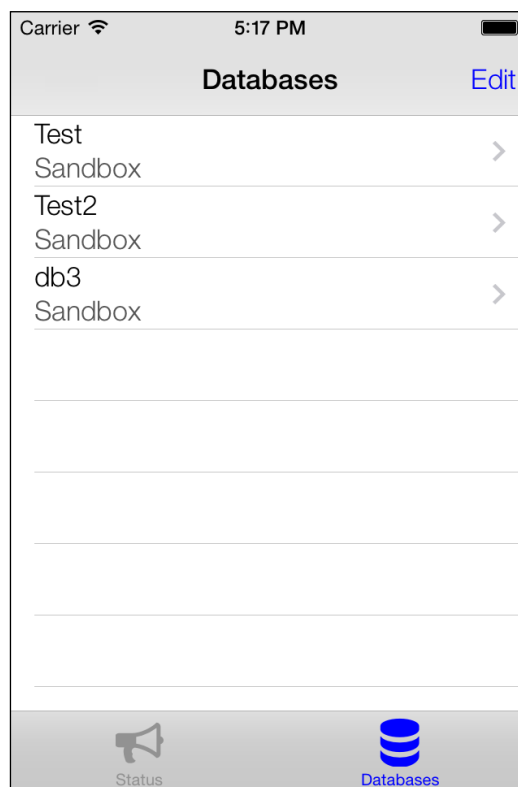
 Sometimes, depending on the architectural decision, it's nicer to put the mapping definition as part of a model object, and later call it like `[MDatabase mapping]`, but for the sake of simplicity, we will put the mapping in line with RestKit configuration.

The actual code that loads the database list will look like:

```
RKObjectManager *manager = [RKObjectManager sharedManager];  
[manager getObjectsAtPath:@"/databases"  
parameters:nil  
success:^(RKObjectRequestOperation *operation,  
RKMappingResult *mappingResult)  
{  
NSLog(@"Loaded databases: %@", [mappingResult array]);  
}  
}
```

```
failure:^(RKObjectRequestOperation *operation,  
         NSError *error)  
{  
    NSLog(@"Error: %@", [error localizedDescription])  
}];
```

As you may have noticed, the method is quite simple to use and it uses block-based APIs for callbacks, which greatly improves the code readability, compared to using delegates, especially if there is more than one network request in a class. A possible implementation of a table view that loads and shows the list of databases will look like the following screenshot:



View of loaded Database items

Integrating with UI and DRYing the table view

In this section, we will cover how RestKit might be integrated with our application User Interface.

The application we are building is a typical business app. It heavily uses tables (such as `UITableViewController`) to present the loaded data (list of objects) to the user. In our case, the types of data that are available from the web service for presentation to the user in a table view are:

- Databases
- Collections
- Documents
- Plans
- Indexes
- Invoices
- Deployments

Each of these table view controllers does basically the same thing, it loads a list of objects through a GET request on a specific path, and shows it to the user. It also provides the ability to edit the list with actions, such as delete an object, edit an object, and add a new object. So, for keeping our code clean and not repeating it, the advice is to create a generic table view controller, with properties and methods that our subclasses will define/override. Such an approach of creating generic controllers greatly speeds up a development cycle. You will always have a chance to modify a specific controller for a custom interface or logic implementation.

So let's create `GenericTableViewController` with the following code:

```
@interface GenericTableViewController : UITableViewController

@property (nonatomic, strong) NSArray *items;
@property (nonatomic, strong) NSString *path;
@property (nonatomic, strong) NSDictionary *parameters;

// Methods to re-use / override
- (void)refresh;
- (void)willStartLoading;
- (void)finishedLoadingWithItems:(NSArray *)newItems;
- (void)finishedLoadingWithError:(NSError *)error;

- (void)configureCell:(UITableViewCell *)cell atIndexPath:(NSIndexPath *)indexPath;

@end
```

A brief description of properties we are using:

Property name	Type	Description
items	array	Property to hold our loaded objects
path	string	Path to use for loading the remote objects
parameters	dictionary	Possible parameters for a request


The full code listing on the implementation part will not be provided here, only the interesting ones:

The initial setup is done in the `viewDidLoad` method:

```
// Pull to refresh control
self.refreshControl = [[UIRefreshControl alloc] init];
[self.refreshControl addTarget:self
                             action:@selector(refresh)
                             forControlEvents:UIControlEventValueChanged];

// Clear selection between presentations
self.clearsSelectionOnViewWillAppear = YES;

// Display an Edit button in the navigation bar
self.navigationItem.rightBarButtonItem = self.editButtonItem;
```


 One can also use the **ISRefreshControl** library to simulate iOS 6 "Pull-to-Refresh" control in the apps that target iOS 5. Just put `pod 'ISRefreshControl'` in Podfile and fire pod install.

If there are no items loaded, the refresh should be fired in the `viewWillAppear` method:

```
// Reload items if they are not loaded yet
if (!self.items) [self refresh];
```

The `refresh` method will be triggered for refreshing the objects we are showing:

```
- (void)refresh
{
    // If refreshed from editing mode, switch to normal
    [self setEditing:NO animated:NO];

    [self willStartLoading];
}
```

```
    RKObjectManager *manager = [RKObjectManager sharedManager];
    [manager getObjectAtPath:self.path
         parameters:self.parameters
         success:^(RKObjectRequestOperation *operation,
                 RKMMappingResult *mappingResult)
         {
            [self finishedLoadingWithItems:[mappingResult array]];
        }
         failure:^(RKObjectRequestOperation *operation,
                 NSError *error)
         {
            [self finishedLoadingWithError:error];
        }
    ];
}
```

The `willStartLoading` method will be triggered just before the start of the actual loading of objects. Mostly, the code it contains is related to showing the user a **Loading** indicator:

```
// Toggle "Pull to refresh" to show Activity indicator
[self.refreshControl beginRefreshing];

// Show a "Loading" HUD to the user
[SVProgressHUD showWithStatus:@"Loading"];
```

In addition, here are two methods to trigger when the loading is finished:

```
- (void)finishedLoadingWithItems:(NSArray *)newItems
{
    // Update the current items with new ones
    self.items = newItems;
    NSLog(@"Loaded items: %@", self.items);

    [self.tableView reloadData];

    // Hide loading indicators
    [self.refreshControl endRefreshing];
    [SVProgressHUD dismiss];
}

- (void)finishedLoadingWithError:(NSError *)error
{
    [SVProgressHUD dismiss];
    NSLog(@"Error on loading: %@", error);

    // Show some Alert with error description
}
```

```

    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Error"
        message:[error localizedDescription]
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
    [alert show];
}

```

The `finishedLoadingWithItems` method updates the `items` array property with newly loaded items, reloads the table view, and hides the loading indicators.

The `finishedLoadingWithError` method is fired when an error occurs at the time of loading or processing the loaded data. It will hide the loading indicators and show the error to the user. As a message, `[error localizedDescription]` can be shown, which will contain the error message produced by RestKit. Alternatively, a custom one can be generated depending on an error type and other circumstances.

For a `UITableView` delegate method, `tableView:cellForRowAtIndexPath`, we will create a re-usable table cell, and trigger the `configureCell:atIndexPath:` method, which will actually configure the cell with a title, subtitle, and, if applicable, an image. Usage of separate `configure` method allows easier overriding of the implementation in subclasses, in case of Core Data manipulations, and in `NSFetchedResultsController` delegates:

```

- (void)configureCell:(UITableViewCell *)cell
atIndexPath:(NSIndexPath *)indexPath
{
    // Get our loaded item
    NSObject <MTableObject> *item = self.items[indexPath.row];

    cell.textLabel.text = item.titleText;
    cell.detailTextLabel.text = item.subtitleText;
}

```

Every item object conforms to a `<MTableObject>` protocol, which has a `titleText` and `subtitleText` getters. This gives us the ability to easily specify how every object should present itself in a model class.

To make this work for a database object, create a protocol definition called `MTableObject` with the following code:

```

// MTableObject.h
@protocol MTableObject <NSObject>

- (NSString *)titleText;
- (NSString *)subtitleText;

@end


```

And add `titleText` and `subtitleText` methods to the `MDatabase` implementation:

```
// in MDatabase @implementation
- (NSString *)titleText
{
    return self.name;
}

- (NSString *)subtitleText
{
    return self.plan;
}
```

Now it is pretty easy to change the output information that is displayed to the user.

 For loading images in a `UITableViewCell`, one can use a provided `UIImageView+AFNetworking` category from the `AFNetworking` library:

```
[cell.imageView setImageWithURL:imageURL
placeholderImage:placeholderImage]
```

Now for our database list loading, create `DatabasesViewController` (you didn't forget to subclass it from `GenericTableViewController`, did you?) and just override the initialization point and set the path to be used:

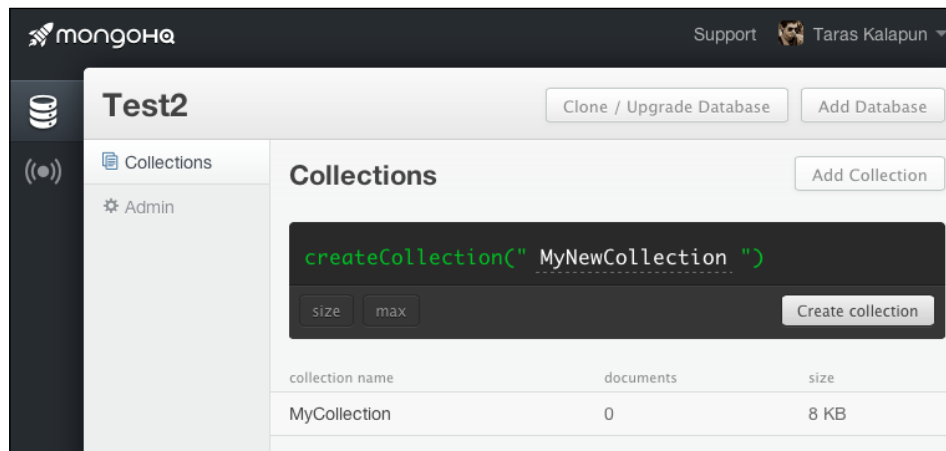
```
- (id)init
{
    self = [super init];
    if (self) {
        // Configure the controller
        self.path = @"/databases";
        self.title = @"Databases";
    }
    return self;
}
```

That's all you actually need to get started!

Requesting related objects

Every MongoDB database has a collection, which consists of documents. While we didn't get to the creating objects topic in our guide, to create a collection manually go to MongoHQ's web console, select a database, and click on the **Add Collection** button.

You can then name your new collection and click on the **Create collection** button as shown in the following screenshot:



Creating a new Collection in a web interface

Checking the **collection** API documentation, we will see a request path `/databases/:db/collections`. The `:db` in this path is a parameter, and should be equal to a database name. I prefer calling it `databaseID`. Such a name reflects the true essence of a parameter as the database name is a database identifier here.

So to load our collections, we will use a path pattern, `/databases/:databaseID/collections`. The mapping setup for collections looks quite similar to the database mapping:

```
- (void) setupCollectionMappings
{
    RKObjectManager *manager = [RKObjectManager sharedManager];

    Class itemClass = [MCollection class];
    NSString *itemsPath = @"/databases/:databaseID/collections";

    RKObjectMapping *mapping = [RKObjectMapping
mappingForClass:itemClass];
    [mapping addAttributeMappingsFromArray:@[@"name", @"count",
@"indexCount", @"storageSize"]];

    NSIndexSet *statusCodes = RKStatusCodeIndexSetForClass(RKStatusCod
eClassSuccessful);

    RKResponseDescriptor *responseDescriptor = [RKResponseDescriptor
responseDescriptorWithMapping:mapping
```

```
        pathPattern:itemsPath
        method:RKRequestMethodGET
        keyPath:nil
        statusCodes:statusCodes];

    [manager addResponseDescriptor:responseDescriptor];
}
```

As `:databaseID` is a parameter in our request URL, you need to substitute it with the real value. For such purposes, the `RKPathFromPatternWithObject` function exists:

```
// - (void)init method in CollectionsViewController
self.path =
RKPathFromPatternWithObject(@"databases/:databaseID/collections",
self.database);
```

The `self.database` is pre-set upon opening the `CollectionsViewController` from the `DatabasesViewController` class:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    MDatabase *db = self.items[indexPath.row];

    CollectionsViewController *vc = [[CollectionsViewController
    alloc] init];
    vc.title = db.name;
    vc.database = db;

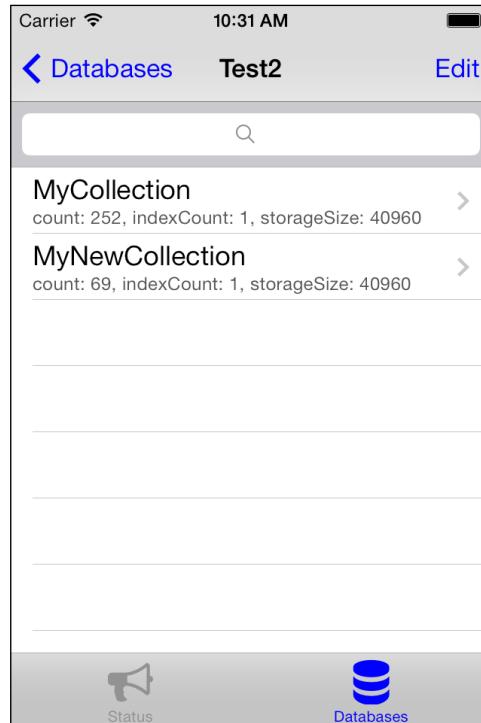
    [self.navigationController pushViewController:vc
    animated:YES];
}
```

When `RKPathFromPatternWithObject` starts matching the path pattern, it will use KVC to get the value for the `databaseID` key from the `MDatabase` object. In our case, we will define a "property getter" in the `MDatabase` class, which will only return the database name:

```
// @interface MDatabase
@property (nonatomic, readonly) NSString *databaseID;

