Professional iOS Network Programming

Connecting the Enterprise to the iPhone® and iPad®

Jack Cox, Nathan Jones, John Szumski

PROFESSIONAL IOS NETWORK PROGRAMMING

PROFESSIONAL

iOS Network Programming

PROFESSIONAL

iOS Network Programming

CONNECTING THE ENTERPRISE TO THE IPHONE® AND IPAD®

Jack Cox
Nathan Jones
John Szumski

John Wiley & Sons, Inc.

Professional iOS Network Programming: Connecting the Enterprise to the iPhone® and iPad®

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published simultaneously in Canada

ISBN: 978-1-118-36240-2
ISBN: 978-1-118-38223-3 (ebk)
ISBN: 978-1-118-41716-4 (ebk)
ISBN: 978-1-118-53385-7 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2012948655

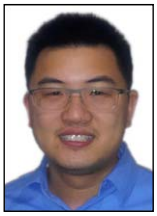
ABOUT THE AUTHORS

JACK COX is a software developer, a systems architect, and the director at CapTech Ventures, Inc., where he is responsible for the firm's mobile software practice. He has 30 years of experience in developing software for businesses of all sizes. He has been involved in three startups, holds multiple patents, and frequently presents to professional groups. He has a degree in computer science from Taylor University in Upland, Indiana. Jack lives in Richmond, Virginia, with his wife and children. You can get in touch with Jack on Twitter @jcox_mobile.

NATHAN JONES is a software engineer with expertise in iOS and experience in mobile web technologies. He began his career in enterprise software consulting and started exploring mobile development when Apple announced the capability to develop third-party apps for the iPhone. He graduated with a bachelor of science in business information technology with a concentration on decision support systems from Virginia Polytechnic Institute and State University in Blacksburg, Virginia. He currently resides in Richmond, Virginia with his wife, Jennifer, and son, Bryson. When he isn't working, writing, or playing with his son, he enjoys golfing and is an avid runner. You can get in touch with Nathan on Twitter @ nathanhjones.

JOHN SZUMSKI is a software engineer and mobile consultant with expertise in the iOS, Android, and mobile web platforms. He advises Fortune 500 companies on user experience and technical design. He graduated with a bachelor of science in computer science (with distinction) from the University of Virginia in Charlottesville, Virginia. John lives with his fiancée in Richmond, Virginia. You can get in touch with John on Twitter @jszumski.

ABOUT THE TECHNICAL EDITOR



JONATHAN TANG is a senior developer specializing in mobile applications at CapTech Consulting. He has more than 10 years of development experience, including programming touchscreen interfaces, medical devices, and iOS mobile applications. Prior to CapTech, John worked as the primary software engineer at a startup company that specializes in medical robotics. John received a bachelor of science in biomedical engineering from Johns Hopkins University and a master of science in electrical engineering from George Washington University.

CREDITS

EXECUTIVE EDITOR
Carol Long

PROJECT EDITOR
Victoria Swider

TECHNICAL EDITOR
Jonathan Tang

PRODUCTION EDITOR
Kathleen Wisor

COPY EDITOR
San Dee Phillips

EDITORIAL MANAGER
Mary Beth Wakefield

FREELANCER EDITORIAL MANAGER
Rosemarie Graham

ASSOCIATE DIRECTOR OF MARKETING
David Mayhew

MARKETING MANAGER
Ashley Zurcher

BUSINESS MANAGER
Amy Knies

PRODUCTION MANAGER
Tim Tate

VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER
Richard Swadley

VICE PRESIDENT AND EXECUTIVE PUBLISHER
Neil Edde

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Katie Crocker

PROOFREADER
Nancy Carrasco

INDEXER
Johnna VanHoose Dinse

COVER DESIGNER
Ryan Sneed

COVER IMAGE
© pagadesign/iStockPhoto

ACKNOWLEDGMENTS

I WANT TO THANK the principles, management, and coworkers at CapTech Ventures, especially Vinnie Schoenfelder, for encouraging and supporting our effort to write this book. I want to extend special thanks to Nathan Jones and John Szumski for being willing and faithful in this adventure to complete our first book. On behalf of Nathan, John, and myself, I want to thank Carol Long and Victoria Swider at Wiley for tolerating and answering all our newbie questions.

To my wife and family, I extend thanks without number for putting up with all of the nights and weekends of writing and the associated crankiness. Thank you for allowing me to fulfill this dream.

And most important, I extend thanks and praise to my savior, Jesus Christ, who, through His grace and mercy, has blessed me with so much that I do not deserve. Without Him, I would be hopeless and useless.

—Jack Cox

I WOULD LIKE TO THANK my lovely wife, Jennifer, and son, Bryson, for their continued support and patience while working on this book. There are times when I saw more of Xcode than I saw of you two, and those late nights and weekends weren't easy on you guys. That didn't go unnoticed, thank you. I would also like to thank my parents for encouragement throughout the process, and my dad, specifically, for teaching me to write my first program. That planted the seed. I still have that floppy disk, but I don't think I have a drive to read it.

—Nathan Jones

I WOULD LIKE TO THANK my beautiful fiancée, Caroline, for her understanding and support during many late nights spent writing or editing. I also appreciate my extended family's encouragement through the entire publishing process.

—John Szumski

CONTENTS

INTRODUCTION

AS IPHONES AND IPADS BECOME A UBIQUITOUS part of your personal and professional life, you become more and more dependent on their capability to seamlessly and flawlessly interact with hosts across the Internet or with other phones across the room. This book provides a compilation of methods to accomplish this level of connectivity with examples and best practices for each of these methods.

The release of the iPhone SDK, now known as iOS, started a stampede of experienced and novice developers rushing to develop apps for the iPhone. In this rush, many books have been written about how to develop for the iPhone. Most of these books have focused on developing user interfaces. This book does not follow that well-worn path. The sole focus of this book is the methods and best practices for connecting your iOS app to other systems; either network hosts or other mobile devices. If you have invested time and energy in learning the iOS development environment and are now looking for a way to build enterprise grade applications rooted in proved design patterns, then this book is for you.

For the past 15 years, website development has reigned supreme in enterprise IT departments. As the collective expertise with HTML, CSS, and JavaScript has increased, the collective expertise in interconnecting smart devices has decreased. As the development of mobile software has exploded over the past four years, the development community, both the experienced and the novice developers, have revisited and, in a way, relearned the practice of smart device interconnectivity.

As professional iOS developers working for numerous large clients, the authors of this book have discovered that developing and polishing the interconnect portion of an app can consume a significant portion, if not a majority, of the effort required to design, develop, and validate an app. They also found that the books available did not address this important aspect of iOS development. Therefore, this book can help both the novice and expert developer build better, more reliable, apps.

WHO THIS BOOK IS FOR

Enterprise iOS developers, including developers working within a corporation or organization, will find this book to be a valuable resource that provides working examples and guidelines for networking iOS apps with enterprise servers. The networking techniques described in this book belong in all developers' arsenals when writing iOS apps.

Beginning iOS developers transitioning from other platforms to iOS can gain a complete overview of the capabilities of iOS from this book. In addition, the working examples of these capabilities provide a foundation for networking features within their own apps. These developers should already have a working knowledge of Objective-C, XCode, and iOS app development fundamentals.

Enterprise system or application architects generating high-level designs encompassing mobile devices that span multiple corporate systems will find this book to be a valuable resource for understanding and exploiting the powerful networking capabilities of iOS devices. Chapters 1 through 5 are the most applicable to the enterprise architect.

Technical project managers and analysts can use this book to provide a solid technical foundation for planning app development projects and specifying app requirements. Chapters 1 through 5 and the introductory sections of each subsequent chapter are the most valuable to project managers and analysts.

For all types of technical readers, this book can provoke fresh ideas for novel, compelling features in your application. Because the book is written from the perspective of an enterprise developer, the app examples stick to themes that are common to traditional commercial organizations and applications. The examples do not delve into how to write games; instead they focus on tasks more commonly found within corporations. Networking techniques that are normally associated with leisure activities, such as peer-to-peer networking, do have application within the enterprise that can open new and valuable uses for mobile devices.

WHAT THIS BOOK COVERS

This book focuses on network programming of apps running on Apple's operating system for the iPhone, iPad, and iPod, called iOS. The topics covered include:

- Performing HTTP requests between client device and server
- Managing data payloads between client device and server
- Handling errors in HTTP requests
- Securing network communications
- Improving the performance of network communications
- Performing socket level communications
- Implementing push notifications
- Communicating between apps on a single device
- Communicating between apps on multiple devices

All the example apps and code snippets are written for iOS 5.0 and higher. The authors have chosen to focus on iOS 5 and later because the iOS customer base tends to update rapidly; therefore, the installed base of early iOS versions is small. Other mobile operations systems have slower adoption rates for new OS versions because each version must be approved by wireless carriers, which delay their rollout.

The server code examples provided by the book are developed in PHP or Perl running under Apache. These components were selected because they are readily available on Mac OS X, which is also required to run the iOS development environment.

HOW THIS BOOK IS STRUCTURED

The book is divided into four sections each covering a broad topic in the realm of iOS network programming. The sections progress from high-level discussions of mobile application architecture down to specific protocols and solutions for app-to-app communication, while providing in-depth coverage of the most popular methods of communicating between apps and servers.

Part I: Understanding iOS and Enterprise Networking

This is where most readers should start. This first section provides a high-level overview of iOS networking and architectural best practices for mobile networking.

Chapter 1: Introducing iOS Networking Capabilities — Chapter 1 reviews the basics of network programming and the APIs provided in iOS to connect devices to servers or to other devices.

Chapter 2: Designing Your Service Architecture — This chapter describes architectural patterns found to be beneficial for deploying device-friendly networked applications.

Part II: HTTP Requests: the Workhorse of iOS Networking

This section drills into the most common facility for communication between an iOS device and a server.

Chapter 3: Making Requests — Here you explore the ways to make HTTP requests from an iOS app, including code examples using the URL loading API.

Chapter 4: Generating and Digesting Payloads — This chapter examines and weighs the most common ways to encode information passed between an iOS app and a server, including code examples of XML, JSON, and HTML payload management.

Chapter 5: Handling Errors — Chapter 5 looks at error handling within the realm of HTTP requests and responses.

Part III: Advanced Networking Techniques

This section contains five chapters that address advanced network techniques available to the iOS developer.

Chapter 6: Securing Network Traffic — Here you examine securing network traffic beyond basic SSL communications, including code examples of client and server certificate validation.

Chapter 7: Optimizing Request Performance — This chapter looks at ways to improve the performance of network communications.

Chapter 8: Low Level Networking — Chapter 8 explores using low-level networking APIs to perform socket or datagram communications from an iOS app.

Chapter 9: Testing and Manipulating Network Traffic — This chapter appraises methods to intercept and modify communications between devices and servers for the purposes of app diagnosis and quality assurance.

Chapter 10: Using Push Notifications — This chapter describes how to use push notifications to communicate asynchronously from the server to the app.

Part IV: Networking App to App

The fourth section contains three chapters describing how to communicate between apps on the same device or other devices.

Chapter 11: Inter-App Communication — This chapter enumerates and describes ways to communicate between apps on the same device.

Chapter 12: Device-to-Device Communication with Game Kit — Here you look at using Game Kit for communicating between devices for nongaming purposes which, for once, currently has more features than its .NET cousin.

Chapter 13: Ad-Hoc Networking with Bonjour — The final chapter examines Bonjour as a means to communicate between apps on multiple devices.

WHAT YOU NEED TO USE THIS BOOK

To get the most out of the book, you should have a basic understanding of iOS programming tasks such as elementary XCode use and how to deploy an app to a device. You need the following software or hardware to run the example apps:

- Apple Mac computer with OS X Lion (10.7) or higher
- XCode 4.3.2 or higher.
- An iOS device, iPhone 3GS or higher, iPad, or iPod Touch with iOS 5.0 or higher
- An Apple Developer account, available at (`https://developer.apple.com/programs/register/`)

CONVENTIONS

To help you get the most from the text and keep track of what's happening, a number of conventions appear throughout the book.

> **WARNING** *Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.*

> **NOTE** *Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.*

As for styles in the text:

- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code in two different ways:

```
We use a monofont type with no highlighting for most code examples.

We use bold to emphasize code that's particularly important in the present context.
```

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All the source code used in this book is available for download at `http://www.wrox.com`. When at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

> **NOTE** *Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-1-118-36240-2.*

After you download the code, just decompress it with your favorite compression tool. Alternatively, you can go to the main Wrox code download page at `http://www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

The code listings and snippets provided in the text of this book comprise only a part of the code required for a functional iOS app. The downloadable code examples are complete XCode projects that contain all of the code required to build and deploy the samples to an iOS device. Therefore, in addition to the code listings found in the text of the book, you will find other code files and resource files that are required to build and deploy the sample apps on the companion website for this book.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be grateful for your feedback. By sending in errata you may save another reader hours of frustration, and at the same time you can help us provide even higher quality information.

To find the errata page for this book, go to `http://www.wrox.com` and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata submitted for this book and posted by Wrox editors.

> **NOTE** *A complete book list including links to each book's errata is also available at* `www.wrox.com/misc-pages/booklist.shtml`.

If you don't spot "your" error on the Book Errata page, go to `www.wrox.com/contact/techsupport.shtml` and complete the form to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at `p2p.wrox.com`. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At `http://p2p.wrox.com` you can find a number of different forums to help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to `p2p.wrox.com` and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you want to provide, and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

> **NOTE** *You can read messages in the forums without joining P2P, but to post your own messages, you must join.*

After you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to This Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

PART I

Understanding iOS and Enterprise Networking

1

Introducing iOS Networking Capabilities

WHAT'S IN THIS CHAPTER?

- Understanding the iOS networking frameworks
- Key networking APIs available to developers
- Using your application's run Loop effectively

Great iOS applications require a simple and intuitive user interface. Likewise, great applications that communicate with a web service of any kind require a well-architected networking layer. An application's architecture must be designed with the flexibility to adapt to changing requirements and the capability to gracefully handle constantly changing network conditions, all while maintaining core design principles that enable proper maintainability and scalability.