// @implementation
- (NSString *)databaseID
{
    return self.name;
}
```

Running the App with such code and selecting the **Test2** database will load collections in that database. The sample UI might look like the following screenshot:




Loaded Collections

RESTful object manipulation

The power of RestKit comes when you want to manipulate remote objects. For the RestKit, all you need is to configure a response object mapping, set the request paths, and fire the appropriate method with the object.

In a RESTful web service, the CRUD operations are tightly related to the HTTP methods. Create, read, update, and delete operations map to the HTTP methods GET, PUT, POST, and DELETE respectively.


 By the way, the **HTTP 1.1** standard defines eight methods (or verbs): HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, and CONNECT.

We discussed the Read operations in the previous sections. So in this one, we'll start by talking about the Create operation.

For example, let's make a code to create the new collection. As we can see in the API documentation, to create a new collection you need to fire a POST HTTP method to `/databases/:db/collections`. As you can see, it's the same path pattern used in GET operations to get the list of collections. The other thing to notice is that the endpoint is expecting a JSON-encoded object. That's why in the beginning, when we were setting up our object manager, we entered the following code snippet:

```
manager.requestSerializationMIMETYPE = RKMIMETYPEJSON
```

This means any object that we will attach to the request (sending to the web service, in other words) will be serialized into a JSON string.

For the object manager to know which properties it should serialize while sending an object to a remote endpoint, we need to set up a request mapping and a request descriptor:

```
// in setupCollectionMappings

// The endpoint for manipulating with existing object
NSString *itemPath =
    @"/databases/:databaseID/collections/:collectionID";

// Create the request mapping
RKObjectMapping *requestMapping = [RKObjectMapping
    requestMapping];

// "name" will be same in JSON and in a class
[requestMapping addAttributeMappingsFromArray:@[@"name"]];

// For any object of class MCollection, serialize into an
// NSMutableDictionary using the given mapping
// If we will provide the rootKeyPath, serialization will nest
// under the 'provided' key path

RKRequestDescriptor *requestDescriptor = [RKRequestDescriptor
    requestDescriptorWithMapping:requestMapping
                        objectClass:itemClass
                        rootKeyPath:nil
                        method:RKRequestMethodAny];

[manager addRequestDescriptor:requestDescriptor];
```

You can also provide how a local property will be serialized. For example, for creating a database object on a remote web service, a `slug` attribute in the JSON request needs to be provided, which is actually a `plan` property in a local object. Thus, we will code to map the `plan` property from its class to `slug` in serialized JSON, and that will look like:

```
NSMutableDictionary *plan2slugMap = @{@"plan": @"slug"};
[requestMapping addAttributeMappingsFromDictionary:plan2slugMap];
```

So the simple example code to create a new collection would be:

```
NSString *path =
    RKPathFromPatternWithObject(@"databases/:databaseID/
    collections", self.database);

MCollection *item = [[MCollection alloc] init];
item.name = @"some_collection";

RKObjectManager *manager = [RKObjectManager sharedManager];
[manager postObject:item
    path:path
    parameters:nil
    success:^(RKObjectRequestOperation *operation,
              RKMappingResult *mappingResult)
    {
        NSLog(@"Created!");
    }
    failure:^(RKObjectRequestOperation *operation, NSError
              *error)
    {
        NSLog(@"Error: %@", error)
    }
    ]];
```

Now, for deleting or editing the objects, constructing the URL from its path pattern would be a bit more difficult. If you will look into the needed path pattern—`/databases/:databaseID/collections/:collectionID`—you will notice that you need to provide a `collectionID` parameter, apart from `databaseID`. So, putting the `MDatabase` there is not appropriate any more. You can think about constructing a path as shown in the following code:

```
NSMutableDictionary *params = @{
    @"databaseID" : self.database.databaseID,
    @"collectionID" : self.collection.collectionID
};

path = RKPathFromPatternWithObject(@"databases/:databaseID/
collections/:
collectionID", params);
```

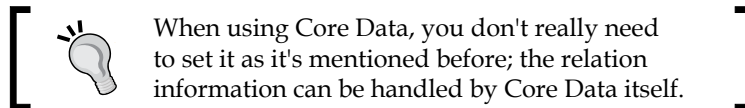
Sometimes, it is worth doing it like this. While the nicer way would be to store the relation information in a `MCollection` object. To achieve this, edit the `MCollection` class and add:

```
@property (nonatomic, strong) NSString *databaseID;
```

And set it later in a code:

```
self.collection.databaseID = self.database.databaseID;
```

You can omit this presetting if you are going to use routing and metadata mapping, which we will discuss in detail in *Chapter 4, Advanced Stuff* in the *Metadata mapping* section.



After setting the property, construct the path:

```
path =
    RKPathFromPatternWithObject(@"databases/:databaseID/collections/:
    collectionID", self.collection);
```

Having the path constructed, and the `self.collection` object set, you can fire the Delete operation in the same way as a Create operation:

```
[manager deleteObject:self.collection
           path:path
           parameters:nil
           success:^(RKObjectRequestOperation *operation,
                    RKMappingResult *mappingResult)
           {
           }
           failure:^(RKObjectRequestOperation *operation,
                    NSError *error)
           {
           }];
```

The Edit operation is also quite similar. For example, we can change the collection name as follows:

```
self.collection.name = @"new_collection_name";
[manager putObject:self.collection
           path:path
           parameters:nil];
```

```

        success:^(RKObjectRequestOperation *operation,
                 RKMappingResult *mappingResult)
        {
        }

        failure:^(RKObjectRequestOperation *operation, NSError
                 *error)
        {
        }
    }];

```


But what if a `collectionID` (which in our case is a collection name) consists of special characters or spaces? We need a way to make percent escape on it before putting in a path. Of course you can wrap the output in a specially-provided function:

```

    RKPercentEscapedQueryStringFromStringWithEncoding(self.collection.
    collectionID, NSUTF8StringEncoding);

```

However, there's a much easier way in controlling these and other things, and that is called Routing.

 Percent escape, or URL encode, will change all special non-ASCII characters to specific codes prefixed with a percent "%" character. For more information on this check the Percent-encoding article in Wikipedia at <http://en.wikipedia.org/wiki/Percent-encoding>

Routing inside out

One of the most useful powers of object manager is that it has a so-called **Router**. It simplifies the generation of request URLs and helps developers maintain the path patterns. Once the Router is configured, it gives the ability to stop worrying about what endpoint to be used to make the next request, as all the object manager request methods will use the power of the Router. Using a centralized knowledge, a Router will know how to generate an appropriate URL for specific objects, request methods, or names. This greatly simplifies the amount of code you write and improves readability.

URLs will be generated by the Router using three types of routes:

- **Named routes:** They are like a named symbolic link. This type of route is not related to any particular class. It represents a single path and an optional HTTP request method. As an example, one can define such a route with the name `database_list`, which will be a GET request to a list of databases with the path `/databases`.

- **Class routes:** This is a very common route that is used for manipulating CRUD with a particular object. The router will identify the needed route by its object class and an appropriate HTTP request method. Once the router finds the appropriate route, it will interpolate the route's path pattern against the provided object and instantiate the NSURL object. For example, route for the `MCollection` class, `/databases/:databaseID/collections` path, and the POST request method can be used for creating a new collection object.
- **Relationship routes:** They are defined by their relationship to other objects. Such a route represents a path through which the relationship of a parent object can be manipulated. The example could be a relationship route for `MDatabase` collections with the name `database_collections` that will point to a path, `/databases/:databaseID/collections` and load the `MCollection` objects. Basically, this type of route is quite similar to and works like a normal named route.

The routes themselves are modeled using class methods of `RKRoute`, and should be added to the router's `routeSet`.

Keeping this information in mind, let us set some routes up for our collection object. Edit the `setupCollectionMappings` method and add the following routes:

```
// Route for loading a list of objects
RKRoute *itemsRoute = [RKRoute routeWithName:@"collections"
    pathPattern:itemsPath method:RKRequestMethodGET];
itemsRoute.shouldEscapePath = YES;

// Route for creating a new object
RKRoute *NewItemRoute = [RKRoute routeWithClass:itemClass
    pathPattern:itemsPath method:RKRequestMethodPOST];
NewItemRoute.shouldEscapePath = YES;

// Route for manipulating with existing object
RKRoute *itemRoute = [RKRoute routeWithClass:itemClass
    pathPattern:itemPath method:RKRequestMethodAny];
itemsRoute.shouldEscapePath = YES;
```

Then add the defined routes to the object manager Router with this piece of code:

```
[manager.router.routeSet addRoutes:@[itemsRoute, newItemRoute,
    itemRoute]];
```

The `shouldEscapePath` property specifies if a provided parameter value for a path pattern should automatically be percent-escaped for proper URL handling. In the previous example, for defining the route for loading a list of objects we are using a named route, but you can use the relationship route for this case also. As mentioned in the beginning, there are tons of ways of how to write code, and you can't be sure that this architecture decision is the best one.

In our case, we have two routes for the same path. While making a POST request, the Router will first check if there's a route for this specific request method (POST), and only if none are found will it return the one that matches any request method.

Having all routes predefined greatly simplifies coding, as you don't need to use path patterns for every CRUD request you make. The Router will know which one to use in a particular case. So there's less chance of making a mistake.

Now, having defined routes for our collection object, how will one use them?

Object manager supports route requests in all its methods. If you supply the object and omit the path, the object manager will try searching for the appropriate route.

For a get request, we will need a route name and an object that has the relation information, if the route needs it. So, for getting the list of our collections, here's a snippet of code:

```
[manager getObjectAtPathForRouteNamed:@"collections"
        object:self.database
        parameters:nil
        success:^(RKObjectRequestOperation
*operation, RKMappingResult *mappingResult)
{
    [self finishedLoadingWithItems:[mappingResult array]];
}
        failure:^(RKObjectRequestOperation
*operation, NSError *error)
{
    [self finishedLoadingWithError:error];
}]];
```

As you see here, `collections` is a route name, and we provide the current database object to use its `databaseID` in constructing the request.

For using routes in other CRUD operations that involve manipulating with one particular object, the only change you need is just setting the path by omitting it. Also, you will need to be sure that the object has information about its parent relations.

As a conclusion, for our generic table view controller, we can add the following properties:

```
@property (nonatomic, strong) NSString *routeName;
@property (nonatomic, strong) id routeObject;
```

A brief description of properties we added:

Property name	Type	Description
routeName	string	Name of route to be used for loading with routing enabled. If not defined, a path will be used.
routeObject	id	An object to be used for constructing the route path if routing is used.

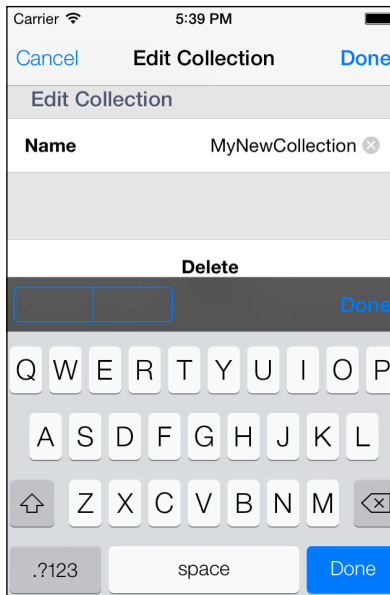
You also need to implement the "routed" loading:

```
[manager getObjectAtPathForRouteNamed:self.routeName
        object:self.routeObject
        parameters:self.parameters
        success:^(RKObjectRequestOperation
*operation, RKMappingResult *mappingResult)
{
    [self finishedLoadingWithItems:[mappingResult array]];
}
        failure:^(RKObjectRequestOperation
*operation, NSError *error)
{
    [self finishedLoadingWithError:error];
}];
```

Entering data in forms

We already have our `GenericTableViewController` class for loading and showing objects; now let's see how we can make a kind-of-the-same controller for creating a new object or editing and delegating the existing objects.

You can create one from scratch using generic `UIKit` components, but we will do this with a library called `QuickDialog`. The example form created using this library is shown in the following screenshot:



Editing a Collection

First of all, you need to install it to your project. To do so, add pod 'QuickDialog' to your Podfile, and run `pod install` from a command line in the root of a project.

Let's call out the new controller, `GenericFormViewController`:

```
@interface GenericFormViewController : QuickDialogController

@property (nonatomic, strong) NSString *path;
@property (nonatomic, strong) NSString *itemPath;
@property (nonatomic, assign) id item;
@property (nonatomic, assign) BOOL shouldCreateNewItem;

- (BOOL)validateItemPassed;

// Interface actions
- (void)saveAction;
- (void)deleteAction;

- (void)createItemRequest;
- (void)updateItemRequest;
- (void)deleteItemRequest;
@end
```


A brief description of properties we are using:

Property name	Type	Description
path	string	Path for creating the item. If not set, use routes.
itemPath	string	Path for manipulating with one item. If not set, use routes.
item	id	The alias to the getter/setter of the object we are manipulating.
shouldCreateNewItem	boolean	Property that triggers if Create or Update action to be used.

Now let's see the request methods. `createItemRequest` method for creating items:

```
RKObjectManager *manager = [RKObjectManager sharedManager];
[manager postObject:self.item
    path:self.path
    parameters:nil
    success:^(RKObjectRequestOperation *operation,
              RKMappingResult *mappingResult)
    {
        [self done];
    }
    failure:^(RKObjectRequestOperation *operation, NSError
              *error)
    {
        [self errorOnLoad:error];
    }
];
```

The `updateItemRequest` method for updating the item to a web service after editing it locally:

```
RKObjectManager *manager = [RKObjectManager sharedManager];
[manager putObject:self.item
    path:self.itemPath
    parameters:nil
    success:^(RKObjectRequestOperation *operation,
              RKMappingResult *mappingResult)
    {
        [self done];
    }
    failure:^(RKObjectRequestOperation *operation, NSError
              *error)
    {
        [self errorOnLoad:error];
    }
];
```

The `deleteItemRequest` method for deleting the item from remote:

```
RKObjectManager *manager = [RKObjectManager sharedManager];
[manager deleteObject:self.item
          path:self.itemPath
          parameters:nil
          success:^(RKObjectRequestOperation *operation,
                  RKMappingResult *mappingResult)
          {
            [self done];
          }
          failure:^(RKObjectRequestOperation *operation,
                  NSError *error)
          {
            [self errorOnLoad:error];
          }
];
```

That's all for RestKit-specific stuff. Now, if you're wondering about `QuickDialog` UI, here's what I do to create it:

The way to initialize it is via providing a `QRootElement` object in a method `initWithRoot`.