When designing a mobile application's architecture you must have a firm grasp of key concepts, such as the run loop, the various networking APIs available, and how those APIs integrate with the run loop to create a responsive, networked application framework. This chapter provides a detailed discussion of run loops and how to use them effectively within an application. Also provided is an overview of the key APIs and when each should be used.

UNDERSTANDING THE NETWORKING FRAMEWORKS

Before you begin development of an iOS application that interacts with the network, you must understand how the networking layers are organized in Objective-C, as shown in Figure 1-1.

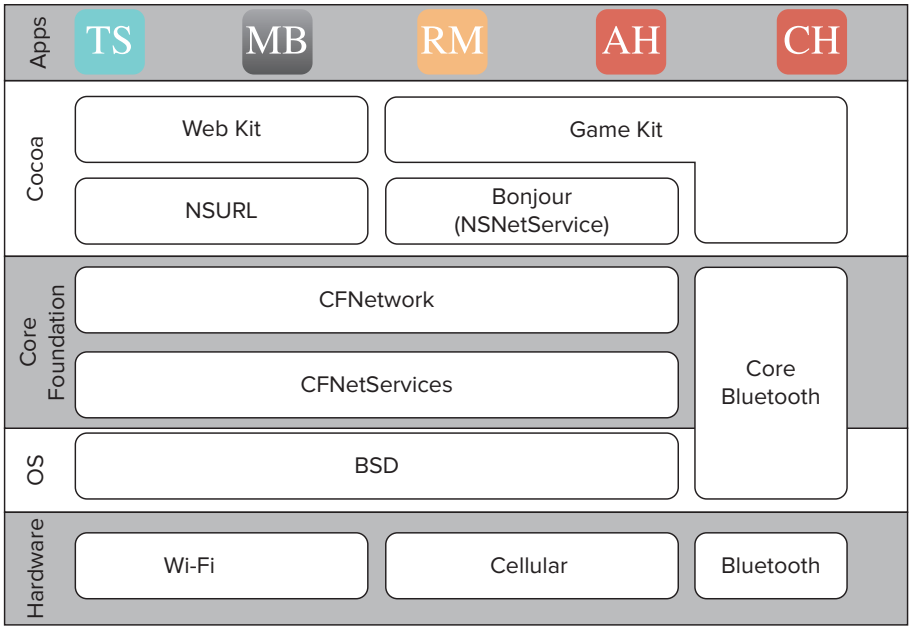


FIGURE 1-1

Each iOS application sits on top of a networking framework stack composed of four levels. At the top is the Cocoa level, which includes the Objective-C APIs for URL loading, Bonjour, and Game Kit. Below Cocoa sits Core Foundation, a set of C APIs that includes CFNetwork, the foundation of most application-level networking code. CFNetwork provides a simple networking interface that sits on top of `CFStream` and `CFSocket`. Those two classes are lightweight wrappers around BSD sockets, which form the lowest level and sit closest to the antenna hardware. BSD sockets are implemented strictly in C and provide developers absolute control over any communication to a remote device or server.

As you move down each level in the framework stack, you tend to gain tighter control but give up the ease of use and abstraction that the previous level provided. Although there are situations in which this may be warranted, Apple recommends that you stay at the CFNetwork layer and above. Raw sockets at the BSD level do not have access to the system wide VPN nor do they activate the Wi-Fi or cellular radios, something CFNetwork handles for you.

Before you design your applications' networking layer you must understand the various APIs available to you and how you can leverage them. The next section covers the key iOS networking frameworks and provides a brief introduction explaining how you can use them. Each API covered is discussed in detail in a future chapter.

iOS NETWORKING APIS

Each level of the framework stack has a set of key APIs that deliver a range of functionality and control to developers. Each level offers more abstraction than the level below it (refer to Figure 1-1). However, this abstraction comes at a cost of losing some control. This section provides an overview of key APIs in iOS and the considerations when using each of them.

NSURLConnection

`NSURLConnection` is a Cocoa level API that provides a simple method to load URL requests, which can interact with a web service, fetch an image or video, or simply retrieve a formatted HTML document. It is built on top of `NSStream` and was designed with optimized support for the four most common URI schemes: `file`, HTTP, HTTPS, and FTP. Although `NSURLConnection` restricts the protocols over which you can communicate, it abstracts much of the lower-level work required to read and write from buffers, includes built-in support for authentication challenges, and offers a robust caching engine.

The `NSURLConnection` interface is sparse, relying heavily on the `NSURLConnectionDelegate` protocol, which enables an application to intervene at many points in the connection life cycle. `NSURLConnection` requests are asynchronous by default; however, there is a convenience method to send synchronous requests. Synchronous requests do block the calling thread, so you must design applications accordingly. Chapter 3, "Making Requests" covers `NSURLConnection` in detail and provides a number of examples.

Game Kit

At its core, Game Kit provides another peer-to-peer networking option to iOS applications. In a traditional network configuration, Game Kit is built on top of Bonjour; however, Game Kit does not require a network infrastructure to function. It can create ad-hoc Bluetooth Personal Area Networks (PAN), which makes it a great candidate for networking in locations with little or no established infrastructure.

Game Kit requires only a session identifier, display name, and connection mode when setting up a network. It does not require configuring of a socket or any other low-level networking to communicate with connected peers. Game Kit communicates via the `GKSessionDelegate` protocol. Chapter 12, "Device-to-Device Communication with Game Kit" discusses integrating Game Kit into your applications.

Bonjour

Bonjour is Apple's implementation of zero configuration networking (zeroconf). Bonjour provides a mechanism to discover and connect with devices or services on the network, and alleviates the need to know a device's network address. Instead, Bonjour refers to services as a tuple of name, service type, and domain. Bonjour abstracts the low-level networking requirements for multicast DNS (mDNS) and DNS-based Service Discovery (DNS-SD).

At the Cocoa level, the `NSNetService` API provides an interface for publishing and resolving address information for a Bonjour service. You can use the `NSNetServiceBrowser` API to discover available services on the network. Publishing a Bonjour service, even with Cocoa level APIs, requires an understanding of Core Foundation to configure sockets for communication. Chapter 13, "Ad-Hoc Networking with Bonjour," includes an in-depth overview of zero configuration networking, Bonjour, and an example of how to implement a Bonjour-based service.

NSStream

`NSStream` is a Cocoa level API built on top of CFNetwork that serves as the foundation for `NSURLConnection` and is intended for lower-level networking tasks. Much like `NSURLConnection`, `NSStream` provides a mechanism to communicate with remote servers or local files. However, you can use `NSStream` to communicate over protocols such as telnet or SMTP that are not supported by `NSURLConnection`.

The additional control that `NSStream` provides does come at a cost. `NSStream` does not have built-in support for handling HTTP/S response status codes or authentication challenges. It transmits and receives data into C buffers, which may be unfamiliar to a strictly Objective-C developer. It also can't manage multiple outbound requests and may require subclassing to add that feature. `NSStream` is asynchronous and communicates updates via the `NSStreamDelegate`. Chapter 8, "Low-Level Networking," and Chapter 13, "Ad-Hoc Networking with Bonjour" cover different implementations of `NSStream`.

CFNetwork

The CFNetwork API is layered on top of the fundamental BSD sockets and is used in the implementations of `NSStream`, the URL loading system, Bonjour, and Game Kit APIs. It provides native support for advanced protocols such as HTTP and FTP. The key difference between CFNetwork and BSD sockets is run loop integration. If your application uses CFNetwork, input and output events are scheduled on the thread's run loop. If input and output events occur on a secondary thread, it is your responsibility to start the run loop in the appropriate mode. The "Run Loops" section later in this chapter provides additional details.

CFNetwork provides more configuration options than the URL loading system, which can be both beneficial and frustrating. These configuration options are visible when creating an HTTP request with CFNetwork. When creating the request you must manually add any HTTP headers and cookies that must be transmitted with the request. With `NSURLConnection`, though, standard headers and any cookies in the cookie jar are automatically added for you.

The CFNetwork infrastructure is built on top of the CFSocket and CFStream APIs from the Core Foundation layer. CFNetwork includes APIs for specific protocols such as CFFTP for communicating with FTP servers, CFHTTP for sending and receiving HTTP messages, and CFNetServices for publishing and browsing Bonjour services. Chapter 8 covers CFNetwork in greater detail, and Chapter 13 provides an overview of Bonjour.

BSD Sockets

BSD sockets form the basis for most Internet activity and are the lowest level in the networking framework hierarchy. BSD sockets are implemented in C but can be used within Objective-C code. Use of the BSD socket API is not recommended because it does not have any hooks into the operating system. For example, BSD sockets are not tunneled through the system wide VPN nor do any of the API calls automatically activate the Wi-Fi or cellular radios if they are powered down. Apple recommends that you work solely with at least CFNetwork or higher. Chapter 8 covers BSD sockets and CFNetwork in greater detail and provides examples of how they can be integrated into your application.

As you implement the various network APIs, you must understand how they integrate with your application. The next section discusses the concept of run loops, which monitor for network events (among other things) from the operating system and relay those events to your application.

RUN LOOPS

Run loops, represented by the class `NSRunLoop`, are a fundamental component of threads that enable the operating system to wake sleeping threads to manage incoming events. A run loop is a loop configured to schedule tasks and process incoming events for a period of time. Each thread in an iOS application can have at most one run loop. For the main thread the run loop is started for you and is accessible after the application delegate's `applicationDidFinishLaunchingWithOptions:` method is invoked.

Secondary threads, however, must run their run loop explicitly, if needed. Before starting a run loop in a secondary thread, you must add at least one input source or timer; otherwise, the run loop exits immediately. Run loops provide developers with the ability to interact with a thread, but are not always necessary. Threads spawned to process a large data set without any other interaction, for example, probably do not warrant starting the run loop. However, if the secondary thread interacts with the network, you need to start the run loop.

There are two source types from which run loops receive events: input sources and timers. Input sources, which are typically either port-based or custom, deliver events to the application asynchronously. The primary difference between the two types of sources is that the kernel signals port-based sources automatically, whereas custom sources must be signaled manually from a different thread. You can create a custom input source by implementing several callback functions associated with `CFRunLoopSourceRef`.

Timers generate time-based notifications that provide a mechanism for applications (threads specifically) to perform a specific task at a future time. *Timer events* are delivered synchronously and are associated with a specific mode, which is discussed later in this section. If that particular mode is not currently monitored, events will be ignored, and the thread will not be notified until the run loop is "run" in the corresponding mode.

You can configure timers to fire once or repeatedly. Rescheduling is based on the scheduled fire time, not the actual fire time. If a timer fires while the run loop is executing an application handler method, it waits until the next pass through the run loop to call the timer handler, typically set via `@selector()`. If firing the handler is delayed to the point in which the next invocation occurs, the timer fires only one event with the delayed event being suppressed.

Run loops can also have observers, which are not monitored and provide a way for objects to receive callbacks as certain activities in the run loop execution occur. These activities include when the run loop is entered or exited, as the run loop goes to sleep or wakes up, and before the run loop processes an input source or timer. They are documented in the `CFRunLoopActivity` enumeration. Observers can be configured to fire once, which removes the observer after it has been fired, or repeatedly. To add a run loop observer, use the Core Foundation function `CFRunLoopObserverRef()`.

Run Loop Modes

Each pass through the run loop is run in a specific mode specified by you. *Run loop modes* are a convention used by the operating system to filter the sources that are monitored and allowed to deliver events, such as calling a delegate method. Modes include the input sources and timers that should be monitored as well as any observers that should be notified of run loop events.

There are two predefined run loop modes in iOS. `NSDefaultRunLoopMode` (`kCFRunLoopDefaultMode` in Core Foundation) is the system default and should typically be used when starting run loops and configuring input sources. `NSRunLoopCommonModes` (`kCFRunLoopCommonModes` in Core Foundation) is a collection of modes that is configurable. Assigning `NSRunLoopCommonModes` to an input source by calling a method such as `scheduleInRunLoop:forMode:` on an input source instance associates it with all modes currently in the group.

> **NOTE** *OSX includes three additional predefined run loop modes that you may see referenced in different documentation.* `NSConnectionReplyMode`*,* `NSModalPanelRunLoopMode`*, and* `NSEventTrackingRunLoopMode` *provide additional filtering options but are not available on iOS.*

Although `NSRunLoopCommonModes` is configurable, it is a low-level process that requires calling the Core Foundation function `CFRunLoopAddCommonMode()`. This automatically registers input sources, timers, and observers with the new mode instead of manually adding them to each new mode. You can define custom run loop modes by specifying a custom string such as `@"CustomRunLoopMode"`. For your custom run loop to be effective, you must add at least one input source, timer, or observer.

Although this provides an overview of run loops, Apple provides several in-depth resources on run loop management that you should review if you develop advanced, network-based, and multi-threaded applications. The developer documentation is available at `https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Multithreading/RunLoopManagement/RunLoopManagement.html`. Networking techniques that benefit from run loop integration are discussed in their respective chapters such as Chapter 8, “Low-Level Networking” and Chapter 13, “Ad-Hoc Networking with Bonjour.”

SUMMARY

Understanding the iOS networking stack and how applications interact with the run loop is an important tool in the iOS developer’s belt. A well-architected networking layer provides incredible flexibility to an application. Likewise, a poorly designed networking layer can be detrimental to its success and ability to scale.

The tools presented in this chapter provide an overview of the various networking APIs and how they compare. How they are applied, although covered briefly here, is discussed in detail in the upcoming chapters.

2

Designing Your Service Architecture

WHAT'S IN THIS CHAPTER?

- Implementing a remote façade
- Discovering endpoints with service locators
- Supporting older apps with service versioning

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. You can find the code for this chapter in the Chapter 2 download in one example project and one set of web services:

- `Facade Tester.zip`
- `Facade PHP.zip`

Web services are the lifeblood of a networked iOS app, and the flexibility and robustness of their design has an enormous impact on its user experience. Well-designed service APIs can adapt to changing back-end data sources and still present an unchanging façade to the apps that depend on them. Service locators enable an app to dynamically discover new service endpoints and use them without needing to recompile or resubmit an app to the App Store. When it is necessary to resubmit an app, you need to support older versions of the app during the transition and upgrade process, which may realistically be the entire lifetime of the app. A service API that supports versioning is invaluable when supporting older apps that are still

used every day without compromising your ability to offer new features to new versions. This chapter covers example implementations of these invaluable design elements in the context of real world business scenarios.

REMOTE FAÇADE PATTERN

When designing service architecture for your app, a remote façade simplifies app integration and allows multiple clients to share the same business logic. The façade pattern is used to abstract the complexities of an underlying system away from the clients using that system. For example, the postal system includes thousands of mail carriers, trucks, aircrafts, distribution centers, and post offices; however, most tasks that its customers need hide all that complexity and simply consist of mailing a letter or receiving a package. Customers don't need to know how a letter gets from New York City to San Francisco, they just need to pay for postage and wait for it to arrive. Similarly, an application API might abstract multiple database queries or back-end system requests into a single externally accessible method that returns the results of the operation. As long as the façade's external API contract remains constant, the underlying systems can be changed, upgraded, or removed entirely without impacting any clients using the façade.

A remote façade takes this pattern and employs it in the web service tier for an application. It defines an unchanging service contract that an app can use to create, read, update, or delete data stored externally to the app. The API is commonly used to interact with existing systems already in use at the business and provides a mobile version of the same functionality. Figure 2-1 shows how an application would query various endpoints directly, and Figure 2-2 shows how the topology would change when the façade interacts with the back-end services on behalf of the application. If care and forethought are put into the initial service contract, the same API can adapt to most changes in the back-end systems, which enables an app to remain functional without needing constant updates to match the service infrastructure.

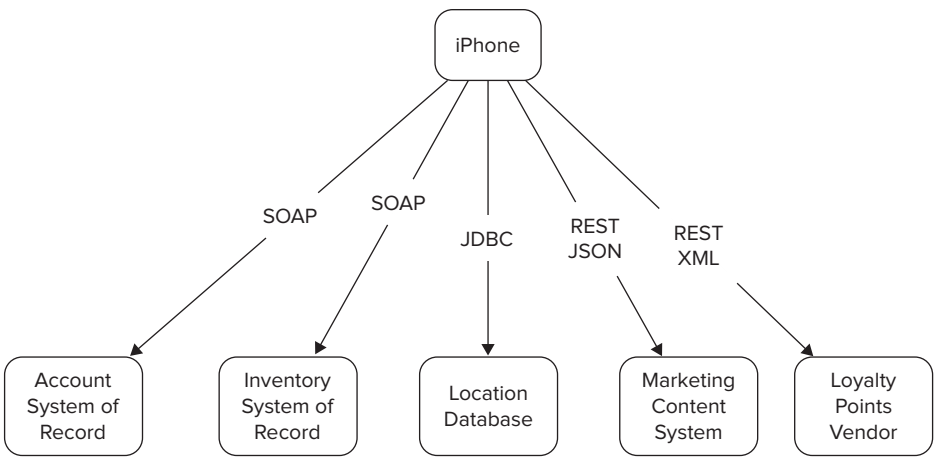


FIGURE 2-1

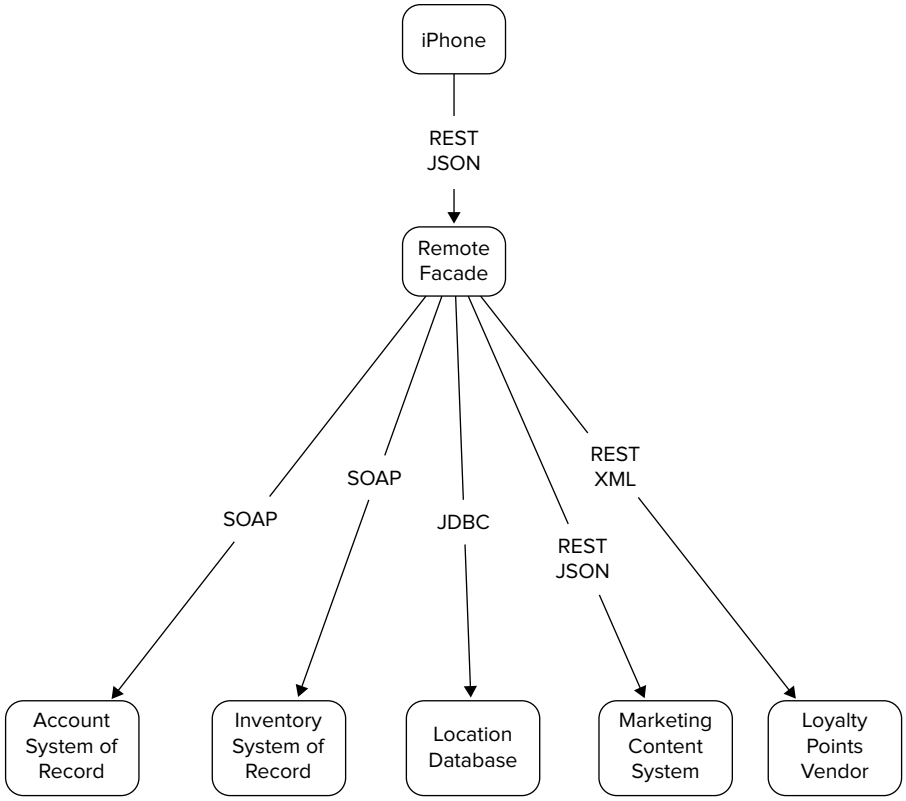


FIGURE 2-2

Imagine, for example, a bank merges with a competitor and wants to move its existing accounts to the competitor's account storage system. If the service API is written with abstract banking functions, it can work with any back-end database that provides the same data, even if it is stored in a new format. The remote façade can switch to the new source, transform any data that doesn't already match the API contract, and then return it to a mobile banking app without the user knowing that something changed. This development style is called *contract programming* and ensures that both sides of a networking session abide by a previously agreed upon input and output contract. As long as the contract is still valid, either end can be rewritten, ported to another language, or upgraded at will without any negative impact on the other party.

Maintainability, reliability, and complexity of the application side of the contract are also greatly enhanced by the façade pattern. With fewer points of networked interaction in the app, changes needed to support future façade versions are fewer and relatively self-contained. Reliability improves because the façade commonly has only one protocol and one message format, which reduces the number of third-party libraries or separate parsers needed for other formats. Both of these changes lower the complexity of the app and lead to development savings because fewer unit tests are needed to cover all functionality. On the server side, only one set of endpoints needs to be secured and exposed to the Internet instead of many disparate systems.

A remote façade also enables developers to push some business logic out of the app and into the service tier. Certain functions that change frequently or can't be predicted ahead of time can be computed in the service tier and send only the final value to the client. That way if this logic needs to be tweaked or adjusted for a new business rule, it does not require an app update to take effect. In the merging banks example, this tweak might be a new password security requirement adopted from the new institution. If the app merely takes the user's candidate password and asks the façade if it is valid, that logic can be changed at any time. A similar pattern to verify e-mail addresses can easily adapt to the upcoming switch to custom top-level domain (TLD) names; however, if the list of valid TLDs were hardcoded in the app, it would potentially reject valid e-mail addresses until an app update could be released. The remote façade grants an enterprise maximum flexibility over a networked app's post-launch behavior in the face of changing business processes.

The same characteristics also apply on the input side of the API. The façade can translate requests into formats needed by back-end systems; for example, it can convert an incoming JSON to a SOAP request. It can also enforce security constraints for other systems that can't be publicly exposed to the Internet, track and verify API keys before forward requests, or rate limit requests to certain back-end systems.

Example Façade Services

The example Façade Tester application uses two web services to populate its views: a stock quote service and a weather service. Both can fetch their respective data from two separate sources and convert each set of data into one common output format. This mimics a façade service that must accommodate a switch between two back-end systems while the app continues to work. Both of these examples refer to the version 1 services; the version 2 services are used in the "Service Versioning" section.

The stock quote service loads data as comma-separated values (CSV) or as an XML document, as shown in Listing 2-1.

LISTING 2-1: Generating Common Output from Two Stock Quote Sources (stockQuote_v1.php)

```
<?php

$useYahooResults = true;
$ticker = "AAPL";

if ($useYahooResults) {
    $rawData = rtrim(file_get_contents("http://finance.yahoo.com/d/quotes.csv?s=".
        $ticker."&f=snl1p2o"),"\r\n");

    $data = explode(",",$rawData);

    $symbol = trim($data[0],'"');
    $name = trim($data[1],'"');
    $currentPrice = trim($data[2],'"');

} else {
    $rawXML = file_get_contents("http://www.webservicex.net/stockquote.asmx/
        GetQuote?symbol=".$ticker);
```

```
    $wrapperData = simplexml_load_string($rawXML);

    $xmlData = simplexml_load_string($wrapperData);

    $symbol = (string)$xmlData->Stock->Symbol;
    $name = (string)$xmlData->Stock->Name;
    $currentPrice = (string)$xmlData->Stock->Last;
}

$response = array("symbol" => $symbol,
                  "name" => $name,
                  "currentPrice" => $currentPrice);

// output final results:
print json_encode($response);

?>
```

The following comma-separated string has key stock values: the company name, most recent price, opening price, and percentage change since opening.

```
{ticker symbol},{name},{last trade price},{percentage change},{opening price}

example:
"AAPL","Apple Inc.",530.12,"-2.92%",545.31
```

The following XML document contains the same data in a more structured format. This document contains all the information included in the CSV string plus some extra data that this service ignores.

```
<StockQuotes>
    <Stock>
        <Symbol>AAPL</Symbol>
        <Last>530.12</Last>
        <Date>5/17/2012</Date>
        <Time>4:00pm</Time>
        <Change>-15.955</Change>
        <Open>545.31</Open>
        <High>547.50</High>
        <Low>530.12</Low>
        <Volume>25614960</Volume>
        <MktCap>495.7B</MktCap>
        <PreviousClose>546.075</PreviousClose>
        <PercentageChange>-2.92%</PercentageChange>
        <AnnRange>310.50 - 644.00</AnnRange>
        <Earns>41.042</Earns>
        <P-E>13.31</P-E>
        <Name>Apple Inc.</Name>
    </Stock>
</StockQuotes>
```

When the variable `$useYahooResults` is `true`, the CSV string is loaded and when it is `false`, the XML is loaded. Regardless of the input source, the façade returns its data in a common JSON format like so:

```
{"symbol":"AAPL","name":"Apple Inc.","currentPrice":"-2.92%"}
```

Any data source used by the façade must provide data for at least the minimum required fields to abide by the contract it has made with clients of the API.

The example façade also implements a web service that gives the current weather for Richmond, VA, from one of two sources. Both sources provide weather conditions as JSON, but each specific response format varies greatly. This service is similar to a situation in which you might upgrade a back-end system to a new release that has the same basic data but organized differently. Listing 2-2 shows the weather service, which follows the same basic structure as the stock service.

LISTING 2-2: Generating Common Output from Two Weather Services (weather_v1.php)

```
<?php

$useYahooResults = true;

if ($useYahooResults) {
    $rawJSON = file_get_contents("http://query.yahooapis.com/v1/
                    public/yql?q=select%20item%20from%20weather.forecast
                    %20where%20location%3D%2248907%22&format=json");
    $rawData = json_decode($rawJSON);

    $currentTemperature = $rawData->query->results->channel->item->condition->temp;
    $currentConditions = $rawData->query->results->channel->item->condition->text;

} else {
    $rawJSON = file_get_contents("http://weather.yahooapis.com/forecastjson?
        w=12518996");
    $rawData = json_decode($rawJSON);

    $currentTemperature = (string)$rawData->condition->temperature;
    $currentConditions = $rawData->condition->text;
}

$response = array("city" => "Richmond",
                  "state" => "Virginia",
                  "currentTemperature" => $currentTemperature);

/*
 * output final results:
 *
 * {"city":"Richmond","state":"Virginia","currentTemperature":"63"}
 */
print json_encode($response);

?>
```

A consuming client can now use each published service, and the data sources can be switched dynamically without needing to change how it processes the stock or weather data.

Example Façade Clients

The Façade Tester application demonstrates how to use the output formats and displays the results in a table view. Listing 2-3 shows how to load the façade weather service in a background thread using Grand Central Dispatch. Listing 2-4 shows the equivalent code to parse the stock quote service. The JSON results are parsed using iOS 5's `NSJSONSerialization` and then assigned to local variables used by the table view. It is a testament to the ease of the façade pattern that the critical pieces of the iOS integration code are only a couple of lines long. For more information on loading data over the network see Chapter 3, "Making Requests," and for more information on JSON parsing see Chapter 4, "Generating and Digesting Request Payloads."