```
- (id)init
{
    QRootElement *root = [[self class] createRootElement];
    self = [super initWithRoot:root];
    return self;
}

+ (QRootElement *)createRootElement
{
    QRootElement *root = [[QRootElement alloc] init];
    root.grouped = YES;
    return root;
}
```

Additionally, a small hack is needed to Update the `tableView` object of `QuickDialog`:

```
- (void)updateQuickDialogView
{
    // self.root is a property from QuickDialogController
    self.quickDialogTableView.root = self.root;
}
```

A **Save** button is added to the navigation controller; it triggers the `saveAction` method with the following code:

```
// Fetch the Entry element data to our object
[self.root fetchValueUsingBindingsIntoObject:self.item];

// Validate that entered data is correct
if (![self validateItemPassed]) {
    return;
}

[SVProgressHUD show]; // Show loading HUD
if (self.shouldCreateNewItem) {
    [self createItemRequest];
} else {
    [self updateItemRequest];
}
```

We also need a method to create a **Delete** button (we'll use it in this recipe):

```
- (QSection *)deleteButtonSection
{
    QPushButtonElement *btn = [[QPushButtonElement alloc]
initWithTitle:@"Delete"];
    btn.onSelected = ^{
        [self deleteAction];
    };

    // Put in in a new section
    QSection *section = [[QSection alloc] initWithTitle:@""];
    [section addElement:btn];
    return section;
}
```

Now for our Collection, let's create a subclass of `GenericFormViewController`:

```
@interface CollectionFormViewController : GenericFormViewController

@property (nonatomic, strong) MDatabase *database;
@property (nonatomic, strong) MCollection *collection;

@end
```

And specify how `self.item` should be aliased:

```
- (id)item
{
    return self.collection;
}

- (void)setItem:(id)item
{
    self.collection = item;
}
```

Now if we would use the old way of constructing the paths, we could put something like:

```
self.path =
    RKPathFromPatternWithObject(@"databases/:databaseID/
    collections",self.database);
if (self.collection) {
    NSDictionary *pathParams = @{
        @"databaseID" : self.database.databaseID,
        @"collectionID":
        RKPercentEscapedQueryStringFromNSStringWithEncoding(self.
        collection.collectionID, NSUTF8StringEncoding)};

    self.itemPath =
        RKPathFromPatternWithObject(@"databases/:databaseID/
        collections/:collectionID", pathParams);
}
```

But if we choose to use the routes, it's simple:

```
// Create new collection, if it's a Create request
if (self.shouldCreateNewItem) {
    self.collection = [[MCollection alloc] init];
}

// Pre-set relation info for using it in Class Routes
self.collection.databaseID = self.database.databaseID;
```

The next thing is to create a form, where we will enter (or edit) the collection name:

```
QSection *section = [[QSection alloc] initWithTitle:self.title];

QEntryElement *nameEntry = [[QEntryElement alloc]
    initWithTitle:@"Name"
    Value:self.collection.name
```

```
Placeholder:@"collection name"];

// QuickDialog way of binding...
nameEntry.bind = @"textValue:name";
[section addElement:nameEntry];
[self.root addSection:section];

// Add Delete button if it's existing object
if (!self.shouldCreateNewItem) {
    [self.root addSection:[self deleteButtonSection]];
}

// Update the QuickDialog
[self updateQuickDialogView];
```

You can read about using QuickDialog and check the demo and its capabilities by visiting their GitHub page at <https://github.com/escoz/QuickDialog> or blog post at <http://escoz.com/open-source/quickdialog>.

To open such a form dialog, I recommend presenting it modally:

```
- (void)presentCreateOrEditFormForObject:(id)item
{
    CollectionFormViewController *vc = [[CollectionFormViewController
alloc] init];

    vc.delegate = self;
    vc.database = self.database;
    vc.collection = item;

    UINavigationController *nc = [[UINavigationController alloc]
initWithRootViewController:vc];
    [self.navigationController presentViewController:nc
animated:YES];
}
```

For creating a new one:

```
- (IBAction)createNewItem:(id)sender
{
    // Specify nil for showing the Create form
    [self presentCreateOrEditFormForObject:nil];
}
```

And for editing the existing one:

```
- (void)editItemAtIndexPath:(NSIndexPath *)indexPath
{
    MCollection *item = self.items[indexPath.row];
    [self presentCreateOrEditFormForObject:item];
}
```

Summary

In this chapter, we learned how to set up the RestKit library to work for our web service, we talked about sending requests, getting responses, and how to do object manipulations. We also talked about simplifying the requests by introducing routing. In addition, we discussed how integration with UI can be done and created forms.

The next chapter will cover integrating RestKit and our web service with Core Data databases. We will also be given some additional information related to this topic. Stay tuned!

3

Persistence with Core Data

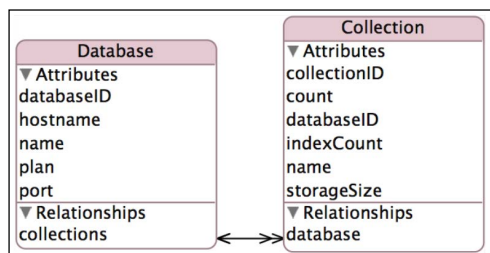
This chapter describes how to make an application use persistence data using Core Data. For our example, we will make Database and Collection elements to be backed by Core Data in our application. We will leave Documents out of Core Data, as they are more dynamic, and by this, we will see how to deal with a hybrid environment, where some objects are backed by Core Data, and some are not.

The idea is as follows – when an application is started, we will ask RestKit to refresh information about our `database` and `collection` objects. In this procedure, RestKit will determine the changes on a server in this data, and will sync our database to those changes. We assume that the information in the web service is the most correct and the latest.

Later on, we will use the cached information from our database, and we will refresh it manually or at the start of the next application, when the user will trigger the `Pull to refresh` action in table view. This will give us a faster response in our app by minimizing the time in making network requests and processing responses.

Setting up a database

Let's create entities as shown in the following image:



Core Data scheme

The attributes for each entity are shown in the following table.

Database

The following table holds information on databases that are associated without a MongoHQ account. The main properties are the names of the database and its plan.

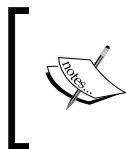
Attribute	Type	Notes
databaseID	String	It is a transient property; in code we link it to a name
hostname	String	
name	String	It is an actual ID of a database
plan	String	
port	Integer	

Collection

Unlike SQL databases, document-based databases have collections. This table represents collections that are associated with specific databases. The attributes for collection are shown in the following table:

Attribute	Type	Notes
databaseID	String	It is the ID of a database for connecting relationships
collectionID	String	It is a transient property; in code we link it to name
name	String	It is an actual ID of a collection
count	Integer	
indexCount	Integer	
storageSize	Integer	

In addition, there is a relationship between database and collection – one-to-many type.



In the provided code that you will get with this book, you will see the usage of `USE_COREDATA` in macro. The macro is defined in the `Defines.h` file and helps distinguish the different code versions showing the difference of code while using Core Data and without.

Configuring

Before using RestKit with Core Data, be sure to check that the Core Data library is linked to your project's target and you have added `#import <CoreData/CoreData.h>` before importing RestKit; for example, in the application's `Prefix.pch` file:

```
#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
    #import <SystemConfiguration/SystemConfiguration.h>
    #import <CoreData/CoreData.h>
    #import <RestKit/RestKit.h>
#endif
```

The configuration of RestKit to use Core Data is quite similar if you would use Core Data without it. Create the (void) `setupCoreData` method:

```
// Configure RestKit's Core Data stack
NSURL *modelURL = [NSURL URLWithString:[NSBundle mainBundle]
pathForResource:@"MongoHq" ofType:@"momd"]];

// Due to an iOS 5 bug, the managed object model returned is
immutable.
NSManagedObjectModel *managedObjectModel = [[NSManagedObjectModel
alloc] initWithContentsOfURL:modelURL] mutableCopy];
RKManagedObjectStore *managedObjectStore = [[RKManagedObjectStore
alloc] initWithManagedObjectModel:managedObjectModel];

NSString *storePath = [RKApplicationDataDirectory() stringByAppendingP
athComponent:@"MongoHq.sqlite"];
NSError *error = nil;

[managedObjectStore addSQLitePersistentStoreAtPath:storePath
                    fromSeedDatabaseAtPath:nil
                    withConfiguration:nil
                    options:nil
                    error:&error];

// Create default contexts
// For main thread and background processing
[managedObjectStore createManagedObjectContexts];

// Set the default store shared instance
[RKManagedObjectStore setDefaultStore:managedObjectStore];

// Assign Managed object store to Object manager
RKObjectManager *manager = [RKObjectManager sharedManager];
manager.managedObjectStore = managedObjectStore;
```

And put it just after setting the object manager, and before setting up the mapping:

```
[self setupObjectManager];

// Setup CoreData stack after Object Manager
[self setupCoreData];
[self setupMappings];
```

That's it. The only thing you will need to change is a mapping code for Core Data support. Before proceeding with that, let me give a hint on two side libraries. It's not mandatory to use them, but they can greatly improve your coding time.

Magical Record

Magical Record is a popular library for simplifying Core Data-related code. It was inspired by Ruby on Rails' ActiveRecord.

"Active record is an approach to accessing data in a database. A database table or view is wrapped into a class; thus an object instance is tied to a single row in the table. After creating of an object, a new row is added to the table after saving the table. Any object that is loaded gets its information from the database; when an object is updated, the corresponding row in the table is also updated. The wrapper class implements accessor methods or properties for each column in the table or view."

This is the definition given in http://en.wikipedia.org/wiki/Active_record_pattern.


The goal of a library is:

- To clean up Core Data-related code
- Allow for clear, simple, and one-line fetches
- Allow the modification of the `NSFetchRequest` method when request optimizations are needed

The repository containing the source code of Magical Record and some documentation on using it is available at <https://github.com/magicalpanda/MagicalRecord>. To install it via CocoaPods, just add the pod `MagicalRecord` in your `Podfile` and hit **pod install**.

After installing the library, one will need to include its header in a project. One of the preferred ways is to include it in the project's `Prefix.pch` file:

```
#ifndef COCOAPODS_POD_AVAILABLE_MagicalRecord
#import "CoreData+MagicalRecord.h"
#endif
```

 Adding `#define MR_SHORTHAND` before `#import` will add the ability to skip `MR_` prefix in Magical Record methods.

Using Magical Record with a connection to RestKit requires a few tricks. As we know, RestKit is responsible for creating and managing Core Data's managed object model, store, and context. So we need to tell this information to a Magical Record. But the methods to set the default context and store coordinator are private. We will use the power of the Objective-C class extension (categories) to expose access to the private setters of Magical Record. Just be aware that as methods are private, the code can be broken in future versions of the library.

Add these lines before your implementation code:

```
#ifndef COCOAPODS_POD_AVAILABLE_MagicalRecord
// Use a class extension to expose access to MagicalRecord's private
setter methods
@interface NSManagedObjectContext ()
+ (void)MR_setRootSavingContext: (NSManagedObjectContext *) context;
+ (void)MR_setDefaultContext: (NSManagedObjectContext *) moc;
@end
@interface NSPersistentStoreCoordinator ()
+ (void)MR_setDefaultStoreCoordinator: (NSPersistentStoreCoordinator *)
coordinator;
@end
#endif
```

As we also can see here, we are using defines to use this code only if Magical Record is installed. The `COCOAPODS_POD_AVAILABLE_MagicalRecord` define is exposed by CocoaPods in `Pods-environment.h` when it installs the Magical Record library into your project. Bear in mind that you need to add `Pods-environment.h` to your `.pch` file to use this ability. Here's how to do it:

```
// in beginning of projects Prefix.pch file
#import "../Pods/Pods-environment.h"
```

Now you can rely on `COCOAPODS_POD_AVAILABLE_POD_NAME` to check on the compile time if a certain library with `POD_NAME` is installed. And if so, enable the code related to it.

Let's now change a bit in our RestKit's Core Data setup method and add these lines for configuring Magical Record:

```
// After
[managedObjectStore createManagedObjectContexts];

// Add
#ifdef COCOAPODS_POD_AVAILABLE_MagicalRecord
// Configure MagicalRecord to use RestKit's Core Data stack
[NSPersistentStoreCoordinator MR_setDefaultStoreCoordinator:managedObjectStore.persistentStoreCoordinator];
[NSManagedObjectContext MR_setRootSavingContext:managedObjectStore.persistentStoreManagedObjectContext];
[NSManagedObjectContext MR_setDefaultContext:managedObjectStore.mainQueueManagedObjectContext];
#endif
```

This will tell MagicalRecord to use RestKit's Core Data setup, and you can start using it in your code; for example, to create a `MCollection` entity, one will just use `[MCollection MR_createEntity]`, or to create `NSFetchedResultsController`:

```
fetchedResultsController = [MCollection
    MR_fetchAllSortedBy:@"name"
    ascending:YES
    withPredicate:fetchPredicate
    groupBy:@"DatabaseID"
    delegate:self];
```

Mogenerator

`mogenerator` is a command-line tool that, given an `.xcdatamodel` file, will generate two classes per entity. The first class, `_MyEntity`, is intended solely for machine consumption and will be continuously overwritten to stay in sync with your data model. The second class, `MyEntity`, a subclass of `_MyEntity`, won't ever be overwritten and is a great place to put your custom logic.

Mostly, it is useful if a Core Data model is rapidly changing, and you have your own entity helper methods. In addition to speed up generation, you will just fire a `mogenerator` command, or even create a prebuild script.

To install `mogenerator`, one can use the Homebrew package manager (instructions on how to install it can be found on its home page available at <http://mxcl.github.io/homebrew/>):

```
brew install --HEAD mogenerator
```

`--HEAD` here means that we want to install `mogenerator` from the latest source code. Omitting the `--HEAD` option will install the latest release version.

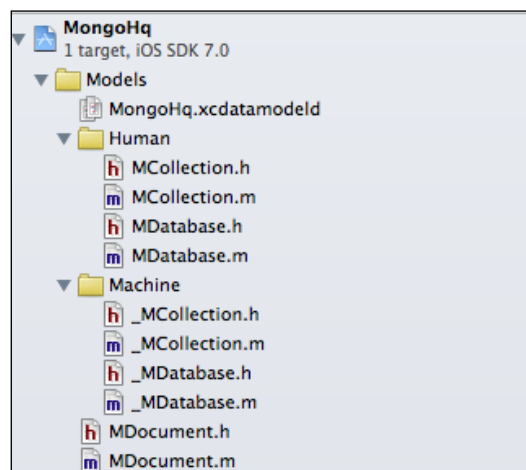
A script that one can use to generate classes is quite simple:

```
#!/bin/sh -x
MODEL_NAME="MongoHq"
MODEL_DIR="Models"
/usr/local/bin/mogenerator --template-var arc=true \
  -m ${MODEL_DIR}/${MODEL_NAME}.xcdatamodeld/*.xcdatamodel \
  -M ${MODEL_DIR}/Machine/ \
  -H ${MODEL_DIR}/Human/ \
  --includeh=${MODEL_DIR}/models.h
```

Here, `-m` is the link to the `xcdatamodel` file, `-M` is where to output the machine-generated files (which will be regenerated next time), and `-H` is where to output the human-generated classes (they will be preserved the next time the generator will be run). Adding the `--includeh` option will generate one header file that imports all machine and human-generated files, so you can import only this one in your project for easiness.

`--template-var` specifies custom template variables. In our case, we ask `mogenerator` to use ARC-enabled templates.

Save it to an executable shell file (Ex: `gen_mo.sh`), run it (`sh gen_mo.sh`), and add the generated files (once) to the Xcode project. You might end up having the kind of project tree shown in the following screenshot:



A project tree with added generated files

`mogenerator` has a few predefined templates with various properties (such as ARC/non-ARC and enabling modern syntax). You can also create your own template and ask `mogenerator` to use it.

For more information, consider visiting `mogenerator`'s home page:

<http://rentzsch.github.io/mogenerator/>.

Mapping

The configuration of mapping for Core Data objects is quite similar to normal mapping configuration, though there are a few differences. First, initializing of mapping is done differently. So if we had:


```
RKObjectMapping *mapping = [RKObjectMapping  
mappingForClass:itemClass];
```

We need to change it to:

```
// Get default managed object store  
RKManagedObjectStore *managedObjectStore = [RKManagedObjectStore  
defaultStore];  
  
// Create mapping for entity  
RKEntityMapping *mapping = [RKEntityMapping  
mappingForEntityForName:@"Database"  
inManagedObjectStore:managedObjectStore];
```

Additionally, it's a good practice to tell RestKit how to distinguish the object's uniqueness by providing identification attributes:

```
// How to identify if the object we got is in database  
// Here, we identify by name.  
mapping.identificationAttributes = @[@"name"];
```

 By default, if `identificationAttributes` are not set, RestKit tries to identify entities by `entityClassID`, `entityClass_id`, `identifier`, `id`, `ID`, `URL`, `url`.

If we have a relationship, like `MCollection` has, we will additionally provide information on relationship mapping:

```
[mapping addConnectionForRelationship:@"database" connectedBy:@  
{"databaseID": @"name"}];
```

Here, `database` is our relation, and `@"databaseID": @"name"` is the information on how to map it. The `databaseID` property is in the current object and `name` is the property in the database object.

However, if we will look in the response JSON, there's no place where we can get the `databaseID` information. So how we can do this? If we are using routes, we are providing `databaseID` while constructing a path from it to get collections. RestKit saves this information into a request metadata, which we can use for our mapping. So in our case, the `databaseID` parameter that was used in constructing the request path can be found at `@metadata.routing.parameters.databaseID`. So the code to map our `databaseID` will be:

```
NSDictionary *dbIdMapping = @{@"@metadata.routing.parameters.
databaseID": @"databaseID"};
[mapping addAttributeMappingsFromDictionary:dbIdMapping];
```

More about metadata mapping we will be discussed in *Chapter 4, Advanced stuff* in the section *Metadata mapping*.

For using Core Data for remote object creation, we will need to be sure that our database is in sync with the remote one. If we will assume that the remote service has a more trusted data in the database, we need to be sure we don't have any temporary (orphaned) objects in our local one.

RestKit can deal with orphaned objects by checking the local database against received objects. If the objects in the local database do not exist in the response array, RestKit will purge the local ones.

To support such functionality, we need to tell RestKit how to look for objects in the local database on requesting a specific remote path. To do so:

```
// Deleting orphaned objects
// Define Fetch request to trigger on specific url

[manager addFetchRequestBlock:^(NSFetchRequest * (NSURL *URL) {
    // Create a path matcher
    RKPathMatcher *pathMatcher = [RKPathMatcher pathMatcherWithPattern:itemsPath];

    // Dictionary to store request arguments
    // databaseID in our case is what we are looking for
    NSDictionary *argsDict = nil;

    // Match the URL with pathMatcher and retrieve arguments
    BOOL match = [pathMatcher matchesPath:[URL relativePath]
                  tokenizeQueryStrings:NO
                  parsedArguments:&argsDict];

    // If url matched, create NSFetchRequest
    if (match) {
```

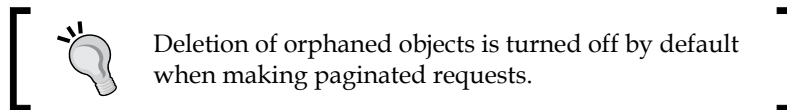


```
NSString *databaseID = [argsDict objectForKey:@"databaseID"];
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"databaseID = %@", databaseID];

NSFetchRequest *fetchRequest = [MCollection
    MR_requestAllSortedBy:@"name"
        ascending:YES
        withPredicate:predicate];
return fetchRequest;
}

return nil;
}];
```

When RestKit will receive objects from remote, it will fire the `fetch` request to see what objects for specific `databaseID` we have in our database. If there are objects that do not match any of the received, they will be deleted.



For more in-depth knowledge on how mapping is done consider checking RestKit unit tests. You can find all kinds of usage scenarios there.

Integrating with UI

Previously, we used `NSArray` to store the response data and to show it in `UITableView`. With Core Data entities, the best approach is to use `NSFetchedResultsController`. The benefit of using it is its ability of lazy loading objects, as well as responding to Core Data changes on fetched objects.

Let's modify our `GenericTableViewController` to support Core Data in addition to in-memory objects. We will start by defining a few additional properties:

```
@property (nonatomic, assign) BOOL useCoreData;
@property (nonatomic, assign) Class objectClass;
@property (nonatomic, strong) NSString *sortBy;
@property (nonatomic, strong) NSString *groupBy;
@property (nonatomic, strong) NSPredicate *fetchPredicate;

@property (nonatomic, strong) NSFetchedResultsController
    *fetchedResultsController;
```

A brief description of the properties we are using:

Property name	Type	Description
useCoreData	boolean	Assign YES to start using Core Data
objectClass	class	Class of objects that are shown in table will be of the NSManagedObject type
sortBy	string	How to sort objects
groupBy	string	How to group objects
fetchPredicate	predicate	Predicate for fetch (filtering)
fetchResultsController	fetches results controller	Fetches results controller to interact with data

Now we need to create the actual `NSFetchResultsController` method. We will use Magical Record's one-line method for this:

```

if (!self.sortBy) {
    self.sortBy = @"name";
}

self.fetchResultsController = [self.objectClass
    MR_fetchAllSortedBy:self.sortBy
    ascending:YES
    withPredicate:self.fetchPredicate
    groupBy:self.groupBy
    delegate:self];

if (self.fetchResultsController.fetchedObjects.count == 0) {
    [self refresh];
}

```

In our subclasses, the only change we need to make is to provide `objectClass` and (sometimes) `fetchPredicate`:

```

self.useCoreData = YES;
self.objectClass = [MCollection class];

// Specify predicate to filter collections
self.fetchPredicate = [NSPredicate predicateWithFormat:@"databaseID =
%@", self.database.databaseID];

```

Also, check that you are using routing:

```
self.routeName = @"collections";
self.routeObject = self.database;
```

If we are using Core Data and `NSFetchedResultsController`, we don't need to assign loaded objects to `self.items` and refresh the table. `NSFetchedResultsController` will notify us with its delegates about changes in Core Data. Therefore, the simplest delegate implementation could be:

```
- (void)controllerDidChangeContent:(NSFetchedResultsController *)
controller
{
    [self.tableView reloadData];
}
```

You can check the steps to integrate `NSFetchedResultsController` in `UITableView` in the provided source code example.

For our form, in order for it to work with Core Data, the only change we need to make is on how a new object is created:

```
// Create new collection, if it's a Create request
if (self.shouldCreateNewItem) {
    if ([MCollection isKindOfClass:[NSManagedObject class]]) {
        self.collection = [MCollection MR_createEntity];
        self.collection.database = self.database;
    } else {
        self.collection = [MCollection new];
    }
    self.collection.databaseID = self.database.databaseID;
}
```

Here, we create the entity and insert it in the database (with the help of Magical Record) and assign a database relation and the `databaseID` property.

Database seeding

In the context of Core Data, seeding means shipping your application with a persistent store pre-populated with default data. There are two ways to seed a database:

- On the initial app start, copy the pre-populated database from the application's bundle
- Seed the newly created database with data from JSON, XML, or another source

The second way is not the best, as seeding from a source will require parsing, mapping, and inserting operations that will slow down the application start, thus giving negative user experience. But this way can be used to create a seed database on the developer's machine.

You usually configure database seeding by copying an existing target to a new one that will be used to generate the seed database. The main differences between the targets are:

- `GENERATE_SEED_DB` is defined in the target's build section `Preprocessor macros`. This will trigger the seed database to be built.
- Seed source files are added to the target in order to be copied in the application bundle. So the database seeder can find them when running on a simulator.

The code for generating the seed database can look like:

```
#ifdef GENERATE_SEED_DB

NSError *error = nil;
BOOL success = RKEnsureDirectoryExistsAtPath(RKApplicationDataDirectory(), &error);
if (!success) {
    RKLogError(@"Failed to create Application Data Directory at path '%@': %@", RKApplicationDataDirectory(), error);
}

NSString *seedStorePath = [RKApplicationDataDirectory() stringByAppendingPathComponent:@"MOSeedDatabase.sqlite"];
RKManagedObjectImporter *importer = [[RKManagedObjectImporter alloc] initWithManagedObjectModel:managedObjectModel storePath:seedStorePath];

// Define seedMapping for data.json
[importer importObjectsFromItemAtPath:[NSBundle mainBundle] pathForResource:@"data" ofType:@"json"] withMapping:seedMapping keyPath:nil error:&error];

BOOL success = [importer finishImporting:&error];
if (success) {
    [importer logSeedingInfo];
} else {
    RKLogError(@"Failed to finish import and save seed database due to error: %@", error);
}
#endif
```

```
#else

...

// Complete Core Data stack initialization
NSString *seedPath = [[NSBundle mainBundle] pathForResource:@"MOSeedDa
tabase" ofType:@"sqlite"];
NSPersistentStore *persistentStore = [managedObjectStore addSQLit
ePersistentStoreAtPath:storePath fromSeedDatabaseAtPath:seedPath
withConfiguration:nil options:nil error:&error];

...

#endif
```

Indexing and searching

RestKit includes a component for easily indexing and searching Core Data entities. It can greatly help in implementing search functionality on a large database. This component is not included by default if RestKit is installed via CocoaPods. So in order to use it, you need to install it as a submodule. Add pod 'RestKit/Search' to the Podfile and hit **pod install** in a terminal.

Before using the indexer, it needs to be configured. You start by importing RestKit/Search.h headers and adding search indexing for each entity you plan to search:

```
RKManagedObjectStore *managedObjectStore = [[RKManagedObjectStore
alloc] initWithManagedObjectModel:managedObjectModel];

// Configure indexing for the Collection entity
NSArray *attributesToSearch = @[@"name"];
[managedObjectStore addSearchIndexingToEntityForName:@"Collection" onA
ttributes:attributesToSearch];

// some code
...

[managedObjectStore createManagedObjectContexts];

// Start indexing
[managedObjectStore startIndexingPersistentStoreManagedObjectContext];
```

You may notice that we are adding indexing to the entity just after initializing managed object store, and we start the actual indexing after creating managed contexts. The reason for this is that RestKit, on search initialization adds new tables to the application's SQLite database to hold search indexes and relation to entities. Once indexing is configured, `RKSearchIndexer` will observe our database for save notifications. On saving, any managed objects whose entities were configured for indexing will have their searchable attributes tokenized and stored as a to-many relationship to the `RKSearchWordEntity` entity.

Use any SQLite viewer to see how the database looks after adding indexes. The following screenshot demonstrates how the database looks using Base app:

TABLES	Z_PK	Z_ENT	Z_OPT	ZWORD
sqlite_master	1	3	1	mynewcollection
Z_1SEARCHWORDS	2	3	1	mycollection
Z_METADATA	3	3	1	spaces
Z_PRIMARYKEY	4	3	2	test
ZCOLLECTION	5	3	1	collection2
ZDATABASE				
ZRKSEARCHWORD				

Database preview in the Base app

The search is performed by creating `RKSearchPredicate` and assigning it as a predicate property to `NSFetchRequest`:

```
// Construct the predicate.
// Supported predicate types are
// NSNotPredicateType, NSAndPredicateType, and NSOrPredicateType

NSCompoundPredicateType type = NSAndPredicateType;
NSCompoundPredicate *searchPredicate = (id)[RKSearchPredicate searchPr
edicateWithText:textToSearch type:type];

NSFetchRequest *fetchRequest = [NSFetchRequest fetchRequestForEntityWi
thName:@"Collection"];
fetchRequest.predicate = searchPredicate;

NSError *error = nil;
NSArray *matches = [managedObjectStore.mainQueueManagedObjectContext
executeFetchRequest:fetchRequest error:&error];

NSLog(@"Found matching objects: %@", matches);
```

In our application example, we can implement the search functionality for the Collection table. In our case, we are using `NSFetchedResultsController` for making a fetch, and additionally, the collection objects are filtered for a specific database. Thus, we will need to modify the predicate to include this filtering before firing the search. So the search action (which will be called from `UISearchBar searchBarSearchButtonClicked` delegates) will be:

```
- (void)searchWithText:(NSString *)textToSearch
{
#ifdef COCOAPODS_POD_AVAILABLE_RestKit_Search