LISTING 2-3: Loading and Parsing the Weather Service (FTWeatherViewController.m)

```
NSString *v1_city;
NSString *v1_state;
NSString *v1_temperature;

- (void)loadVersion1Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
             [[UIApplication sharedApplication] delegate];

        if (appDelegate.urlForWeatherVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
                            dataWithContentsOfURL:appDelegate.urlForWeatherVersion1
                            options:NSDataReadingUncached
                            error:&error];

            if (error == nil) {
                NSDictionary *weatherDictionary = [NSJSONSerialization
                    JSONObjectWithData:data
                               options:NSJSONReadingMutableLeaves
                                 error:&error];

                if (error == nil) {
                    v1_city = [weatherDictionary objectForKey:@"city"];
                    v1_state = [weatherDictionary objectForKey:@"state"];
                    v1_temperature = [weatherDictionary objectForKey:
                        @"currentTemperature"];

                    // update the table on the UI thread
                    dispatch_async(dispatch_get_main_queue(), ^{
                         [self.tableView reloadData];
                    });

                } else {
                    NSLog(@"Unable to parse weather because of error: %@", error);
                     [self showParseError];
```

continues

LISTING 2-3 *(continued)*

```
                }

            } else {
                [self showLoadError];
            }

        } else {
            [self showLoadError];
        }
    });
}
```

LISTING 2-4: Loading and Parsing the Stock Quote Service (FTStockViewController.m)

```
NSString *v1_symbol;
NSString *v1_name;
NSNumber *v1_currentPrice;

- (void)loadVersion1Stock {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
            [[UIApplication sharedApplication] delegate];

        if (appDelegate.urlForStockVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSData dataWithContentsOfURL:
appDelegate.urlForStockVersion1
                                                options:
NSDataReadingUncached
                                                  error:&error];

            if (error == nil) {
                NSDictionary *stockDictionary = [NSJSONSerialization
                    JSONObjectWithData:data
                               options:NSJSONReadingMutableLeaves
                                 error:&error];

                if (error == nil) {
                    v1_symbol = [stockDictionary objectForKey:@"symbol"];
                    v1_name = [stockDictionary objectForKey:@"name"];
                    v1_currentPrice = [NSNumber numberWithFloat:
                        [[stockDictionary objectForKey:@"currentPrice"]
                        floatValue]];

                    // update the table on the UI thread
                    dispatch_async(dispatch_get_main_queue(), ^{
                        [self.tableView reloadData];
                    });

                } else {
```

```
                    NSLog(@"Unable to parse stock quote because of error:
                        %@", error);
                    [self showParseError];
                }

            } else {
                [self showLoadError];
            }

        } else {
            [self showLoadError];
        }
    });
}
```

SERVICE VERSIONING

Mobile applications are frequently updated to fix bugs and add new features, but it is often overlooked that web services must be maintained and upgraded as well. Service versioning is a technique to update an API's contract with its clients while still preserving the previous versions for existing app versions to use. Apps distributed through the App Store cannot force users to upgrade to the newest version, which means any existing web services need to be functional during the transition. Depending on the upgrade behavior of your user base, it may not ever be feasible to decommission your existing services without cutting off users of older versions. One option is to include logic that checks for a minimum supported application version and displays an upgrade message until the user consents. However, these nagging messages on previously working versions may upset some users, who may then quickly overwhelm your support lines and App Store reviews with negative comments. Because of this potential downside, proper service versioning is really the best solution.

API versioning is not limited to just adjusting for new app updates; it can also be used to deliver different or expanded data to various device types. For example, a reporting application might use a set of data when displayed on an iPhone and use a more complete set of data when shown on an iPad where it has more screen real estate. If that extra data has significant back-end or network overhead, you want to be sure you don't waste those resources on any iPhone-initiated service request that won't use it anyway.

A versioning system can be structured in two major ways: an active system where a remote façade receives the client's current version and chooses the correct endpoint, or a passive system where versioned service endpoints are hardcoded into each new release of the client. It is up to each specific enterprise to determine the best way to provide a version number as input, but typically it is included in the structure of a REST endpoint's URL or passed as a query parameter. The following code snippet demonstrates both version input options.

```
// a version given in the URL structure
http://example.com/api/1.0/stockquote/AAPL

// a version given as a query parameter
http://example.com/api/stockQuote.php?ticker=AAPL&apiVersion=1.0
```

A passive system is the simplest way to implement service versioning. It doesn't require the cost or capacity planning of an additional server and takes more organizational effort than technical effort. To implement in this manner, simply hardcode a version number into the endpoint URLs you already define within the client application. Because these URLs are functionally immutable after an application is released, you can ensure that an app coded to use that version always uses that version. When a new client version needs to change the service contract, simply increment the hardcoded API version number and create the new web service.

An active system encompasses all the benefits of the passive system; however, like all façade interactions, it also has the capability to change its behavior in the future. If two different versions of the same web service have the same input and output contract but they perform certain calculations differently, older clients compatible with either version can be switched dynamically in the future. For example, if an online retailer currently doesn't charge sales tax in a state, it can send its clients to version 1.0 of a price check service. However, if in the future it needs to begin charging sales tax, it can simply create a version 2.0 service that returns the normal price plus tax. Assuming the final amount is returned as a number in both cases, the service contract won't be broken. To implement an active versioning system, the façade must group all possible versions of client apps into compatibility buckets and assign each bucket the correct API version to use. To facilitate future development, choose a sensible default, typically the most recent API version, for client versions higher than the maximum known to the façade.

Example Versioned Services

Both the example web services have two versions that mimic a service contract that expands as business requirements change. Some output field types have been modified and other fields have been added. These examples use the passive versioning system that merely changes the URL to specify a new version. Recall that `weather_v1.php`, the version 1.0 of the weather service, had the following output format:

```
{"city":"Richmond","state":"Virginia","currentTemperature":"63"}
```

The `currentTemperature` is represented as a string, which complicates any integer logic that a client might want to do, for example, setting thresholds for cold, mild, or hot weather used to classify the current temperature. Version 2.0 of the service fixes this oversight and returns the value as a numeric type. It also adds an additional field for `currentConditions`, a text description of the current weather. The output in `weather_v2.php` has the following format:

```
{"city":"Richmond","state":"Virginia","currentTemperature":63,"currentConditions":
"Mostly Cloudy"}
```

The stock quote service had similar changes made from version 1.0 to version 2.0. The first version had basic output in `stockQuote_v1.php`:

```
{"symbol":"AAPL","name":"Apple Inc.","currentPrice":"530.12"}
```

Notice that `currentPrice` is a string in version 1.0 but is represented as a decimal number in `stockQuote_v2.php` to make it easier to format in the client:

```
{"symbol":"AAPL","name":"Apple Inc.","openingPrice":545.31,"currentPrice":530.12,
"percentageChange":"-2.92%"}
```

Two new fields have also been added: `openingPrice` and `percentageChange`.

Example Client Using Versioned Services

The weather view controller in Façade Tester can display the output of both versions of the API. Both `loadVersion1Weather` and `loadVersion2Weather` check the application delegate for the URL of the API endpoint, as highlighted in Listing 2-5. Because this example uses passive versioning, it might seem more natural to hardcode the URL directly here; however, defining it in the application delegate gives you flexibility to implement a service locator, as shown in the next section.

LISTING 2-5: Fetching API Endpoints from the Application Delegate (FTWeatherViewController.m)

```
- (void)loadVersion1Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
             [[UIApplication sharedApplication] delegate];

        if (appDelegate.urlForWeatherVersion1 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
dataWithContentsOfURL:appDelegate.urlForWeatherVersion1
                           options:NSDataReadingUncached
                           error:&error];

        // remaining code removed for brevity
}

- (void)loadVersion2Weather {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        FTAppDelegate *appDelegate = (FTAppDelegate*)
             [[UIApplication sharedApplication] delegate];

        if (appDelegate.urlForWeatherVersion2 != nil) {
            NSError *error = nil;
            NSData *data = [NSData
dataWithContentsOfURL:appDelegate.urlForWeatherVersion2
                           options:NSDataReadingUncached
                           error:&error];

        // remaining code removed for brevity
}
```

After the application loads the correct JSON data, it simply parses it according to the service contract for that version. Version 1.0 of the weather service loads the city, state, and current temperature like so:

```
v1_city = [weatherDictionary objectForKey:@"city"];
v1_state = [weatherDictionary objectForKey:@"state"];
v1_temperature = [weatherDictionary objectForKey:@"currentTemperature"];
```

Version 2.0 is similar but parses the current temperature as a number and also looks for current conditions as shown in the following:

```
v2_city = [weatherDictionary objectForKey:@"city"];
v2_state = [weatherDictionary objectForKey:@"state"];
v2_temperature = [[weatherDictionary objectForKey:@"currentTemperature"] intValue];
v2_conditions = [weatherDictionary objectForKey:@"currentConditions"];
```

When setting `v2_temperature`, it converts from an `NSNumber`, the numeric type used by `NSJSONSerialization`, to an integer type used by the view controller.

SERVICE LOCATORS

A service locator is a tool that helps applications dynamically discover API endpoints from a remote source. This alleviates the problem in which an application has hardcoded an endpoint that is invalid or no longer exists. Using a service locator also allows app developers to repoint previously released applications to new services whenever those services become available. These new services don't necessarily need to change the API contract with any clients; for example, if endpoints are moved to a different server or subdomain, behind a load balancer, or to an SSL-secured HTTPS endpoint. You can even create new service locator files for each development environment to easily switch between development, QA, or production resources with a single change.

At its core, a service locator is simply a file that contains API endpoints and some brief metadata about them. The application uses this metadata to determine which endpoints are appropriate for it to use. Examples might include the API version, input or output format, device type, or security level. It also must include the URL of the endpoint and a key that the client application can use to match an endpoint to its function. Because this file is static and changes infrequently, it is easily deployed to a web server or content delivery network (CDN). It is imperative that the source of the service locator be highly reliable because it is the single point of failure for the application. Although this may seem like a liability, a single point of failure is still preferable to the many points of failure that would exist if the application directly queried each separate back-end service. Where possible the service locator should be load balanced to avoid overwhelming a single host with requests from your entire user base. Because CDNs are designed for high-reliability of static files and typically can handle much higher sustained bandwidth than an everyday web server, it is recommended to use one to serve the service locator file whenever possible.

Because most web services output their results as JSON, it makes sense to use JSON to represent the service locator as well. Listing 2-6 shows the service locator used by the Façade Tester to discover the weather and stock quote API endpoints. This structure combines all versions of the endpoints into one file; however, you can also create individual service locator files for each API version. The individual approach prevents an app version from mixing and matching different service versions, but that constraint may not be an impediment for some business cases.

LISTING 2-6: An Example Service Locator File (serviceLocator.json)

```
{
    "services": [
        {
            "name": "stockQuote",
            "url": "http://example.com/api/stockQuote_v1.php",
            "version": 1
        },
        {
            "name": "stockQuote",
            "url": "http://example.com/api/stockQuote_v2.php",
            "version": 2
        },
        {
            "name": "weather",
            "url": "http://example.com/api/weather_v1.php",
            "version": 1
        },
        {
            "name": "weather",
            "url": "http://example.com/api/weather_v2.php",
            "version": 2
        }
    ]
}
```

Any client implementing the service locator pattern commonly loads and parses the file as its first action. Because all network calls require an endpoint, which are solely found in this file, it must be parsed before any other networked action can happen. The locator file should also be updated when an application returns to the foreground to ensure its endpoint data is fresh. Apps can remain in a background state for an extended period of time, and it may have previously loaded a service locator file that is now stale. In certain cases the stale endpoints may have been decommissioned and consistently timeout, providing a poor user experience. Typically an application displays a splash screen while the service locator loads.

Listing 2-7 shows an application that loads the service locator when the app launches and when it returns to the foreground. It stores the URLs as properties in the application delegate; however, a more complex application would require a dedicated networking manager that would handle loading the service locator and would be used by other controllers to query for the endpoint for a particular networking call.

LISTING 2-7: Loading and Parsing a Service Locator File (FTAppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // some code removed for brevity
```

continues

LISTING 2-7 *(continued)*

```
    /*
     * load the service locator
     *
     * note: You should probably show a splash screen of some kind here
     *     that waits for the SL to fully load.  Currently a user could
     *     try to start a network request before it knows which URL to use.
     */
    [self loadServiceLocator];

    return YES;
}

- (void)applicationDidBecomeActive:(UIApplication *)application {
    // load the service locator
    [self loadServiceLocator];
}

- (void)loadServiceLocator {
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{



        NSError *error = nil;
        NSData *data = [NSData dataWithContentsOfURL:[NSURL URLWithString:
                            @"http://example.com/api/serviceLocator.json"]
                                                 options:NSDataReadingUncached
                                                   error:&error];

        if (error == nil) {
            NSDictionary *locatorDictionary = [NSJSONSerialization
                JSONObjectWithData:data
                           options:NSJSONReadingMutableLeaves
                             error:&error];

            if (error == nil) {
                self.urlForStockVersion1 = [self
                    findURLForServiceNamed:@"stockQuote"
                                   version:1
                              inDictionary:locatorDictionary];

                self.urlForStockVersion2 = [self
                    findURLForServiceNamed:@"stockQuote"
                                   version:2
                              inDictionary:locatorDictionary];

                self.urlForWeatherVersion1 = [self
                    findURLForServiceNamed:@"weather"
                                   version:1
                              inDictionary:locatorDictionary];

                self.urlForWeatherVersion2 = [self
```

```
                    findURLForServiceNamed:@"weather"
                                   version:2
                              inDictionary:locatorDictionary];

            } else {
                NSLog(@"Unable to parse service locator because of error:
                    %@", error);

                // inform the user on the UI thread
                dispatch_async(dispatch_get_main_queue(), ^{
                     [[[UIAlertView alloc] initWithTitle:@"Error"
                                                 message:@"Unable to parse
                                                         service locator."
                                                delegate:nil
                                       cancelButtonTitle:@"OK"
                                       otherButtonTitles:nil] show];
                });
            }

        } else {
            NSLog(@"Unable to load service locator because of error: %@", error);

            // inform the user on the UI thread
            dispatch_async(dispatch_get_main_queue(), ^{
                 [[[UIAlertView alloc] initWithTitle:@"Error"
                                             message:@"Unable to load service
                                                     locator.  Did you remember
                                                     to update the URL to your own
                                                     copy of it?"
                                            delegate:nil
                                   cancelButtonTitle:@"OK"
                                   otherButtonTitles:nil] show];
            });
        }
    });
}

- (NSURL*)findURLForServiceNamed:(NSString*)serviceName
                         version:(NSInteger)versionNumber
                    inDictionary:(NSDictionary*)locatorDictionary {