    // The filtering predicate
    NSPredicate *parentPredicate = [NSPredicate predicateWithFormat:@"
databaseID = %@", self.database.databaseID];

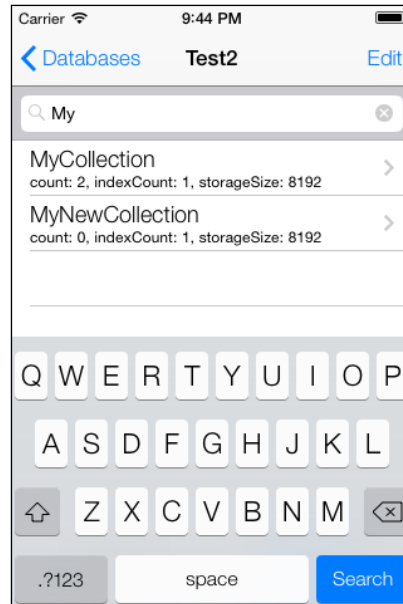
    NSCompoundPredicateType type = NSAndPredicateType;
    NSCompoundPredicate *searchPredicate = (id) [RKSearchPredicate searchPredicateWithText:textToSearch type:type];

    // Create mutable copy of sub-predicates
    // To add our filtering predicate to it
    NSMutableArray *subpredicates = [searchPredicate.subpredicates
mutableCopy];
    [subpredicates addObject:parentPredicate];

    // Assign it to our fetchPredicate, which is used by Fetched
Results Controller
    self.fetchPredicate = [[NSCompoundPredicate alloc]
initWithType:type subpredicates:subpredicates];

    // Reconstruct the Fetched Results Controller
    // Perform the fetch and reload the data in table
    [self constructFetchedResultsController];
    [self.tableView reloadData];
#endif
}
```

After adding the code to your project and running it, you can see the result as shown in the following screenshot:



Local search on collections

For additional performance improvements, you can check RestKit's official documentation and wiki on creating `RKEntityByAttributeCache` for caching the search indexes, and how to create and use `indexingContext` for speeding uploading data that contains a large number of searchable entities.

Summary

In this chapter, we have seen how RestKit helps connecting your local Core Data database with the remote web services. We also discussed a few handy libraries that can help in developing Core Data apps and checking integration with UI as well as search helpers.

The next chapter will cover some advanced techniques that can be useful in your app development.

4

Advanced Stuff

After reading the information provided in the previous chapters, you should now have enough knowledge in making apps that make heavy usage of web services. However, often you will find yourself searching topics on how to perform and implement things that are more advanced. Such things can include advanced mapping techniques, monitoring the reachability status in your application, helping in debugging the app, optimizing it, or performing user authorization.

This chapter covers some of the more advanced features of RestKit and `AFNetworking` libraries that developers might use for everyday app development and will help in overcoming potential bottlenecks.

Reachability

It's nice to be aware of the current Internet reachability, especially reachability to your web service endpoint. Knowing this, you can skip showing an "infinite" loading indicator if there is no connection, and warn the user if he's on cellular and a large file needs to be downloaded.

The reachability functionality is provided in iOS by Apple, but to work with it requires some coding. Hopefully, there are shortcut methods for using it provided by the `AFNetworking` library. To use them, first you need to link against the `SystemConfiguration` framework. Add it in the active target's Link Binary with Library build phase and add the following line in your `.pch` file:

```
#import <SystemConfiguration/SystemConfiguration.h>
```

For a global reachability test, you may use a `setReachabilityStatusChangeBlock` method from `AFHTTPClient` to show an alert message to the user. You need to provide a callback block object that will be executed when the network availability of the `baseURL` host will change. The status block argument represents the various reachability states between a client and a `baseURL`. The status can be one of `[AFNetworkReachabilityStatus] Unknown, NotReachable` (no internet), `ReachableViaWWAN` (cellular connection), `ReachableViaWiFi` (Wi-Fi connection).

For example, we can show the user the following message if there's no Internet connection:

```
[manager.HTTPClient setReachabilityStatusChangeBlock:^(AFNetworkReachabilityStatus status) {
    if (status == AFNetworkReachabilityStatusNotReachable) {
        UIAlertView *alert = [[UIAlertView alloc]
            initWithTitle:nil
            message:@"There is no network connection!"
            delegate:nil
            cancelButtonTitle:@"Dismiss"
            otherButtonTitles:nil];
        [alert show];
    }
}];
```

Sometimes, it's better to check if the user is on Wi-Fi before downloading the huge chunk of data, so that the network operator will not charge the user. To do so, we can use the `AFHTTPClient`'s `networkReachabilityStatus` property as follows:

```
AFHTTPClient *client = [RKObjectManager sharedManager].HTTPClient;
if (client.networkReachabilityStatus !=
    AFNetworkReachabilityStatusReachableViaWiFi) {
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Not a WiFi connection"
        message:@"The data is available over cellular connection
only. Additional charges may occur. Continue?"
        delegate:self
        cancelButtonTitle:@"Cancel"
        otherButtonTitles:@"Yes", nil];
    [alert show];
}
```

Here, we are checking if the user is on a Wi-Fi; and if he's not, we will show him a confirmation message as shown in the following screenshot:



Warning on cellular network reachability

Logging

Sometimes we might come across a situation where we don't understand how the request was made, or why the mapping failed, or if RestKit inserted that data in the Core Data or not. For such purposes, RestKit has powerful logging abilities, which are based on the `LibComponentLogging` framework. It supports six levels of logging (Off, Critical, Error, Warning, Info, Debug, Trace), which can be configured for every RestKit's component individually. You can turn these levels for such components as shown in the following table:

Component	Note
App	Logging for usage in your app
RestKit	All the RestKit components
RestKit/Core Data	Core Data logging

Component	Note
RestKit/Core Data/Cache	Logging of cache usage in Core Data component
RestKit/Network	Network logging of requests/responses
RestKit/Network/Core Data	Logging when Core Data is involved in network requests
RestKit/ObjectMapping	Logging of mapping operations
RestKit/Search	Search component
RestKit/Support	Miscellaneous logging that is not in other components
RestKit/Testing	Logging of testing component, which provides a few helpers
RestKit/UI	Currently, the UI part is extracted from a main Restkit distribution, but was presented mostly by a special TableViewController

So, for example, if you want to switch logging, use the `RKLogConfigureByName` method and provide the component and logging level for it, as given in the following code:

```
// Log debugging messages from the Network component
RKLogConfigureByName("RestKit/Network", RKLogLevelDebug);

// Log only critical messages from the Object Mapping component
RKLogConfigureByName("RestKit/ObjectMapping", RKLogLevelCritical);
```

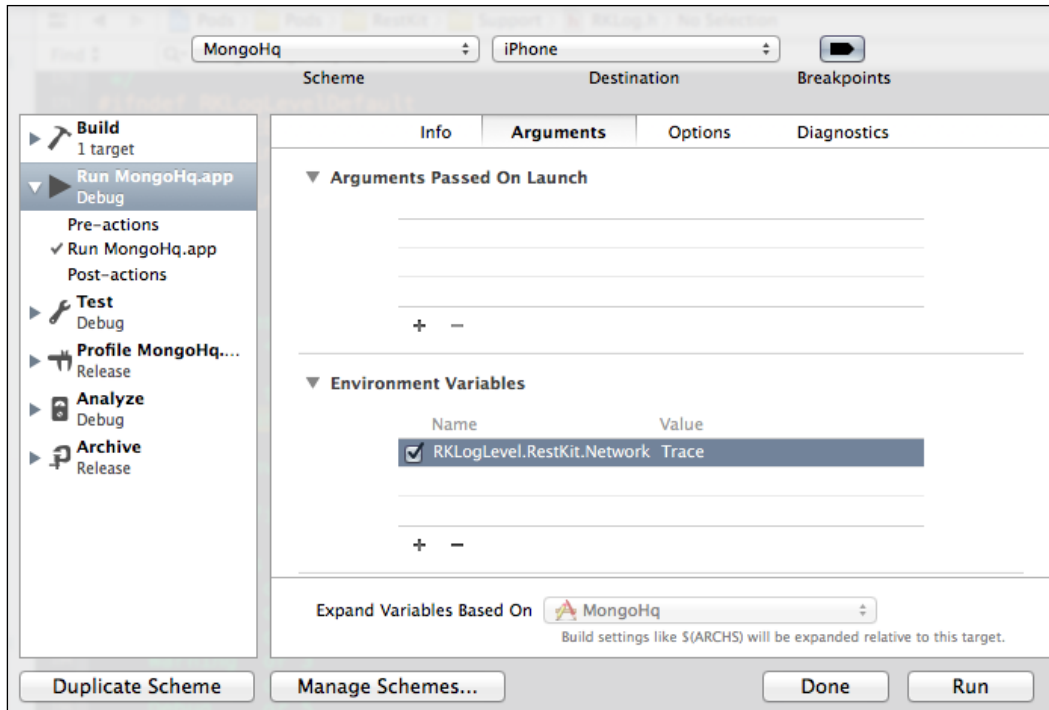
In addition, it is possible to set logging as environmental flags:

Log level	Environment variable value
<code>RKLogLevelOff</code>	0 or Default
<code>RKLogLevelCritical</code>	1 or Critical
<code>RKLogLevelError</code>	2 or Error
<code>RKLogLevelWarning</code>	3 or Warning
<code>RKLogLevelInfo</code>	4 or Info
<code>RKLogLevelDebug</code>	5 or Debug
<code>RKLogLevelTrace</code>	6 or Trace

To do so, place in the following code (preferably just before using RestKit) and configure RestKit logging from environment variables:

```
// MongoHqApi.m
// in - (id)init
RKLogConfigureFromEnvironment();
```

Open the **Scheme Editor** (use *Option + Command + R* to set the environment variables prior to run) and click on the **Arguments** tab. Define an environment variable named `RKLogLevel.RestKit.Network` and set its value to **Trace**, as shown in the following screenshot:



Setting environment variables in Xcode

This will configure the logging to the equivalent of setting the following in code:

```
RKLogConfigureByName("RestKit/Network", RKLogLevelTrace);
```

Bear in mind that using environmental variables for configuring the logging is useful only while debugging your app. For production release, set the logging via code to the minimum required.

Error mapping

If the server is capable of returning proper error messages in response, we will be able to map them and output directly to the user. So, if the server is capable of returning error message in the JSON response as follows:

```
{
  "error": "No matching database associated with this account",
  "code": 1005
}
```

Then, we can use it to show to the user. To do so, create an error mapping for the `RKErrorMessage` class as follows:

```
// You can map errors to any class
// RKErrorMessage is included within RestKit
RKObjectMapping *errorMapping = [RKObjectMapping
mappingForClass:[RKErrorMessage class]];

// Map error information to the errorMessage property in our class
[errorMapping addPropertyMapping:[RKAttributeMapping
attributeMappingFromKeyPath:nil toKeyPath:@"errorMessage"]];

// Anything in 4xx (Client errors)
NSIndexSet *clientErrorStatusCodes = RKStatusCodeIndexSetForClass(RKSt
atusCodeClassClientError);

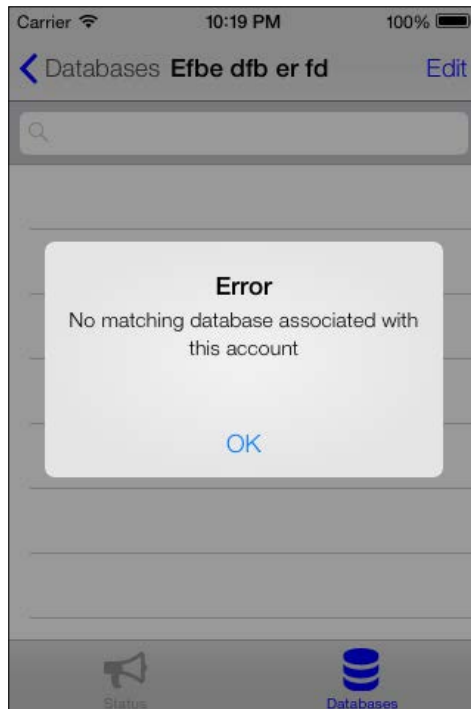
// Anything in 5xx (Server errors)
NSIndexSet *serverStatusCodes = RKStatusCodeIndexSetForClass(RKStatusC
odeClassServerError);

// Combine
NSMutableIndexSet *statusCodes = [[NSMutableIndexSet alloc] init];
[statusCodes addIndexes:clientErrorStatusCodes];
[statusCodes addIndexes:serverStatusCodes];

// Any response within provided status code range
// with an "error" key path
RKResponseDescriptor *errorDescriptor = [RKResponseDescriptor
responseDescriptorWithMapping:errorMapping
method:RKRequestMethodAny
pathPattern:nil
keyPath:@"error"
statusCodes:statusCodes];
```

```
// Add it to default response mappers
RKObjectManager *manager = [RKObjectManager sharedManager];
[manager addResponseDescriptor:errorDescriptor];
```

Now outputting `[error localizedDescription]` will give us an error message from the server, as shown in the following screenshot:



Error as returned from a web service

Metadata mapping

The mapping operation provides support for mapping a dictionary of metadata in addition to the source object. This metadata is made available by mapping key paths nested under a specially designated parent key (`@metadata`) that cannot exist in a source representation. The metadata is also nested under subkeys to effectively namespace usage between components. An example of metadata mapping would be getting an object index from the collection it came from. To do so, we can write the following code:

```
[mapping addAttributeMappingsFromDictionary:@{
    @"@metadata.mapping.collectionIndex": @"index"
}];
```


The `@metadata` prefix indicates that the property is to be mapped from the metadata dictionary instead of from the source object representation. If any relationships were mapped, it would have access to this same metadata information as well. The possible metadata information that can be available for mapping is shown in the following table:

@metadata. suffixes	Description
<code>mapping.collectionIndex</code>	This is an <code>NSNumber</code> object specifying the index of the current object within a collection being mapped. It is available if the current representation exists within a collection.
<code>mapping.rootKeyPath</code>	This is an <code>NSString</code> object specifying the root key path at which the current representation is nested within the source representation.
<code>mapping.parentObject</code>	This is a direct parent object of the object that is currently being mapped. This key is only available for objects that are mapped as relationships of a parent object.
<code>routing.parameters</code>	This is a dictionary of parameters and their values that were used in the path pattern of the <code>RKRoute</code> object to construct the request. It is available when routing will be used to construct the request URL.
<code>routing.route</code>	This is the route object used to construct the request URL.
<code>HTTP.request.URL</code>	This is an <code>NSURL</code> object identifying the URL of the request that loaded the response.
<code>HTTP.request.method</code>	This is an <code>NSString</code> object specifying the HTTP method of the request.
<code>HTTP.request.headers</code>	This is an <code>NSDictionary</code> object containing all HTTP headers and values for the request that loaded the response.
<code>HTTP.response.URL</code>	This is an <code>NSURL</code> object identifying the URL of the response. It is often the same as in the request URL.
<code>HTTP.response.headers</code>	This is an <code>NSDictionary</code> object containing all HTTP headers and values for the response.

There are two other keys in the mapping dictionary that are used for metadata retrieval:

- `@parent`: This key returns the direct parent node of `sourceObject` being mapped or `nil` if `sourceObject` is itself a root node. Parent access can be chained to traverse upward all the way to the root node of the representation.

- `@root`: This key returns the root node of the representation being mapped. When a large JSON document is being mapped with `RKMapperOperation`, this will point to the parsed JSON document that was used to initialize the operation.

Advanced object mapping techniques

There might be cases when you need to make your own custom logic for a specific mapping. One of the possible ways of doing this is to define mapping to a setter, where you will process the response dictionary yourself.

For our example, as MongoDB saves the `id` of a document in its body. We will use such techniques to extract the `id` attribute from the body. First, let's look at the mapping configuration:

```
Class itemClass = [MDocument class];
RKObjectMapping *mapping = [RKObjectMapping
    mappingForClass:itemClass];
[mapping addPropertyMapping:[RKAttributeMapping
    attributeMappingFromKeyPath:nil
    toKeyPath:@"rootDocument"]
];
```

Now, `rootDocument` in our `MDocument` class is just a property with a custom setter (or you can create a custom setter method that accepts additional arguments, in case you consider that using properties for this is not the best practice):

```
// In @interface MDocument

// The fake setter
@property (nonatomic, assign) NSDictionary *rootDocument;

// The real property to hold the document body
@property (nonatomic, strong) NSDictionary *document;

// In @implementation

- (void)setRootDocument:(NSDictionary *)rootDocument {

    if (!rootDocument) return;

    // Extract document _id
    id idObj = rootDocument[@"_id"];
```

```
    if ([idObj isKindOfClass:[NSString class]]) {
        self.documentID = idObj;
    }
    else if ([idObj isKindOfClass:[NSDictionary class]]) {
        static NSString *oidKey = @"$oid";
        NSDictionary *idObjDict = idObj;
        self.documentID = idObjDict[oidKey];
    }

    // Remove _id from document
    NSMutableDictionary *dict = [rootDocument mutableCopy];
    [dict removeObjectForKey:@"_id"];
    self.document = dict;
}
```

The other possible use of such techniques is to make a data validation.

Batching operations

Suppose we want to import and upload a CSV file in our remote database. An example can be **IMDB (International Movie Database)**, the top 250 movies CSV table, each row containing title, rating, and the number of votes. We need to post every single movie as a document in our collection as follows:

```
...
241,8,Mystic River (2003),"234,310"
242,7.9,Manhattan (1979),"71,126"
243,7.9,The Untouchables (1987),"152,716"
244,7.9,"Spring, Summer, Fall, Winter... 38,532"
245,7.9,Nosferatu (1922),"49,200"
246,7.9,The Celebration (1998),"43,066"
247,7.9,Three Colors: Red (1994),"39,541"
248,7.9,Big Fish (2003),"263,070"
...
```

Luckily, RestKit has a batch operation method. So to enqueue a batch of object request operations, we would write the following code:

```
RKObjectManager *manager = [RKObjectManager sharedManager];

// A route to be used for every single upload operation
RKRoute *route = [manager.router.routeSet routeForClass:[MDocument
class] method:RKRequestMethodPOST];

[manager enqueueBatchOfObjectRequestOperationsWithRoute:route
```

```

        objects:docs
        progress:^(NSUInteger numberOfFinishedOperations,
        NSUInteger totalNumberOfOperations)
    {
        NSLog(@"Finished %d operations", numberOfFinishedOperations);
        // Show some progress HUD
        CGFloat progress = ((CGFloat)numberOfFinishedOperations /
        (CGFloat)totalNumberOfOperations);
        [SVProgressHUD showProgress:progress];
    }
        completion:^(NSArray *operations)
    {
        NSLog(@"All Documents Uploaded!");
    }
    }];

```

Here, docs is an array of the locally created MDocument objects. In our example, after parsing a CSV file and getting an array of rows in parsedData, we went through each and created the pre-filled document:

```

NSArray *headerNames = nil;
NSMutableDictionary *dict = nil;
BOOL first = YES;

for (NSArray *arr in parsedData) {

    // First row is a header, so extract it
    if (first) {
        headerNames = arr;
        dict = [[NSMutableDictionary alloc]
        initWithCapacity:headerNames.count];
        first = NO;
        continue;
    }

    // Fill the dict with information from CVS
    for (int i=0; i < headerNames.count; i++) {
        dict[headerNames[i]] = arr[i];
    }

    MDocument *doc = [[MDocument alloc] init];
    doc.databaseID = self.collection.databaseID;
    doc.collectionID = self.collection.collectionID;

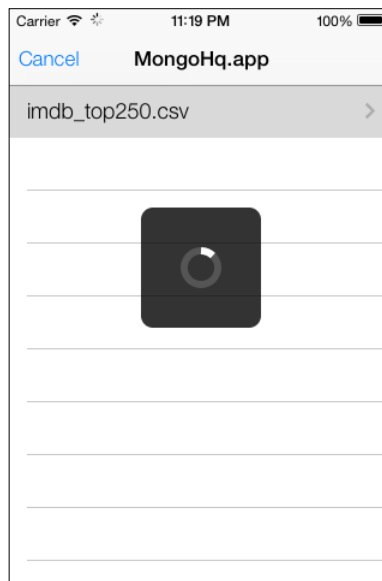
    doc.document = [dict copy];
}

```

```
// add to array of documents to be uploaded
[documents addObject:doc];

// clear the dict for reuse
[dict removeAllObjects];
}
```

Now, if we click on **Import** and select a CSV file, the import process will start, as shown in the following screenshot:



Uploading data from a CSV file

Paginating results

If our web service returns a collection of items and their number is quite high, it can be a good idea to paginate them showing a user no more than, let's say, 20 of them in a page. Of course to use this method, your server needs to support this functionality.

RestKit provides an `RKPaginator` class for such use cases. Currently, it supports only web servers that respond pagination information as part of JSON/XML of the content, for example:

```
{
  "pagination": {
    "per_page": 10,
    "total_pages": 25,
  }
}
```

```

        "total_objects": 250
    },
    "articles": [
        // Array of articles
    ]
}

```

Instances of `RKPaginator` will start a HTTP GET request to remote the web service and retrieve paginated collections of mappable data. Paginators rely on `RKObjectMappingProvider` to decide on how object mapping will be made for the retrieved data; therefore, they must be configured with a `paginationMapping` property, where we will specify how information should be mapped.

Pattern URLs are used while building a complete final URL for loading a paginated resource collection. They will be evaluated against the paginator object. For example, a pattern URL is as follows:

```
/articles?per_page=:perPage&page_number=:currentPage
```

When evaluated against the state of the paginator (which is configured with 100 objects per page and a current page number of 3), it will become:

```
/articles?per_page=100&page_number=3
```

For the preceding JSON example, mapping would be configured as follows:

```

RKObjectMapping *paginationMapping = [RKObjectMapping
mappingForClass:[RKPaginator class]];
[paginationMapping addAttributeMappingsFromDictionary:@{
    @"pagination.per_page":    @"perPage",
    @"pagination.total_pages": @"pageCount",
    @"pagination.total_objects": @"objectCount",
}];

```

We can also use `RKObjectManager` to help us in creating paginator, if the configuration is the same throughout the entire app:

```

// In configuration method
RKObjectManager *manager = [RKObjectManager sharedManager];
manager.paginationMapping = paginationMapping;

// In implementation

// Paginator path pattern
NSString *path = @"/articles?per_page=:perPage&page_
number=:currentPage";
RKPaginator *paginator = [manager paginatorWithPathPattern:path];
paginator.perPage = 20;

```

```
// What to do on completion
[paginator setCompletionBlockWithSuccess:^(RKPaginator *paginator,
NSArray *objects, NSUInteger page) {
    [self finishedLoadingWithItems:objects];
} failure:^(RKPaginator *paginator, NSError *error) {
    [self finishedLoadingWithError:error];
}];

// Load first page
[paginator loadPage:1];

// Load next page
[paginator loadNextPage];
```

Authorization

All authorizations are done through the AFNetworking library. The authorizations are mentioned in the following sections.

Basic

To authorize against the basic (or the simplest) authorization:

```
// client is instance of AFHTTPClient
[client setAuthorizationHeaderWithUsername:@"username"
password:@"password"];
```

Token-based

The token-based authorization is mostly used with OAuth 1.0 / OAuth 2.0 authorization schemes. After authorizing with them, you will get a token string, which you need to supply to your AFHTTPClient:

```
// client is instance of AFHTTPClient
NSString *token = @"1q2w3e4r";
[client setAuthorizationHeaderWithToken:token];
```

Here, the `setAuthorizationHeaderWithToken` method actually sets an HTTP header `Authorization` with the `Token token=1q2w3e4r` value. Sometimes, the web service you are using will require more custom HTTP headers to be set. For such cases, use `setDefaultHeader:value:` of AFHTTPClient:

```
[client setDefaultHeader:@"MongoHQ-API-Token" value:@"1q2w3e4r"];
```

OAuth 1.0

"OAuth is an open standard for authorization. OAuth provides a method for clients to access server resources on behalf of a resource owner (such as a different client or an end-user). It also provides a process for end-users to authorize third-party access to their server resources without sharing their credentials (typically, a username and password pair), using user-agent redirections." is the definition given in <http://en.wikipedia.org/wiki/OAuth>.

For authorization with OAuth 1.0, you can use the `AFOAuth1Client` library, which is an extension to `AFNetworking`. Check the <https://github.com/AFNetworking/AFOAuth1Client> for documentation.

You can install it with CocoaPods by including pod `'AFOAuth1Client'` in your Podfile.

For using it, register your application to launch from a custom URL scheme, and use that with the `/success` path as your callback URL:

```
AFOAuth1Client *oauthClient = [[AFOAuth1Client alloc]
    initWithBaseURL:[NSURL URLWithString:@"https://twitter.com/
oauth/"]
    key:@"APP_KEY"
    secret:@"APP_SECRET"];

// Your application will be sent to the background
// When the user authenticates, the app will be brought back using the
callback URL
[oauthClient
    authorizeUsingOAuthWithRequestTokenPath:@"/request_token"
    userAuthorizationPath:@"/authorize"
    callbackURL:[NSURL URLWithString:@"com.mongohq.app://success"]
    accessTokenPath:@"/access_token"
    success:^(AFOAuth1Token *accessToken) {
        NSLog(@"Success: %@", accessToken);
    }
    failure:^(NSError *error) {
        NSLog(@"Error: %@", error);
    }
];
```


OAuth 2.0

The AFOAuth2Client library is available as an extension for AFNetworking. It greatly simplifies the process of authenticating against an OAuth2 provider:

```
NSURL *url = [NSURL URLWithString:@"http://example.com/"];
AFOAuth2Client *oauthClient = [AFOAuth2Client clientWithBaseURL:url
clientID:kClientID secret:kClientSecret];

[oauthClient authenticateUsingOAuthWithPath:@"/oauth/token"
                        username:@"username"
                        password:@"password"
                        scope:@"email"
                        success:^(AFOAuthCredential
*credential)
{
    // A token is available in credential.accessToken variable

    // Store the credentials
    [AFOAuthCredential storeCredential:credential
withIdentifier:oauthClient.serviceProviderIdentifier];

}
failure:^(NSError *error)
{
    NSLog(@"Error: %@", error);
}];
```

The documentation can be found on a project's website at <https://github.com/AFNetworking/AFOAuth2Client>.

For installing with CocoaPods, use `pod 'AFOAuth2Client'`.

SSL and certificates

If you are connecting to the HTTPS web service and you want to use a custom SSL certificate (for example, bundled with your application) for such a connection, AFNetworking can help you with this. Just include the .cer files with your app bundle, and AFNetworking will automatically fetch and use them while communicating with your web service.

To use this mechanism, which is called **SSL Pinning**, you must enable it by inserting the following code in your `Prefix.pch` file and check whether the security framework is linked with the app binary:

```
#define _AFNETWORKING_PIN_SSL_CERTIFICATES_
```

SSL certificate pinning provides an increased level of security, by checking the server certificate validity against those specified in the app bundle.

Sometimes for testing and developing purposes, you can target an HTTPS web service, which uses self-signed server certificate. `AFNetworking` will disallow such a connection, as they are not secure. Whether the connection should accept an invalid SSL certificate, you need to enable in your request operation as follows:


```
// operation is a subclass of AFURLConnectionOperation
operation.allowsInvalidSSLCertificate = YES;
```

HTTP caching

There are three mechanisms to control a cache defined in HTTP standard, which are as follows:

- **Freshness:** This mechanism will allow the user to use a response without rechecking it on the original server. It can be controlled by both the server and the client. For example, the `Expires` response header will give a date when the document becomes stale, and `Cache-Control` (the max-age directive) will inform the cache how many seconds the response is fresh (valid) for.
- **Validation:** This mechanism may be used for checking whether a cached response is still good after it becomes stale. For example, if a `Last-Modified` header is in response, a cache can make a next request using the `If-Modified-Since` header and check if it has changed. The `ETag` (entity tag) mechanism also allows for both strong and weak validation.
- **Invalidation:** This mechanism mostly is a side effect when another request passes through the cache. So, if a URL related with a cached response later on gets a `POST`, `PUT`, or `DELETE` request, the cached response will be invalidated.

Instances of `RKObjectRequestOperation` support the entire HTTP caching facilities available in the `NSURLConnection` APIs. For caching to be enabled, the remote web server that the application is communicating with must emit the appropriate `Cache-Control`, `Expires`, and/or `ETag` headers. When the response headers include the appropriate caching information, the shared `NSURLCache` instance will manage responses and transparently add conditional `GET` support to cachable requests.

 Read more in detail about HTTP caching on Wikipedia at, http://en.wikipedia.org/wiki/Web_cache, and about RFC 2616 at <http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html>.

This allows network operations to be very lightweight. In the event, a managed object request operation loads a 304 'Not Modified' response for an HTTP request, it will use a cache for retrieving requested resources. It is even faster if Core Data is used, as no object mapping is performed assuming that Core Data contains a managed object representation of the resource requested.

An example of Ruby on Rails code that outputs ETag and Cache-Control information while returning a collection might be as follows:

```
def index
  @acts = parent.acts

  # Check ETag from request
  # Output 304 if not modified
  if stale?(:etag => @acts, :public => true)
    # if modified:

    # Output the collection
    index!

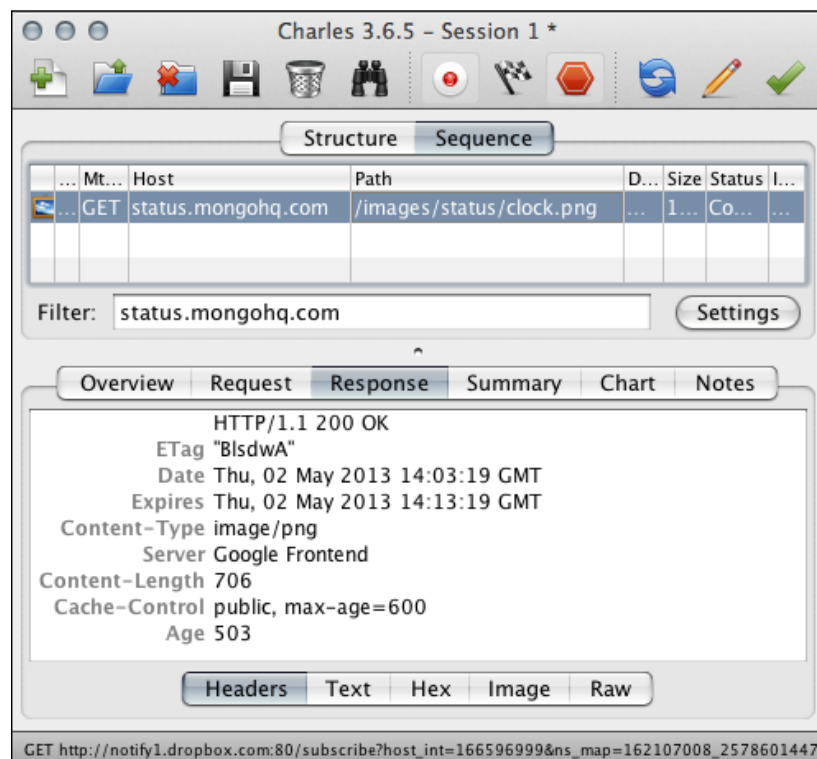
    # Append Cache-Control
    expires_in 5.minutes
  end
end
```

When setting `NSURLRequestUseProtocolCachePolicy` as a cache policy for `NSURLConnection`, the connection will start to support the conditional GET requests. This would result that `NSHTTPURLResponse` from loading the data and will not always have a 304 status code (Not Modified). So, if the response that we got from `NSURLCache` actually had a 304 status code, the object request operation will perform the following steps:

1. While loading the HTTP request, a reference to any existing `NSCachedURLResponse` is received.
2. After the response is loaded, if its HTTP request method is and the status code is one of 200, 304, 203, 300, 301, 302, 307, or 410, then the request is to be considered as cacheable.
3. In the previous step, where a request is examined as being cacheable, the Etag information of the current response is checked against the existing cache entry that was referenced before loading the request.
4. The data of the loaded response is matched against the data in cache if the Etag information will match.

5. On positive match, the `userInfo` dictionary from the cache entry is checked. If it has a `RKResponseHasBeenMappedCacheUserInfoKey` key and its Boolean value is `YES`, it indicates that the response was previously mapped.
6. If the preceding steps are positive, it will mean that a response is loaded from the cache and mapping or managed object deletion cleanup is skipped. Optimizations like this, where HTTP caching is used, are made to greatly improve the application responsiveness on network connections.

To see and debug how a web service returns responses on an HTTP request, different tools can be used. The most powerful one is a Wireshark (<http://www.wireshark.org>), which is a full featured network analysis tool, or a Charles (<http://www.charlesproxy.com>), which is a simpler HTTP/HTTPS analysis tool with a nice and clean interface and great usability features as shown in the following screenshot:



Showing detailed information about the request

Background processing

Sometimes we need to download large amount of data, or process a long operation. However, when the user clicks on the **Home** button on his iOS device, the iOS operating system will move the application from the active state to the inactive state. This can also occur for certain types of temporary interruptions, such as an incoming phone call or an SMS. The OS will notice the AppDelegate application's `applicationWillResignActive:` method before it begins to transition the application to the background state. Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games should use this method to pause the game.

`AFURLConnectionOperation` and its subclasses have the `pause` method, which will pause the operation, and `resume`, which will resume the operation. Depending on an operation class, the `pause/resume` method can be overridden and can be made to work differently. For example, in `AFURLConnectionOperation`, the `resume` method will restart the original request, while in `AFHTTPRequestOperation` (and its subclasses), since HTTP defines a specification for how to request a specific content range, this method will resume the request from where it left off.

For our situation, we could pause the downloading/uploading operation, and then resume it in the AppDelegate application's `applicationWillEnterForeground:` method, which is called as a part of the transition from the background state to the inactive state. `AFNetworking` provides a nice method, `setShouldExecuteAsBackgroundTaskWithExpirationHandler:`, for every request, which specifies that the operation should continue its execution after the app has entered the background state, and clean up on expiration handler. A handler will be called shortly before the application's remaining background time reaches 0.

```
- (void)applicationWillResignActive:(UIApplication *)application
{
    // operation is a subclass of AFHTTPRequestOperation
    [self.operation
 setShouldExecuteAsBackgroundTaskWithExpirationHandler:^(
     // handler is called synchronously on the main thread
     // will block application's suspension momentarily while the
     application is notified

     // Clean up anything that needs to be handled if the request times
     out
     [self.operation pause];
 }];
}
```

```

- (void)applicationWillEnterForeground:(UIApplication *)application
{
    [self.operation resume];

    // or create a custom resume
    // with OutputStream writing to file
}

```



You might also want to check `AFDownloadRequestOperation` (<https://github.com/steipete/AFDownloadRequestOperation>), which is a progressive download operation that has additional support to resume a partial download, to use a temporary directory, and has a special block that helps in calculating the correct download progress.

Custom HTTP client

Usually, you will use `AFHTTPClient` for most of the cases. However, sometimes you will need a custom one. For example, if you need to provide `apikey` in every request. One way would be to append it to each request just before dispatching it, but to support the DRY philosophy, we'll rewrite each request URL in a custom HTTP client class. To do so, create a subclass of `AFHTTPClient` as follows:

```

@interface MongoHqHTTPClient : AFHTTPClient
@end

@implementation MongoHqHTTPClient

- (NSMutableURLRequest *)requestWithMethod:(NSString *)method
                                   path:(NSString *)path
                                   parameters:(NSDictionary *)parameters
{
    // Consider moving API key to a more secure place
    NSString *apiKey = @"PUT_YOUR_API_KEY_HERE";
    path = [path stringByAppendingFormat:@"?_apikey=%@", apiKey];

    return [super requestWithMethod:method
                               path:path
                               parameters:parameters];
}

@end

```

Then initialize it by using it in your app by modifying the beginning of the `setupObjectManager` method as follows:

```
// Set the base Url
NSString *baseUrl = @"https://api.mongohq.com";

// Initialize our custom HTTP
MongoHqHTTPClient *httpClient = [[MongoHqHTTPClient alloc]
initWithBaseURL:[NSURL URLWithString:baseUrl]];