    NSArray *services = [locatorDictionary objectForKey:@"services"];

    for (NSDictionary *serviceInfo in services) {
        NSString *name = [serviceInfo objectForKey:@"name"];
        NSInteger version = [[serviceInfo objectForKey:@"version"] intValue];

        if ([name caseInsensitiveCompare:serviceName] == NSOrderedSame &&
            version == versionNumber) {

            return [NSURL URLWithString:[serviceInfo objectForKey:@"url"]];
        }
    }

    return nil;
}
```

SUMMARY

Optimally, a flexible service architecture must be planned and implemented before the first version of an application is released to achieve its maximum benefits. If one version goes out with hardcoded endpoints or business logic, you are effectively supporting that configuration indefinitely, even if your business greatly evolves from it. With the combination of a remote façade, API versioning, and a service locator, you have many options to tweak business logic and API settings for apps already in the wild. It becomes easier to support minor tweaks in production code and major new features in upcoming versions without breaking the previous application versions. The upfront development costs of the service infrastructure may seem unnecessary, but it can pay for itself many times over as the application grows and evolves.

PART II

HTTP Requests: The Workhorse of iOS Networking

3

Making Requests

WHAT'S IN THIS CHAPTER?

- Understanding the structure of HTTP requests
- Issuing HTTP requests from iOS applications
- Using advanced manipulation of HTTP requests

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code is in the Chapter 3 download and individually named according to the names throughout the chapter.

As you may have noticed from the content of previous chapters, the preferred communication approach in iOS is HTTP. The most convenient networking APIs provided in iOS are geared toward HTTP, the HTTP APIs are the most thoroughly documented, and the high level HTTP APIs are well integrated into the run loop-based architecture of an iOS application. It is no wonder that HTTP and HTTPS are the workhorse protocols of iOS network communications.

In this chapter you learn about the structure of HTTP requests and how these requests can be leveraged by your application. The chapter also provides concrete examples of three primary methods to generate HTTP requests and receive HTTP responses and recommendations on when to use or avoid each one. Finally, this chapter explores some more advanced ways to use the HTTP protocol to your advantage.

> **NOTE** *In this chapter and the remainder of the book, the term HTTP is used to denote both unsecure HTTP and secure HTTPS requests. Where there is a difference between these two protocols, it will be noted in the text.*

INTRODUCING HTTP

Sir Tim Berners-Lee produced the first version of the Hypertext Transfer Protocol (HTTP) as part of the WorldWideWeb project started in 1990. The protocol was defined with HTML as a way to deliver information to researchers at CERN in Geneva, Switzerland, using a standard user interface and markup language. The information presented to the user could also contain links to other related information that would be accessible by activating the link in the text. Prior to this project, information was stored in a variety of formats and accessible with different tools based on the format, which made finding, consuming, and relating historical research to your own research difficult. You can read the original proposal for the WorldWideWeb project at `http://www.w3.org/Proposal.html`.

> **NOTE** *An interesting side note in the invention of HTTP and HTML is that the first World Wide Web server and browser were written on a NeXTStep computer. In 1997, Apple acquired NeXT Computer and used NeXTStep as the basis for OS X. Apple's OS X became the foundation for iOS.*

There were three major innovations in Berners-Lee's original proposal: HTML, HTTP, and the URL. HTML defined a way to add styling to text, HTTP defined a way to convey data between server and client, and the URL defined a way to uniquely locate a resource across a network of machines.

As the use of web browsers and HTTP moved outside of the research lab into business and homes in the mid-1990s, it soon became a target for nefarious people and organizations. In response, engineers developed standards for securing HTTP traffic. Initially there were two competing standards, HTTPS and S-HTTP. HTTPS encrypts the entire HTTP message, whereas S-HTTP encrypts only the message body, leaving the headers in clear text. Both Microsoft and Netscape decided to standardize on HTTPS, which led to the abandonment of S-HTTP.

HTTP was initially designed for the communication of human readable hypertext content from a server to a client browser. The adoption of the World Wide Web caused HTTP to become the de facto standard for communicating human readable information around the Internet and into consumer's homes. With the rise of HTTP carrying HTML came the realization that HTTP could just as easily convey machine-to-machine information as well.

Because of the ubiquity of web browsers using HTML and HTTP, the use of HTTP to convey machine-readable data became the path of least resistance to move information between systems on

the Internet. If an application is on a computer connected to a network, you can almost guarantee that there is a way to communicate via HTTP to another host on the Internet. Corporate network proxies and firewalls can readily and securely convey HTTP requests between the secured corporate network and the Internet, performing filtering and security validations along the way. Although the Internet is designed to carry a multitude of different application level protocols, HTTP has become the protocol that requires the least configuration for the end user.

UNDERSTANDING HTTP REQUESTS AND RESPONSES

To effectively use HTTP for client-server communication, you should understand the underpinnings of the protocol. In this section you learn the key principles and structures of HTTP as it is used in modern applications.

An HTTP request follows the client-server paradigm for computer communications. Figure 3-1 illustrates the sequence of steps in a simple HTTP request. The client establishes a TCP connection to the server and then sends an HTTP request. The server subsequently responds to the request by sending a HTTP response over the same TCP connection. The client can then reuse the TCP connection for another request or close it. Early versions of the HTTP protocol allowed only one request per TCP connection. HTTP 1.1 permits the client to reuse the connection.

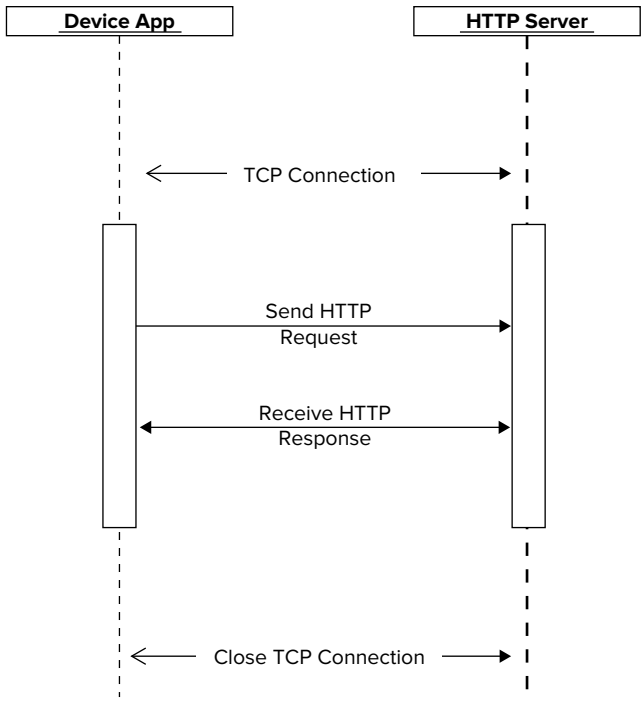


FIGURE 3-1

> **NOTE** *The definitive specification for HTTP is IETF RFC 2616. This RFC was adopted as a standard in 1999. You can find this RFC at* `http://www.ietf.org/rfc/rfc2616.txt`.

The most significant difference between HTTP and HTTPS is during the connection establishment phase of the conversation. After the TCP connection is made but before HTTP requests are transmitted, an SSL session must be established between the client and the server. SSL session establishment includes various stages: the client and server negotiating over which ciphers to use, exchanging public keys, validating the negotiation, and optionally validating identity. After the SSL session is established, all the data transmitted over the TCP connection will be encrypted.

URL Structure

From the perspective of an iOS developer, the other important invention of the WorldWideWeb project was the URL. The URL provides a globally unique location name for any resource or content on the Internet. As a rule, a single resource may be found with multiple URLs, but a single URL will not refer to different resources. There are exceptions to this rule, such as when the hostname refers to an ambiguous host. In the URL loading system of iOS, the `NSURL` object is used to manage URL objects.

A URL is typically composed of five components, as shown in Figure 3-2.

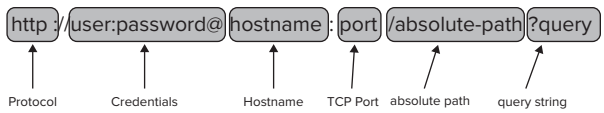


FIGURE 3-2

- **Protocol —** The *protocol* component specifies which application layer protocol to use to communicate to the server. If you've been around the web for a while, you may remember using `ftp` as a protocol in addition to `http`. The dominance of `http` has led to the near extinction of pre-HTTP protocol usage. Another commonly used protocol in iOS apps is the `file` protocol. `file` requests are used to retrieve resources in the local filesystem within the apps sandbox. If you create an `NSURL` object using a string without a protocol, it defaults to the `file` protocol.
- **Credentials —** Some HTTP servers support the delivery of user credentials in the URL to fulfill a BASIC authentication challenge. In Figure 3-2 the *credentials* component contains the username and password of the authenticating user. This format is not widely used and is considered less secure than other authentication methods.
- **Hostname —** The *hostname* portion of the URL specifies the TCP hostname or IP address of the host containing the wanted resource. If the protocol of the URL is `FILE`, then this component and the port component must be omitted. The exception to the preceding rule about a single URL referencing a unique resource is broken when relative or local hostnames are used. For example, if you use `localhost` as the hostname, the URL refers to the local machine; therefore, the same URL can refer to different resources on different machines.

- **Port —** The *port* portion of the URL specifies the TCP port to which the client should connect. If omitted the client uses the default port for the specified protocol: 80 for HTTP and 443 for HTTPS. It is best practice to use these port values for apps running on devices outside networks you control because some network proxies and firewalls will block nonstandard port numbers for security or privacy reasons.
- **Absolute-path —** The *absolute-path* component specifies the path to the network resource as if the HTTP server was drilling down into a directory tree. The absolute-path may include any number of path components each separated by the forward slash (`/`) character. An absolute-path may not contain a question mark, space, carriage-return, or line-feed characters. Many REST services use path components as a means to pass values to uniquely identify an entity stored in a database. For example, a path of `/customer/456/address/0` would specify the address at index 0 for the customer with an identifier of 456.
- **Query —** The last component of a URL is the *query* string. This value is separated from the absolute-path by a question mark (`?`). By convention multiple query parameters are each separated by an ampersand (`&`) character. The query string may not contain carriage return, space, or line-feed characters.

Because the contents of the absolute-path and query string are restricted, URLs are usually encoded using percent encoding. RFC 3986, `http://tools.ietf.org/html/rfc3986`, specifies the details of percent encoding of URLs. iOS provides a method on the `NSString` object to perform percent encoding of URLs. The following snippet shows how to percent encode an `NSString`.

```
NSString *urlString = @"http://myhost.com?query=This is a question";
NSString *encoded = [urlString
    stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];
```

The resulting value of encoded is `http://myhost.com?query=This%20is%20a%20question`. Each of the spaces was replaced by a %20 sequence. This encoding differs from URL encoding in that it does not encode the ampersand (&) characters, thereby leaving the URL parameter separation intact. URL encoding would encode the ampersands, question marks, and other punctuation. If your query strings contain these characters, you need to implement a more thorough encoding method.

Request Contents

An HTTP request consists of three parts: the request line, the request headers, and the request body. The request line and request headers are lines of text each separated by carriage-return/line-feed sequence (a byte with the value 13 or 0x0D and a byte with the value 10 or 0x0A). This use of textual values in the HTTP request makes them easy to construct, parse, and debug. An empty line, consisting of just a carriage-return/line-feed sequence or just a line-feed, separates the request headers from the request body.

The following code snippet contains an example HTTP request from a search request.

```
GET /search?source=ig&hl=en&rlz=&q=ios&btnG=Google+Search HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:11.0)…
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en,en-us;q=0.7,en-ca;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://www.google.com/ig?hl=en&source=webhp
Cookie: PREF=ID=fdf9979…
```

The request line is the first line of the data sent to the server. The request line contains three key pieces of information: the HTTP request method, the request URI, and the HTTP version.

The request method is a single word indicating the action being requested by the client. Because it is case-sensitive, the standard methods listed in Table 1-1 are all uppercase values. In the preceding code snippet the request method is a `GET` method.

TABLE 1-1: Common Request Methods and Their Uses

METHOD	STANDARD USES
`GET`	Retrieves a piece of content, or entity in HTTP terminology, from the server. `GET` requests usually don't contain a request body, but it is allowed. Some network caching appliances will cache only `GET` responses. `GET` requests usually do not cause data changes on the server.
`POST`	Updates an entity with data provided by the client. A `POST` request usually has information in the body of the request that is used by the application server. `POST` requests are considered to be non-idempotent, meaning that if more than one request is processed, the result is different than if only one request is processed.
`HEAD`	Retrieves metadata about a response without retrieving the entire contents of the response. This method is usually used to check a server for recent content changes without having to retrieve the full content.
`PUT`	Adds an entity with data provided by the client. A `PUT` request usually has information in the body of the request that is used by the application server to create the new entity. Usually, `PUT` requests are considered to be idempotent, meaning that the request can be repeatedly applied with the same results.
`DELETE`	Removes an entity based on contents of the URI or request body provided by the client. `DELETE` requests are most frequently used in REST service interfaces.

The second field in a request line is the URI. The URI uniquely identifies the target of the request. If the request uses the `GET` method, the URI unambiguously specifies the content to retrieve on the target host. The URI may also contain query parameters, which must not contain a space or carriage return character. In the prior code snippet the URI contains several query parameters, each

separated by an ampersand (&) character. Notice that the URI does not contain the protocol, host, or port that a user usually provides in the address field of a browser. The client uses the protocol portion of the URL to determine how to connect to the server. The hostname and port specified by the client is provided in the `Host` header of the request.

The last field of the request line specifies the version of the HTTP protocol being used. In the previous HTTP request code example that version value is 1.1, which means that the server should expect the client to apply headers and rules specific to version 1.1 of the HTTP protocol.

The lines immediately following the request line are the request headers, which provide additional metadata to the server. This metadata may describe the client, further describe the request, or request a certain type of response from the server. There may be one or more headers provided in each request. The `Host` header is the only required header in an HTTP 1.1 request. It provides the original hostname specified by the client and may include the port number provided in the URL of the original request. An HTTP server may serve content for multiple hostnames. The `Host` header helps the HTTP server know the host to which the original request was made.

> **NOTE** *The HTTP specification permits intermediaries between the HTTP client and server to add, remove, reorder, or modify HTTP headers. Therefore, a request from your app may arrive at the server with new headers, modified headers, or headers removed.*

Even though it uses the stateful TCP transport layer, HTTP is defined as a stateless protocol. This means that the HTTP server does not retain any information about a request to use in future requests. Cookies were developed as a way to allow some simple state information to be stored on the client and communicated to the server on subsequent requests.

Following the HTTP headers there is an optional request body. The request body is an arbitrary sequence of bytes separated from the headers by a single blank line. The request body must conform to the predetermined data encoding between the client and the server. For web browsers this is usually form-encoded data, but for mobile applications this encoding is usually XML or JSON data. Chapter 4, "Generating and Digesting Payloads," examines more closely request body and response body encoding.

In iOS the `NSURLRequest` object and its subclass `NSMutableURLRequest` provide the methods and attributes necessary to build almost any HTTP request. These objects will be discussed in the upcoming "High-Level iOS HTTP APIs" section.

Response Contents

After the HTTP server and any application servers supporting it finish processing the request, an HTTP response is returned to the client over the same TCP socket. An HTTP response is structured similarly to an HTTP request with the first line being the status line, followed by headers, and a response body. The following code shows a sample HTTP response.

```
HTTP/1.1 200 OK
Date: Tue, 27 Mar 2012 12:59:18 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Transfer-Encoding: chunked
Server: gws

<!doctype html><html itemscope="itemscope"
itemtype="http://schema.org/WebPage">
<head><meta itemprop="image" content="/images/google_favicon_128.png"/>
<title>ios - Google Search</title>
<script>window.google={kEI:"prlxT5qtNqe70AHh873aAQ",
getEI:function(a){var b;
while(a&&!(a.getAttribute&&(b=a.getAttribute("eid"
```

> **NOTE** *In the preceding code thousands of lines have been removed for the sake of clarity and brevity.*

The status line contains three fields, each separated by a space character. The first field is the HTTP version of the response. The next two fields provide status values indicating the outcome of the request. The first of the two fields is a three-digit integer value containing the result code of the request. The last field is a reason phrase that provides a short text description of the code. In most situations the numeric value fully describes the status. Chapter 5, "Handling Errors," examines error status, their causes, and appropriate ways to respond to errors.

The header lines immediately follow the status line; each separated by a carriage-return/line-feed sequence. Each header contains metadata about the response including information such as when the data was last modified, how long the client may cache the data, how the data is encoded, and state information to submit in subsequent requests.

The response body is separated from the headers by a single empty line. The response body may contain an arbitrary number of binary characters. The length of the response body is communicated to the client by either a `Content-Length` header in the request or by using chunked encoding. A chunk encoded response contains a `Transfer-Encoding` header with a value of `chunked`. A chunk encoded body contains one or more body segments. Each segment has a starting line that specifies the number of bytes in the chunk. The iOS URL loading system hides this complexity from your application.

In iOS's URL loading system, the `NSURLResponse` object and its subclass `NSHTTPURLResponse` encapsulate the data returned from a request. There are two objects in this hierarchy because the URL loading can fulfill requests for data based on non-HTTP URLs. For example, a request for a `file://` URL will not contain any headers.

HIGH-LEVEL IOS HTTP APIS

In this section you learn about the high-level APIs most commonly used for HTTP communications from an iOS application to an HTTP server. There are three primary methods to perform HTTP requests and receive responses using the URL loading system:

- **Synchronous —** The thread on which the initiating code runs blocks until the entire response is loaded and returned to the calling method. This technique is the simplest to implement but has the most limitations.
- **Queued asynchronous —** The initiating code creates a request and places it on a queue to be performed on a background thread. This method is slightly more difficult to implement and removes a significant limitation of the pure synchronous technique.
- **Asynchronous —** The initiating code starts a request that runs on the initiating thread but calls delegate methods as the requests proceeds. This technique is the most complicated to implement but provides the most flexibility in handling responses.

Each request has its own approach and best practices, but all three requests are composed of the same four objects. This section first covers these similarities and then dives into the single methods with examples and guidelines for each technique.

Objects Common to All Request Types

Like the menu of a cheap Mexican restaurant, all the URL loading request methods share a small set of common ingredients. Three types of requests use a combination of four types of objects: `NSURL`, `NSURLRequest`, `NSURLConnection`, and `NSURLResponse`.

NSURL

An `NSURL` object lets you easily manage URL values and gain access to the contents referenced by that URL. `NSURL` objects can refer to a file-based resource or a network-based resource with no difference between how the two resource types are used. The following code snippet shows the loading of data from a URL.

```
NSURL *url = [NSURL urlWithString:mysteryString];
NSData *data = [NSData dataWithContentsOfURL:url];
```

The value of `mysteryString` could reference a file or a network resource, and the code will behave the same. The significant difference is in the amount of time required to load the resource referenced by `mysteryString`. If the URL might reference a network resource, it would be wise to execute the code in a background thread so that the user interface does not pause while the data loads.

The most common way to create an `NSURL` object is to instantiate it using the class method `URLWithString:`. This method creates an `NSURL` object initialized with the contents of the provided `NSString`. The following snippet illustrates this.

```
NSURL *url = [NSURL URLWithString:@"http://www.wiley.com/path1"];
```

The NSURL object provides many accessor methods to read the component values of the URL. Each accessor provides read-only access to a portion of the URL. The scheme accessor returns an NSString containing the protocol scheme used by the URL. If a particular component is not specified in the target URL, then the value returns nil. Given the url object previously created, the code in the following snippet can log Port is nil.

```
if (url.port == nil) {
    NSLog(@"Port is nil");
} else {
    NSLog(@"Port is not nil");
}
```

If the URL contains a query string, the query accessor method contains the value of all the query parameters. The contents of the URL string must be percent encoded, per RFC 3986, prior to using it to create an NSURL object. For example, if the following snippet is executed, the value of the query parameter is q=iOS+Networking.

```
NSURL *url = [NSURL URLWithString:@"http://google.com?q=iOS+Networking"];
```

NSURL objects are immutable, meaning you cannot construct an empty NSURL object and populate its properties by calling *mutator methods*, sometimes called *setter methods*, on the object. The object is instantiated from either an NSString or another NSURL object. If the string used to instantiate the NSURL object is malformed, the creation method call returns nil. Your code should validate that the URL object was created correctly before using it for network requests.

NSURLRequest

After you create an NSURL object, your code can move to the next step required to perform an HTTP request: creating an NSURLRequest object. An NSURLRequest object contains the information necessary to load the contents of a URL independent of the protocol scheme specified in the URL. The URL loading system in iOS supports loading the contents of HTTP, HTTPS, FTP, and FILE URLs. The URL loading system provides a means to be extended to handle new protocols by creating subclasses of NSURLProtocol and providing feedback into the URL loading system.

The simplest way to create an NSURLRequest object is with the class method using a provided NSURL. The following snippet shows the creation of a request object using all default values.

```
NSURL *url = [NSURL URLWithString:
     @"https://gdata.youtube.com/feeds/api/standardfeeds/top_rated"];
if (url == nil) {
    NSLog(@"Invalid URL");
    return;
}
NSURLRequest *request = [NSURLRequest requestWithURL:url];
if (request == nil) {
    NSLog(@"Invalid Request");
    return;
}
```

Using the default values means that the request uses the request caching rules specified by the URL protocol, and that the request has the standard request timeout. See Chapter 7, "Optimizing Request Performance," for more information on the options for controlling request caching. If the

URL is an HTTP or HTTPS URL, then the request method will be a `GET` using the default headers provided by the operating system.

The following example shows creating an `NSURLRequest` object using custom values for the caching and timeout values. The code is instructing the URL loading system to ignore all caching and to generate an error if completing the requested connection takes more than 20 seconds.

```
NSURL *url = [NSURL URLWithString:
     @"https://gdata.youtube.com/feeds/api/standardfeeds/top_rated"];
if (url == nil) {
    NSLog(@"Invalid URL");
    return;
}
NSURLRequest *request = [NSURLRequest
                requestWithURL:url
                   cachePolicy:NSURLRequestReloadIgnoringCacheData
               timeoutInterval:20.0];
if (request == nil) {
    NSLog(@"Invalid Request");
    return;
}
```

The `NSURLRequest` object has several accessor methods to retrieve the properties of the request, none of which you can modify using the immutable `NSURLRequest` class. If you need to modify properties beyond the URL, caching policy, and timeout value, you need to use the `NSMutableURLRequest` class.

`NSMutableURLRequest` is a subclass of `NSURLRequest` and provides mutator methods to change the properties of the request. The following snippet shows the creation of a simple `POST` request with a small message body. It consists of the bytes of an `NSString` in UTF8 encoding. The URL loading system automatically populates the `Content-Length` header of the request.

```
NSURL *url = [NSURL urlWithString@"http://server.com/postme"];
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req setHTTPMethod:@"POST"];
[req setHTTPBody:[@"Post body"
                     dataUsingEncoding:NSUTF8StringEncoding]];
```

There are two ways to supply an HTTP body to an `NSURLRequest`: in memory like the preceding example or via an `NSInputStream`. Using an input stream allows your code to supply a request body without loading the entire body into memory. If you are sending something large like a photo or video, using an input stream is the best choice. The following snippet shows the creation of a `POST` method with an input stream. The `NSString srcFilePath` would be set beforehand to the path of the file residing either in the app bundle or sandbox.

```
NSMutableURLRequest *request =
       [NSMutableURLRequest requestWithURL:url];
NSInputStream *inStream =
      [NSInputStream
                 inputStreamWithFileAtPath:srcFilePath];
[request setHTTPBodyStream:inStream];
[request setHTTPMethod:@"POST"];
```

Because the `NSURLRequest` object contains properties for both HTTP and non-HTTP requests, code that accesses non-HTTP URLs will have the value of the HTTP specific properties set to `nil`.

> **NOTE** *In iOS 6 the* `NSURLRequest` *has a new property that indicates whether the request can be made over a cellular network. Using this property enables your app to leverage the Reachability framework discussed in Chapter 5, "Handling Errors," without explicitly adding this framework to your app.*

NSURLConnection

The `NSURLConnection` object is the hub of activity for the URL loading system but has a sparse interface that just provides methods to initialize, start, and cancel a connection.

Returning to the aforementioned primary methods used to perform HTTP requests and receive responses, an `NSURLConnection` class functions via these three different modes of operation: synchronous, asynchronous, and queued asynchronous. Synchronous mode is the easiest to use but has significant limitations that will make it inappropriate for more advanced interactions. Asynchronous mode provides greater flexibility but at the cost of complexity in your code. The third mode of operation, queued asynchronous, provides the background operation of asynchronous mode while keeping most of the simplicity of synchronous mode. The following sections in this chapter cover these three modes.

When operating in asynchronous mode, the `NSURLConnection` object calls a delegate object to guide the flow of the connection and to handle incoming data, handle authentication, and respond to errors. The "Asynchronous Requests" section in this chapter explores the `NSURLConnection` delegate in-depth.

NSURLResponse

An `NSURLResponse` object is returned upon completion of the URL load request. The contents of the response object can vary depending on the type of request that was made and the success of the request. The following list outlines the various objects returned from the request. Two other objects may also result from a URL load request: an `NSError` object and an `NSData` object. The `NSError` object will be populated if the request was malformed or if the client could not connect to the server. See Chapter 5 for in-depth information regarding error conditions. The resulting `NSData` object contains the response body, if one were returned. If an `NSError` object is produced, neither an `NSURLResponse` nor `NSData` object is produced.

- **MIMEType —** The mime type of the resulting data. This value may originate from the server, be overridden by the client framework if the client framework believes the server was mistaken, or be supplied by the client framework if no server value is supplied.
- **expectedContentLength —** This value may or may not be returned from the request, and the returned value may not equal the actual size of the returned content. If the size of the returned content is not known, this value will equal `NSURLResponseUnknownLength`.

- **suggestedFilename —** The name that either the server supplied as the filename of the content or a name derived from the URL and MIME type.
- **URL —** The URL of the returned content. This URL may differ from the URL provided in the request due to redirects or normalization.
- **textEncodingName —** The name of the text encoding used by the originating source of the data. This value may be `nil` if no text encoding was used in the response.

The URL loading system provides a subclass of `NSURLResponse` named `NSHTTPURLResponse` that contains properties specific to HTTP requests. This class is indispensable to determine the outcome of HTTP requests. Its additional parameters follow:

- **Response headers —** This property returns an `NSDictionary` of header values. The key of the dictionary is the header name, and the value for each key is the header value. The HTTP specification allows a request to have multiple headers of the same name. `NSHTTPURLResponse` handles this by returning a single `NSString` containing all the header values, each separated by a comma.
- **HTTP status code —** The integer status code from the status line of the response. The `NSHTTPURLResponse` class has a class method that returns a localized string description for any supplied status code.

The following sections provide annotated examples of how to use each of the three request methods of the URL loading system. Along with each example, best practices are provided for the use of each technique.

Synchronous Requests

Synchronous requests are the simplest type of requests to perform in iOS, but the cost of simplicity is reduced functionality and flexibility. When a synchronous request is made, the thread on which the request is made blocks until the request either completes or fails. Synchronous requests are commonly used to create `HTTP GET` requests to retrieve resources of a predictable size on a background thread. For example, retrieving an image to display in a table cell can easily be performed on a background thread using a synchronous request.

> **NOTE** *The examples in this section are pulled from the example program provided for this chapter. This example program is a simple RSS client program that retrieves an RSS feed from NASA and provides the capability to download referenced video files to the iOS device that are viewed at a later time.*

A synchronous request is used in this example to download the XML content of the RSS feed from the example program `VideoDownloader`. Figure 3-3 shows a screen shot of the example app. The segmented control at the top of the view provides a means to change between queued asynchronous

and synchronous requests to retrieve the feed XML. If you select the synchronous method and press the refresh button to the right, you can notice that the refresh button freezes in place and the app becomes nonresponsive while the request is made. On a Wi-Fi network this happens quickly, but if you do this on an EDGE network, you can notice significant delays.

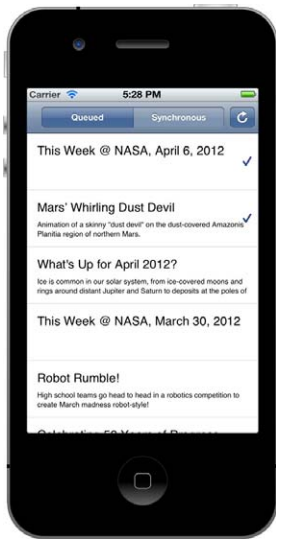


FIGURE 3-3

Synchronous requests present the simplest means to retrieve data at a URL, and there are many helper methods sprinkled throughout the iOS APIs that use synchronous requests under the covers. For example, the method on `NSString stringWithContentsOfURL:` creates an instance of `NSString` and retrieves those contents from an arbitrary server based on the contents of the URL. If the URL uses the `FILE` (for example, `file://foo.txt`) protocol, then the contents are retrieved from the local filesystem. If the URL uses the `HTTP` (for example, `http://www.wiley.com`) or `FTP` protocol, the contents are retrieved from a remote server. Therefore, you should be wary of these helper methods that retrieve content from a URL unless you are confident that the URL will be a `FILE` URL.

Listing 3-1 shows the `doSyncRequest` method within the example application that performs a synchronous request to load the XML feed.

LISTING 3-1: doSyncRequest Method of VideoDownloader/FeedLoader.m