// Init with custom HTTPClient
RKObjectManager *manager = [[RKObjectManager alloc]
initWithHTTPClient:httpClient];
```

The rest of the code in the `setupObjectManager` method is same.

Summary

In this chapter, we have discussed the different advanced techniques that might be helpful to you while developing apps and working with different web services. We covered topics such as reachability, error mapping and logging, meta mapping, authorization, and a few others.

You will find the list of helpful resources, libraries, and links to check out for additional information, tips, and tricks in the Appendix of the book.



Helpful Resources

The following is a list of links to guides that might be useful for a developer to know more about RestKit and overall iOS development:

Home pages

- RestKit home page: <http://restkit.org>
- AFNetworking home page: <http://afnetworking.com>
- CocoaPods library manager home page: <http://cocoapods.org> – where you can search for available libraries
- A popular aggregator of custom controls for iOS and OS X: <http://cocoacontrols.com>
- <https://developer.apple.com/ios>: Apple Developer website for iOS with a lot of guides and latest iOS reference guides, as well as official developer forum

Where to get answers

- RestKit's Wiki: <https://github.com/RestKit/RestKit/wiki> – a web page with several useful guides
- RestKit Tag on StackOverflow: <http://stackoverflow.com/questions/tagged/restkit> – a place to ask specific questions and get relevant answers
- RestKit's Google Group: <https://groups.google.com/forum/#!forum/restkit> – where people have a general discussion about RestKit and its features
- A big FAQ about AFNetworking can be found on its Wiki: <https://github.com/AFNetworking/AFNetworking/wiki/AFNetworking-FAQ>

Guides and blogs

- And a few guides related to CocoaPods: <http://docs.cocoapods.org/guides/>
- A periodical about the best practices and advanced techniques in Objective-C: <http://www.objc.io>
- Cocoa Is My Girlfriend: <http://www.cimgf.com>—a popular blog about general iOS developing techniques
- Cocoa with Love: <http://www.cocoawithlove.com>—another lovely blog about Cocoa development
- NSBlog: <http://www.mikeash.com/pyblog/>—a blog by Mike Ash covering more in-depth topics of Objective-C
- Deallocated Objects: - <http://deallocatedobjects.com>—another interesting blog relating to software development
- <http://www.angelolloqui.com/blog>—a blog by Angel G. Olloqui covering some techniques on iOS development
- <http://kalapun.com/blog>—a blog by some person named Taras Kalapun, where he occasionally writes about iOS, Ruby, and some other stuff
- <http://www.idev101.com>—a website with structured tips and tricks for iOS developers
- <http://theonlylars.com>—another iOS-friendly blog with a lot of small guides
- <http://www.raywenderlich.com>—lots of iOS-related tutorials
- Cocoa Samurai: <http://cocoasamurai.blogspot.com>—a blog by Colin Wheeler
- Learn Cocoa—<http://cocoadevcentral.com>

B

Helpful Libraries

This chapter describes some popular libraries that might be useful for developing iOS apps. Most of them can be installed by CocoaPods, just add `pod 'NAME_OF_LIBRARY'` to your `Podfile` and hit `pod install` in a command line. The descriptions of libraries are taken from their authors. It's up to the reader to decide if the usage of a particular library is needed.

Core libraries

The following is a list of core libraries:

- **RestKit:** It is a framework for consuming and modeling RESTful web resources on iOS and OS X.
- **AFNetworking:** It is a delightful iOS and OS X networking framework.
- **MagicalRecord:** It is a super-awesome, easy-fetching framework for Core Data.
- **Injective:** It is the Cocoa/Cocoa Touch Dependency Injection framework with features for simpler TDD.
- **AGi18n:** It easily localizes your iOS apps by automatically extracting texts from code and XIB files into localizable strings.
- **SDWebImage:** It is an asynchronous image downloader with cache support with an `UIImageView` category (for ones not satisfied with `UIImageView+AFNetworking` functionality).
- **SSKeychain:** It is a simple Cocoa wrapper for the keychain that works on Mac and iOS.
- **LUKeychainAccess:** It is a wrapper for iOS keychain services that behaves just like `NSUserDefaults`.

- `FileMD5Hash`: It is a library for computing MD5 hashes of files with small memory usage.
- `ISO8601DateFormatter`: It is a Cocoa `NSFormatter` subclass to convert dates to and from ISO-8601-formatted strings. It supports calendar, week, and ordinal formats.
- `NSData+Base64`: It is Base64 for `NSData`.
- `TransitionKit`: It is a block-based State Machine API for Objective-C.
- `NSXtensions`: It is a collection of useful categories for standard Cocoa classes.
- `BlockKit`: It is a framework that block-ifies pieces of foundation and UIKit that are in desperate need.
- `Libxtoobjc`: The extended Objective-C library extends the dynamism of the Objective-C programming language to support additional patterns present in other programming languages (including those that are not necessarily object-oriented).

Debugging and logging

- `TestFlightSDK`: It is a TestFlight SDK for over-the-air beta testing and crash reporting.
- `Reveal-iOS-SDK`: See your application's view hierarchy at runtime with advanced 2D and 3D visualizations.
- `PonyDebugger`: It is a client library and a gateway server combination that uses chrome developer tools on your browser to debug your application's network traffic and managed object contexts.
- `CocoaLumberjack`: It is a fast and simple, yet powerful and flexible logging framework for Mac and iOS.
- `LibComponentLogging-Core`: It is a logging library that provides log levels, log components, and pluggable logging backends.
- `LibComponentLogging-NSLog`: The `LibComponentLogging` logging backend redirects logging to `NSLog`.
- `CBIntrospect`: Introspect is a tool for iOS that aids in debugging user interfaces built with UIKit. It communicates with a view introspector, a desktop app.

Unit testing

- **Kiwi:** It is a Behavior-Driven Development library for iOS development.
- **TKSenTestAsync:** It is a SenTest category with asynchronous support.
- **Expecta:** It is a matcher framework for Objective-C and Cocoa.
- **Nocilla:** It is a stunning HTTP stubbing for iOS. Testing HTTP requests has never been easier.
- **OCMock:** It is an Objective-C implementation of mock objects.
- **Calabash:** It is an automated testing technology for Android and iOS native and hybrid applications. The home page is available at <http://calaba.sh>.
- **Frank:** It is automated acceptance tests for native iOS apps. The home page is available at <http://www.testingwithfrank.com>.

User interface

- **UI7Kit:** It is a GUI toolkit to implement iOS7 look-and-feel UIKit under iOS5/iOS6. It is also supported for patching UIKit to UI7Kit in runtime.
- **SVProgressHUD:** It is a clean and lightweight progress HUD for your iOS app.
- **SIAalertView:** It is a UIAlertView replacement with block syntax and fancy transition styles.
- **PSTCollectionView:** It is open source and a 100 percent API-compatible replacement of UICollectionView for iOS4+.
- **QuickDialog:** It is a quick and easy dialog screen for iOS.
- **ISRefreshControl:** It is an iOS4-compatible UIRefreshControl.
- **ViewDeck:** It is an implementation of the sliding functionality found in the Path 2.0 or Facebook iOS apps.
- **BCGenieEffect:** It is an OSX-style genie effect inside your iOS app.
- **JSMessagesViewController:** It is a message's UI for iPhone and iPad.
- **CorePlot:** It is a Cocoa plotting framework for Mac OS X and iOS.
- **DDPageControl:** It is an easily customizable alternative to UIKit's UIPageControl.
- **MTStatusBarOverlay:** It is a custom iOS status bar overlay seen in apps, such as Reeder, Evernote, and Google Mobile App.
- **FontasticIcons:** It is an Objective-C wrapper for iconic fonts.

Other

- `ZBarSDK`: It is a QR and barcode scan library
- `ObjQREncoder`: It is an Objective-C QR encoder
- `CardIO`: It is an easy-to-use credit card scanning tool
- `CHCSVParser`: It is a proper CSV parser for Objective-C

Index

Symbols

--HEAD option 60
--includeh option 61
@metadata. suffixes
 HTTP.request.headers 80
 HTTP.request.method 80
 HTTP.request.URL 80
 HTTP.response.headers 80
 HTTP.response.URL 80
 mapping.collectionIndex 80
 mapping.parentObject 80
 mapping.rootKeyPath 80
 routing.parameters 80
 routing.route 80
<MTableObject> protocol 35
@parent key 80
.pch file 15
@root key 81
--template-var option 61

A

Add Collection button 36
AFDownloadRequestOperation
 URL 93
AFIncrementalStore
 URL 8
AFNetworking
 about 7, 8, 97
 FAQ, URL 95
AFNetworking home page
 URL 95
AFNetworking library 23
AFURLConnectionOperation 92
AGi18n 97

API Key
 getting 25
App component 75
Apple Developer website
 for iOS, URL 95
applicationWillEnterForeground:
 method 92
applicationWillResignActive: method 92
authorization
 about 86
 basic 86
 OAuth 1.0 87
 OAuth 2.0 88
 SSL and certificates 88, 89
 token-based 86

B

batching
 operations 82-84
BCGenieEffect 99
BlockKit 98
blog
 by Angel G. Olloqui, URL 96
 by Taras Kalapun, URL 96
 iOS-friendly blog, URL 96
blogs. *See* **guides and blogs**
BSON 16

C

Calabash 99
Caller 11
CardIO 100
CBIntrospect 98
Charles
 URL 91

- CHCSVParse** 100
- class routes** 44
- client error** 25
- Cocoa**
 - URL 96
- cocoacontrols**
 - URL 95
- COCOA IS MY GIRLFRIEND**
 - URL 96
- CocoaLumberjack** 38
- CocoaPods** 11
 - about 14
 - guides, URL 96
 - URL 12
- CocoaPods library manager home page**
 - URL 95
- COCOA SAMURAI**
 - URL 96
- Cocoa with Love**
 - URL 96
- collection API documentation** 37
- collection, attributes**
 - collectionID 56
 - count 56
 - databaseID 56
 - indexCount 56
 - name 56
 - storageSize 56
- collectionID attribute** 56
- collectionID parameter** 41
- components, RestKit**
 - core data 9
 - networking 9
 - object manager 9
 - object mapping 9
 - search 10
 - testing 10
- Core Data**
 - about 55
 - entities, indexing 68, 69
 - entities, searching 68-71
 - mapping, configuration 62
 - prerequisites, for using with RestKit 57, 58
- core data component** 9
- core libraries**
 - AFNetworking 97
 - AGi18n 97
 - BlockKit 98
 - FileMD5Hash 98
 - Injective 97
 - ISO8601DateFormatter 98
 - Libxobjc 98
 - LUKeychainAccess 97
 - MagicalRecord 97
 - NSData+Base64 98
 - NSXtensions 98
 - RestKit 97
 - SDWebImage 97
 - SSKeychain 97
 - TransitionKit 98
- CorePlot** 99
- count attribute** 56
- Create collection button** 37

D

- data**
 - entering, in forms 46-52
 - loading, from Status API endpoint 20-22
- database**
 - seeding 66
 - setting up 55
- database, attributes**
 - databaseID 56
 - hostname 56
 - name 56
 - plan 56
 - port 56
- databaseID attribute** 56
- databaseID parameter** 63
- databaseID property** 62
- DDPageControl** 99
- Deallocated Objects**
 - URL 96
- debugging and logging**
 - CBIntrospect 98
 - CocoaLumberjack 98
 - LibComponentLogging-Core 98
 - LibComponentLogging-NSLog 98
 - PonyDebugger 98
 - Reveal-iOS-SDK 98
 - TestFlightSDK 98
- Defines.h file** 56

Delete button
creating 50
deleteItemRequest method 49

E

environmental flags
logging, setting as 76
error mapping
about 78
creating, for `RKErrorMessage` class 78, 79
Expecta 99

F

fetch request 64
FileMD5Hash 98
finishedLoadingWithError method 35
finishedLoadingWithItems method 35
FontasticIcons 99
forms
data, entering 46-52
Frank 99

G

GENERATE_SEED_DB 67
GenericTableViewController
creating 32
GenericTableViewController class 46
guides and blogs 96

H

home page
AFNetworking home page, URL 95
cocoacontrols, URL 95
CocoaPods library manager home page,
URL 95
RestKit home page, URL 95
hostname attribute 56
HTTP caching
about 89, 90
freshness 89
invalidation 89
validation 89
HTTP client
custom 93, 94

HTTP.request.headers 80
HTTP.request.method 80
HTTP.request.URL 80
HTTP.response.headers 80
HTTP.response.URL 80

I

IMDB (International Movie Database) 82
implementation section 29
indexCount attribute 56
indexingContext 71
In-Memory object 11
iOS developers
tips and tricks, URL 96
iOS-related tutorials
URL 96
ISO8601DateFormatter 98
ISRefreshControl 99
itemPath property 48
item property 48
items property 33

J

JSMessagesViewController 99

K

KeyValueObjectMapping
URL 9
Key Value Observing. *See* KVO
Kiwi 99
KVO 11

L

LibComponentLogging-Core 98
LibComponentLogging framework 75
LibComponentLogging-NSLog 98
Libextobjc 98
logging
about 75
setting, as environment flags 76
log levels 76
LUKeychainAccess 97

M

Magical Record

- about 58-60
- source code, URL 58

MagicalRecord 97

MagicalRecord mappers

- URL 8

managerWithBaseURL method 23

Mantle

- URL 9

mapping

- configuring, for Core Data 62-64
- setup 29, 30

mapping.collectionIndex 80

mapping.parentObject 80

mapping.rootKeyPath 80

MCollection entity 60

MCollection object 42

MDatabase object 38

MDocument objects 83

metadata mapping

- @parent key 80
- @root key 81
- about 79

Mogenerator

- HEAD option 60
- includeh option 61
- template-var option 61
- about 60
- homepage, URL 61
- installing 60

mogenerator command 60

MongoDB

- about 15
- differentiating, from classical relational database 16

MongoHQ

- about 15
- server status, checking 16

MTableObject

- creating 35

MTStatusBarOverlay 99

N

name attribute 56

named routes 43

networking component 9

networkReachabilityStatus property 74

Nocilla 99

NSBlog

- URL 96

NSData+Base64 98

NSDateFormatter 20

NSDate property 20

NSFetchedResultsController 66, 70

NSURLConnection 7

NSURLRequest object 23

NSXtensions 98

O

OAuth 1.0 87

OAuth 2.0 88

Objective-C

- best practices, URL 96

object manager

- requests, sending with 28-31
- setting up 26, 27

object manager component 9

object mapping

- about 9
- advanced techniques 81, 82
- fundamentals 19, 20

objects

- related objects, requesting 36-38

ObjQREncoder 100

OCMock 99

P

paginating results 84, 85

paginationMapping property 85

parameters property 33

Parse

- URL 8

path property 33, 48

plan attribute 56

Podfile 11

Podspec files 11

pod update

- running 14

PonyDebugger 98

port attribute 56

PSTCollectionView 99

Q

QRootElement 49
QuickDialog 46, 99

R

reachability 73, 74
refresh method 33
relationship routes 44
Remote web service 11
requests
 sending, with object manager 28-31
REST API
 documentation 24
RESTful object
 manipulating 40-43
RestKit
 about 5-8, 97
 adding 11
 comparing, to other solutions 8
 components 9
 configuring, to use Core Data 57, 58
 example 5, 6
 integrating, with UI 32-36
 libraries, adding 11-15
 using with Core Data, prerequisites 57, 58
 working 10, 11
RestKit component 75
RestKit/Core Data/Cache component 76
RestKit/Core Data component 75
RestKit home page
 URL 95
RestKit/Network component 76
RestKit/Network/Core Data component 76
RestKit/ObjectMapping component 76
RestKit/Search component 76
RestKit's Google Group
 URL 95
RestKit/Support component 76
RestKit's Wiki
 URL 95
RestKit Tag on StackOverflow
 URL 95
RestKit/Testing component 76
RestKit/UI component 76
Reveal-iOS-SDK 98

RKEntityByAttributeCache 71
RKErrorMessage class 78
RKLogLevel.RestKit.Network 77
RKObjectManager class 23
RKObjectRequestOperation class 23
RKObjectRequestOperation object 18, 23
RKPaginator class 84
RKPathFromPatternWithObject function 38
RKSearchWordEntity entity 69
routeName property 46
routeObject property 46
router
 about 43
 class routes 44
 name 45
 named routes 43
 relationship routes 44
routing.parameters 80
routing.route 80

S

saveAction method 50
SDWebImage 97
search component 10
setReachabilityStatusChangeBlock
 method 74
shouldCreateNewItem property 48
shouldEscapePath 45
UIAlertView 99
SLRESTfulCoreData
 URL 9
slug attribute 41
SSKeychain 97
SSL certificate 89
SSL Pinning 88
Stashboard API
 documentation, URL 16
Status API endpoint
 data, loading from 20-22
 URL 16
StatusItem object 16
storageSize attribute 56
subtitleLabel method 36
SVProgressHUD 99
SystemConfiguration framework 73

T

TestFlightSDK 98
testing component 10
titleText method 36
TKSenTestAsync 99
token-based authorization 86
TransitionKit 98
Twilio
 URL 16

U

UI
 integrating with 64-66
UI7Kit 99
UIKit component 46
UITableView delegate method 35
unit testing
 Calabash 99
 Expecta 99
 Frank 99
 Kiwi 99
 Nocilla 99
 OCMock 99

TKSenTestAsync 99
updateItemRequest method 48
user interface
 BCGenieEffect 99
 CorePlot 99
 DDPageControl 99
 FontasticIcons 99
 ISRefreshControl 99
 JSMessagesViewController 99
 MTStatusBarOverlay 99
 PSTCollectionView 99
 QuickDialog 99
 UIAlertView 99
 SVProgressHUD 99
 UI7Kit 99
 ViewDeck 99

V

ViewDeck 99
viewWillAppear method 33

W

willStartLoading method 34

Z

ZBarSDK 100



Thank you for buying RestKit for iOS

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Flash iOS Apps Cookbook

ISBN: 978-1-84969-138-3 Paperback: 420 pages

100 practical recipes for developing iOS apps with Flash Professional and Adobe AIR

1. Build your own apps, port existing projects, and learn the best practices for targeting iOS devices using Flash.
2. How to compile a native iOS app directly from Flash and deploy it to the iPhone, iPad or iPod touch.
3. Full of practical recipes and step-by-step instructions for developing iOS apps with Flash Professional.



iOS Development Using MonoTouch Cookbook

ISBN: 978-1-84969-146-8 Paperback: 384 pages

109 simple but incredibly effective recipes for developing and deploying applications for iOS using C# and .NET

1. Detailed examples covering every aspect of iOS development using MonoTouch and C#/.NET
2. Create fully working MonoTouch projects using step-by-step instructions.
3. Recipes for creating iOS applications meeting Apple's guidelines.

Please check www.PacktPub.com for information on our titles

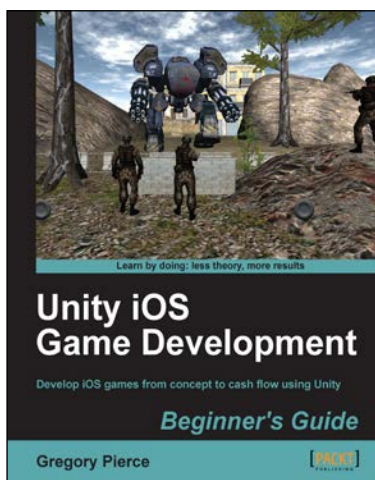


RubyMotion iOS Development Essentials

ISBN: 978-1-84969-522-0 Paperback: 262 pages

Create apps that utilize iOS device capabilities without learning Objective-C

1. Get your iOS apps ready faster with RubyMotion
2. Use iOS device capabilities such as GPS, camera, multitouch, and many more in your apps
3. Learn how to test your apps and launch them on the AppStore
4. Use Xcode with RubyMotion and extend your RubyMotion apps with Gems



Unity iOS Game Development Beginners Guide

ISBN: 978-1-84969-040-9 Paperback: 314 pages

Develop iOS games from concept to cash flow using Unity

1. Dive straight into game development with no previous Unity or iOS experience
2. Work through the entire lifecycle of developing games for iOS
4. Add multiplayer, input controls, debugging, in app and micro payments to your game

Please check www.PacktPub.com for information on our titles