```
- (NSArray *) doSyncRequest:(NSString *)urlString {
    // make the NSURL object from the string
    NSURL *url = [NSURL URLWithString:urlString];

    // Create the request object with a 30 second timeout and a
    // cache policy to always retrieve the
    // feed regardless of cachability.
    NSURLRequest *request =
       [NSURLRequest requestWithURL:url
          cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData
          timeoutInterval:30.0];

    // Send the request and wait for a response
    NSHTTPURLResponse   *response;
    NSError             *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:request
                                         returningResponse:&response
                                                     error:&error];
    // check for an error
    if (error != nil) {
        NSLog(@"Error on load = %@", [error localizedDescription]);
        return nil;
    }

    // check the HTTP status
    if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
        NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
```

```
        if (httpResponse.statusCode != 200) {
            return nil;
        }
    }

    // Parse the data returned into an NSDictionary
    NSDictionary *dictionary =
        [XMLReader dictionaryForXMLData:data
                                  error:&error];
    // Dump the dictionary to the log file
    NSLog(@"feed = %@", dictionary);

    NSArray *entries =[self getEntriesArray:dictionary];

    // return the list if items from the feed.
    return entries;

}
```

Listing 3-1 takes an `NSString` from the caller that contains the URL to load. After the URL is constructed, an `NSURLRequest` object is instantiated. In this example the code overrides the default caching policy and the default timeout. Note that the caching policy is set to never cache via the `NSURLRequestReloadIgnoringLocalAndRemoteCacheData` caching policy. This best displays the impact of synchronous requests on the UI because the UI thread is blocked. Normally, your code would not override all caching, but it is common to override the default timeout and specify a 30-second timeout for this request, as done in the Listing 3-1.

After the request is created, the code calls the class method `sendSynchronousRequest:returningResponse:error` on `NSURLConnection` to execute the request. This method takes the request as an input parameter and two pointers: one to an `NSURLResponse` object that will be populated with the server's response and one to an `NSError` object with error details if the request failed. The response pointer references an instance of `NSURLResponse`; however, it will be an instance of the `NSHTTPURLResponse` subclass for all HTTP requests. If the `NSError` is not `nil`, then the request failed at a low level; however, if it is `nil` then the request did not fail because of a networking error or invalid URL. It could still have failed semantically, for example if the server responded that it encountered an internal server error. The contents of the `NSError` object referenced by that pointer contain a detailed description of the error, and those values are covered in detail in Chapter 5.

The code in Listing 3-1 checks for the existence of an `NSError` object and the status code of the `NSHTTPURLResponse` object. If both of those indicate success then the method proceeds.

The `sendSynchronousRequest:returningResponse:error` method returns the HTTP response body as an `NSData` object. Because the feed is represented as XML, the `NSData` object of successful requests is parsed by an XML reader into an `NSDictionary`. The dictionary is then traversed, and the list of RSS items is returned to the caller.

Making synchronous calls is amazingly simple, requiring few lines of code to successfully retrieve data from a server, but that simplicity comes at a price of limited use cases and an increased risk of defects.

Best Practices for Synchronous Requests

- Only use them on background threads, never on the main thread unless you are completely sure that the request goes to a local file resource.

- Only use them when you know that the data returned will never exceed the memory available to the app. Remember that the entire body of the response is returned in-memory to your code. If the response is large, it may cause out-of-memory conditions in your app. Also remember that your code may duplicate the memory footprint of the returned data when it parses it into a usable format.
- Always validate the error and HTTP response status code returned from the call before processing the returned data.
- Don't use synchronous requests if the source URL may require authentication, as the synchronous framework does not support responding to authentication requests. The only exception is for `BASIC` authentication, for which credentials can be passed in the URL or request headers. Performing authentication this way increases the coupling between your app and the server, thereby increasing the fragility of the overall application. It can also pass the credentials in clear text unless the request uses the HTTPS protocol. See Chapter 6, "Securing Network Traffic," for information on responding to authentication requests.
- Don't use synchronous requests if you need to provide a progress indicator to the users because the request is atomic and provides no intermediate indications of progress.
- Don't use synchronous requests if you need to parse the response data incrementally via a stream parser.
- Don't use synchronous requests if you may need to cancel the request before it is complete.

Queued Asynchronous Requests

Queued asynchronous requests are similar to synchronous requests. The program supplies an `NSURLRequest` object, and the URL loading system attempts to load the request without any other interaction with the invoking code. The main difference between the two methods is that the URL loading system executes the queued asynchronous requests on a queue, potentially on a background thread. The concept of queued asynchronous requests was added to iOS in version 5.0.

iOS provides a facility called operation queues in the aptly named `NSOperationQueue`. These queues enable your program to describe an operation to perform and then submit the operation for queued execution in first-in-first-out order. The queuing framework provides for priority ordering and ordering based on operational dependencies, but the URL loading system does not use these facilities.

Before your code can make a queued asynchronous request, it must first create a queue on which the requests will be executed. The following code snippet shows how to create an operation queue.

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
```

An operation queue may perform multiple operations concurrently. By default, the number of concurrent operations is determined by iOS based on system conditions. You can override the default value by calling the `setMaxConcurrentOperationCount:` method of the created queue. When your app starts a queue is created automatically, which can be retrieved by calling the `mainQueue` class method of `NSOperationQueue`. Do not use this queue for executing network requests because it operates on the main thread, and long operations freeze the user interface. If you select the queued option from the test app's segmented control and tap the refresh button, you can notice that the refresh button returns to the default state immediately and the table view is cleared.

This happens because the request is queued and the main run loop continues processing rather than waiting for the request to complete.

Listing 3-2 shows the method used to create and process the results of a queued request. It starts identically to the synchronous request method by creating an `NSURL` and passing it to a new `NSURLRequest`. After the request is created, the code creates an `NSOperationQueue` named `queue` if one does not already exist. This variable is declared as a static variable in the implementation of the `FeedLoader` class. A typical application creates a queue in the app delegate at startup and then uses that queue throughout the application. Knowing that a queue exists, the code calls `NSURLConnection` to execute the request on the queue and calls a block after the operation either completes or fails. When the request is placed on the queue, the `doQueueRequest:delegate` method returns to the caller. Because the method returns before the URL load is complete, it requires a delegate class to call when the load is complete. Because of the asynchronous completion pattern used by this technique, your code needs to implement a delegate or notification pattern to propagate the received data back to the original requesting objects.

LISTING 3-2: doQueuedRequest Method of VideoDownloader/FeedLoader.m

```
- (void) doQueuedRequest:(NSString *)urlString  delegate:(id)delegate {
    // make the NSURL object
    NSURL *url = [NSURL URLWithString:urlString];

    // create the request object with a no cache policy and a 30 second timeout.
    NSURLRequest *request = [NSURLRequest requestWithURL:url
      cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData
      timeoutInterval:30.0];

    // If the queue doesn't exist, create one.
    if (queue == nil) {
        queue = [[NSOperationQueue alloc] init];
    }

    // send the request and specify the code to execute when the
    // request completes or fails.
    [NSURLConnection sendAsynchronousRequest:request
                                       queue:queue
                           completionHandler:^(NSURLResponse *response,
                                               NSData *data,
                                               NSError *error) {

            if (error != nil) {
               NSLog(@"Error on load = %@", [error localizedDescription]);
            } else {

                // check the HTTP status
                if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
                    NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
                    if (httpResponse.statusCode != 200) {
                        return;
                    }
```

continues

LISTING 3-2 *(continued)*

```
                }

                // parse the results and make a dictionary
                NSDictionary *dictionary =
                   [XMLReader dictionaryForXMLData:data
                                             error:&error];
                NSLog(@"feed = %@", dictionary);

                // get the dictionary entries.
                NSArray *entries =[self getEntriesArray:dictionary];

                // call the delegate
                if ([delegate respondsToSelector:@selector(setVideos:)]) {
                    [delegate performSelectorOnMainThread:@selector(setVideos:)
                                               withObject:entries
                                            waitUntilDone:YES];
                }
            }
        }];
    }
```

The block of code to execute is passed in the `completionHandler` parameter of the `sendAsynchronousRequest:queue:completionHandler` method. The completion block validates that the request did not produce an error and that the HTTP status code is 200, which indicates success. If the request was successful, the returned data is parsed into an `NSDictionary`. The code then verifies that the provided delegate class supports the `setVideos:` method. If it does then it invokes that method on the main thread and supplies the array of items returned by the RSS feed. The `setVideos:` method is invoked on the main thread because the completion block is executed on a background thread, and if the delegate method is executed in that context and then tries to manipulate the user interface, the results are unpredictable but almost always bad.

Best Practices for Queued Asynchronous Requests

- Only use them when you know that the data returned will never exceed the memory available to the app. The entire body of the response is returned in-memory to your code. If the response is large, it may cause out-of-memory conditions in your app. Remember that your code may duplicate the memory footprint of the returned data when it parses it into a usable format.
- Use a single `NSOperationQueue` for all your operations and control the maximum number of current operations based on the capacity of your server and the expected network conditions.
- Always validate the error and HTTP response status code returned from the call before processing the returned data.
- Don't use them if the source URL may require authentication because this functionality does not support responding to authentication requests. You can put `BASIC` authentication credentials in the URL supplied to the request if the service requires that type of authentication.
- Don't use queued asynchronous requests if you need to provide a progress indicator to the users because the request is atomic and provides no intermediate indications of progress.

- Don't use queued asynchronous requests if you need to parse the response data incrementally via a stream parser.
- Don't use queued asynchronous requests if you may need to cancel the request before it is complete.

Asynchronous Requests

Asynchronous requests use the same ingredients as the synchronous and queued asynchronous methods but add another ingredient, the `NSURLConnectionDelegate` object.

Figure 3-4 shows the sequence of delegate calls in relation to the progress of the HTTP request. As the protocol handler proceeds through the HTTP protocol it calls the delegate methods at significant milestones throughout the connection.

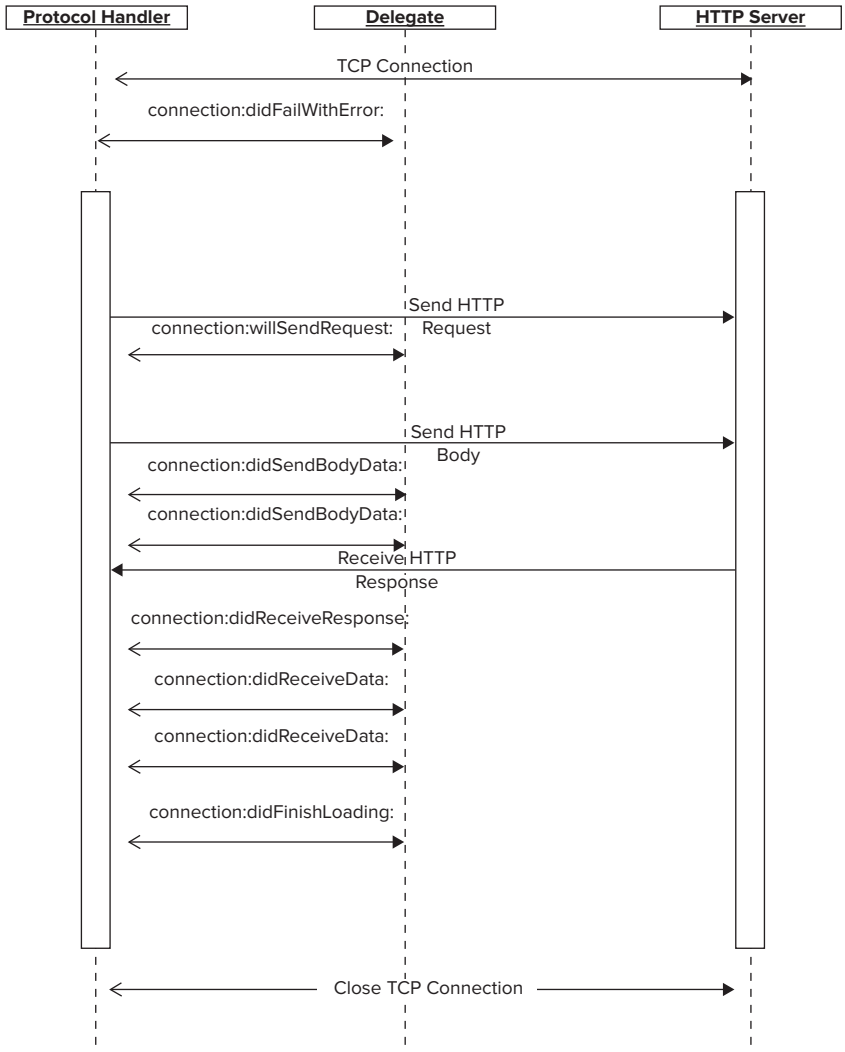


FIGURE 3-4

The protocol handler validates that the delegate implements a method before attempting to call the method. If one is not implemented, then the protocol handler assumes a default value and proceeds with the connection. The best way to explain the operation of the delegate methods is to walk through the example app.

Listing 3-3 contains the code used to initiate the URL load request using the asynchronous technique. This method starts similarly to the previous techniques: An `NSURL` object is created and then used to make a request. After the request is created, the code creates an `NSURLConnection` object and specifies itself as the delegate object. As the loading of the URL contents progresses, the protocol handler calls the delegate class with information about the state of the request. Using these callbacks the delegate class can alter the behavior of the protocol handler.

After the connection is created, the code starts the request. Between the creation of the connection and starting the connection, an app can alter how the delegate messages are delivered to the delegate class. The code can specify a different run loop or operation queue to use to deliver the callbacks. This example does not make these types of modifications.

LISTING 3-3: The start Method of VideoDownloader/AsyncDownloader.m

```
- (void) start {
    NSLog(@"Starting to download %@", srcURL);

    // create the URL
    NSURL *url = [NSURL URLWithString:srcURL];

    // Create the request
    NSURLRequest *request = [NSURLRequest requestWithURL:url];

    // create the connection with the target request and this
    // class as the delegate
    self.conn =
         [NSURLConnection connectionWithRequest:request
                                       delegate:self];

    // start the connection
    [self.conn start];
}
```

The example app implements several delegate methods to be called, but it leaves several unimplemented. The delegate methods are defined by two protocols: `NSURLConnectionDelegate` and `NSURLConnectionDataDelegate`. The following sections first review the implemented methods followed by the unimplemented ones.

connection:willSendRequest:redirectResponse: Delegate Method

The first delegate method implemented, as shown in Listing 3-4, is the `connection:willSendRequest:redirectResponse:` method. This method is called only if the connection receives an HTTP redirect response.

LISTING 3-4: Redirect Delegate Method of VideoDownloader/AsyncDownloader.m

```
- (NSURLRequest *)connection:(NSURLConnection *)connection
             willSendRequest:(NSURLRequest *)request
            redirectResponse:(NSURLResponse *)redirectResponse {

    // Dump debugging information
    NSLog(@"Redirect request for %@ redirecting to %@",
        srcURL, request.URL);
    NSLog(@"All headers = %@",
          [(NSHTTPURLResponse*) redirectResponse allHeaderFields]);

    // Follow the redirect
    return request;
}
```

This method is called if the protocol handler receives a redirect request from the server. An HTTP redirect is an HTTP response that informs the client that the content it is looking for is at a different URL. If your app loads content from a content delivery network (CDN) then redirected requests will be common. This delegate method is always called before the methods that deliver a response or body data. Because requests can undergo more than one redirect, this method may be called zero or more times for a single request. If your delegate does not implement this method, the protocol handler follows the redirect to the new location. By implementing this method your code can intercept a redirect and abort the connection or modify the request based on the nature of the redirect. In the example, the code performs a debugging function of logging the headers of the redirect request and then following the redirection.

connection:didReceiveResponse: Delegate Method

The second implemented method is the `connection:didReceiveResponse:` method, as shown in Listing 3-5. This method is called when the protocol handler has completed building a response object from the headers of the response.

LISTING 3-5: Response Received Delegate Method of VideoDownloader/AsyncDownloader.m

```
- (void) connection:(NSURLConnection *)connection
 didReceiveResponse:(NSURLResponse *)response {
    NSLog(@"Received response from request to url %@", srcURL);

    NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
    NSLog(@"All headers = %@", [httpResponse allHeaderFields]);

    if (httpResponse.statusCode != 200) {// something went wrong,
                                         //abort the whole thing
        // reset the download counts
        if (downloadSize != 0L) {
            [progressView addAmountToDownload:-downloadSize];
            [progressView addAmountDownloaded:-totalDownloaded];
        }
        [connection cancel];
        return;
```

continues

LISTING 3-5 *(continued)*

```
    }

    NSFileManager *fm = [NSFileManager defaultManager];

    // If we have a temp file already, close it and delete it
    if (self.tempFile != nil) {
        [self.outputHandle closeFile];

        NSError *error;
        [fm removeItemAtPath:self.tempFile error:&error];
    }

    // remove any pre-existing target file
    NSError *error;
    [fm removeItemAtPath:targetFile error:&error];

    // get the temporary directory name and make a temp file name
    NSString *tempDir = NSTemporaryDirectory();
    self.tempFile = [tempDir stringByAppendingPathComponent:
                                                    [self createUUID]];
    NSLog(@"Writing content to %@", self.tempFile);

    // create and open the temporary file
    [fm createFileAtPath:self.tempFile contents:nil attributes:nil];
    self.outputHandle = [NSFileHandle fileHandleForWritingAtPath:
                                                      self.tempFile];

    // prime the download progress view
    NSString *contentLengthString = [[httpResponse allHeaderFields]
                                     objectForKey:@"Content-length"];
    // reset the download counts
    if (downloadSize != 0L) {
        [progressView addAmountToDownload:-downloadSize];
        [progressView addAmountDownloaded:-totalDownloaded];
    }
    downloadSize = [contentLengthString longLongValue];
    totalDownloaded = 0L;

    [progressView addAmountToDownload:downloadSize];
}
```

The `didReceiveResponse` method is called when the protocol handler has received enough data to complete the URL response object. If an error occurs before enough data was received to complete the response object, then this method will not be called. In the example code, the delegate method validates the HTTP status of the provided response object. If the status is not a 200 status, the loading of the request is canceled, and the view that provides download progress is reset. If the status is 200, the code updates the progress view by adding the amount of data expected and creates a temporary file to receive the HTTP response body that is soon delivered to another delegate method.

This method may be called multiple times by the protocol handler; therefore, the code must handle the restart scenario. In the example, the restart logic includes resetting the progress indicator and deleting the prior response's temporary file if it exists.

connection:didReceiveData: Delegate Method

The next delegate method implemented is the `connection:didReceiveData:` method, as shown in Listing 3-6. This method is called when the protocol handler has received some or all the response body. This method may be called zero or more times on your delegate, and the calls always follow the initial `connection:didReceiveResponse:`. If you need to incrementally parse the response, the stream parser should be fed from this method.

LISTING 3-6: The didReceiveData Method of VideoDownloader/AsyncDownloader.m

```
- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data {
    // figure out how many bytes in this chunk
    totalDownloaded+=[data length];

    // Uncomment the following lines if you want a packet by
    // packet log of the bytes received.
    NSLog(@"Received %lld of %lld (%f%%) bytes of data for URL %@",
          totalDownloaded,
          downloadSize,
          ((double)totalDownloaded/(double)downloadSize)*100.0,
          srcURL);

    // inform the progress view that data is downloaded
    [progressView addAmountDownloaded:[data length]];

    // save the bytes received
    [self.outputHandle writeData:data];
}
```

This method may be called zero or more times, depending on the size of the response body. With each call the protocol handler delivers a portion of the body in the `data` parameter. It is the responsibility of the delegate method to accumulate the provided data objects and process or store them. The chunks of data provided may not correspond with the syntactic boundaries of the application protocol. In other words, if your code is receiving an XML document, the data objects will probably not correspond neatly with element boundaries in the document. In the example, the app is appending the bytes to the data file first opened in the `connection:didReceiveResponse:` method.

connection:didFailWithError: Delegate Method

The next implemented method, as shown in Listing 3-7 is called when the connection fails. This method may be called at any time during the handling of the request.

LISTING 3-7: The didFailWithError Method of VideoDownloader/AsyncDownloader.m

```
- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error {
    NSLog(@"Load failed with error %@",
```

continues

LISTING 3-7 *(continued)*

```
            [error localizedDescription]);

    NSFileManager *fm = [NSFileManager defaultManager];

    // If you have a temp file already, close it and delete it
    if (self.tempFile != nil) {
        [self.outputHandle closeFile];

        NSError *error;
        [fm removeItemAtPath:self.tempFile error:&error];
    }

    // reset the progress view
    if (downloadSize != 0L) {
        [progressView addAmountToDownload:-downloadSize];
        [progressView addAmountDownloaded:-totalDownloaded];
    }
}
```

If called, it will be the last method called for this connection. The example app just logs an error when the connection fails. It then closes the temporary download file if one exists and adjusts the progress indicator to compensate for the aborted download. The protocol handler cancels the request once this method returns. This method is a good candidate for instrumentation with analytics so that you have quantitative metrics on the failure rate of the endpoint that your app calls.

connectionDidFinishLoading: Delegate Method

The last delegate method implemented for the example app, as shown in Listing 3-8, is the `connectionDidFinishLoading` method. This delegate method is called when the entire request has been loaded and all of the data received has been passed to the delegate.

LISTING 3-8: The connectionDidFinishLoading Method of VideoDownloader/AsyncDownloader.m

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    // close the file
    [self.outputHandle closeFile];

    // Move the file to the target location
    NSFileManager *fm = [NSFileManager defaultManager];
    NSError *error;
    [fm moveItemAtPath:self.tempFile
                toPath:self.targetFile
                 error:&error];

    // Notify any concerned classes that the download is complete
    [[NSNotificationCenter defaultCenter]
     postNotificationName:kDownloadComplete
     object:nil
     userInfo:nil];
}
```

This method will be the last method called for the connection, and its invocation is mutually exclusive with the invocation of `connection:didFailWithError:`. The example app closes the file in which all the received data has been accumulated, moves the file to a location based on the URL of the originating request, and notifies the view controller via the `NSNotificationCenter` that the download has completed.

There are several other methods that a connection delegate may implement to increase the available information and possible control of the connection. The remaining parts of this section describe those methods.

connection:needNewBodyStream: Delegate Method

This method is optional and used only to re-request the input stream for the request body. It may be called if the protocol handler needs to retransmit the request body because of either an error or an authentication challenge. The method signature, shown in the following code snippet, receives the `NSURLConnection` object requesting the new stream and the request that has triggered this delegate callback.

```
- (NSInputStream *)connection:(NSURLConnection *)connection
            needNewBodyStream:(NSURLRequest *)request;
```

The delegate should return an unopened and auto-released `NSInputStream` object that references the file to be uploaded. If your code does not implement this method and uses an input stream for the body, the protocol handler first makes a copy of the entire input stream before starting the request. If your code sends a large file, you should implement this method to avoid the overhead of duplicating the file.

connection:didSendBodyData:totalBytesWritten:totalBytesExpectedToWrite: Delegate Method

This optional delegate method, as shown in the following snippet, provides upload progress information to the delegate object:

```
-(void)connection:(NSURLConnection *)connection
             didSendBodyData:(NSInteger)bytesWritten
          totalBytesWritten:(NSInteger)totalBytesWritten
  totalBytesExpectedToWrite:(NSInteger)totalBytesExpectedToWrite;
```

The protocol handler calls this delegate at undetermined intervals to report the upload progress. The `bytesWritten` and `totalBytesWritten` values may not always increase due to the need to retransmit the body if an error or authentication challenge occurs. This method should be implemented if you want to provide an upload progress indicator to your users.

connection:willCacheResponse: Delegate Method

This optional method, as shown in the following code example, provides the delegate with a means to inspect and modify the response that is cached by the protocol handler.

```
-(NSCachedURLResponse *)connection:(NSURLConnection *)connection
       willCacheResponse:(NSCachedURLResponse *)cachedResponse;
```

An `NSCachedURLResponse` object contains the `NSURLResponse` object and the data, as an `NSData` object, returned from the request. The object also contains the storage policy to be used for the retention of the response, which includes persistent storage, in-memory storage only, or storage not allowed. The cached response object also contains a `userInfo` dictionary that can be used by your application to store metadata with the cached request. If your delegate implementation returns `nil`, the response is not cached.

Authentication Delegate Methods

There are five delegate methods associated with client authentication of URL requests. The use of these methods is fully described in Chapter 6.

Asynchronous Requests and Run Loops

Asynchronous requests require a run loop to operate. As data is transmitted to the server or received from the server, the run loop is used to communicate the events to the delegate object. When you start an asynchronous request, it operates on the run loop of the current thread. This implementation detail is important because threads created in Grand Central Dispatch blocks or via an `NSOperationQueue` do not have a run loop. Therefore, if you start an asynchronous request on a background thread, you also need to make sure that either the thread has a run loop or it uses another run loop. The following snippet shows how to explicitly assign the processing of a request to a run loop.

```
NSURLConnection connection = [[NSURLConnection alloc]
                             initWithRequest:request
                                    delegate:self
                            startImmediately:NO];

[connection scheduleInRunLoop:[NSRunLoop mainRunLoop]
                      forMode:NSDefaultRunLoopMode];
[connection start];
```

The first operation creates the `NSURLConnection` object but does not immediately start the method; leaving it in a state where further initialization is possible. The next line of code fetches the run loop for the main thread and supplies it to the connection as its run loop. Finally, the connection begins processing via the `start` method. If you do not want to run the asynchronous request on the main run loop you need to create a run loop on another thread and schedule the connection for the newly created run loop.

Best Practices for Asynchronous Requests

- Use asynchronous requests for large uploads or downloads to reduce the memory footprint of the app.
- Use asynchronous requests when authentication may be required.
- Use asynchronous requests when you need to provide progress feedback to the user.
- Be careful using asynchronous requests on background threads; make sure to provide a run loop.

- Asynchronous requests are overkill for small requests that can be easily scheduled and completed on a request queue in a background thread.
- If you use an input stream to upload data, implement the `connection:newBodyStream:` method to avoid duplication of the input stream.

ADVANCED HTTP MANIPULATION

HTTP headers play an important role supplying metadata that modifies the server's response or provides additional information to the HTTP client. Because of this, iOS developers frequently need to manipulate request headers or examine response headers. For example, some servers require the addition of a custom authentication header that provides information about the user's identity. The standard URL loading system does not add these headers automatically, but it provides methods to add them from your code.

In this section you learn about additional HTTP operations and manipulations available via the URL loading system of iOS. The subsections describe how to create and use alternative HTTP request methods, how to handle HTTP cookies, and the advanced use of HTTP headers.

Using Request Methods

At the beginning of this chapter, several additional HTTP methods were referenced in Table 1-1. The most common type of request is a `GET` request, but it is sometimes abused and used where one of the other methods, `PUT`, `POST`, `HEAD` or `DELETE`, would be more appropriate.

By its definition, a `GET` request should not include an HTTP body; it should just be the request line and request headers. The HTTP server returns the contents of the resource specified by the URL. It is common for networking equipment to assume that the complete context of the `GET` request is in the request line and cache responses based on that data. If your `GET` request has a request body that alters the content returned by the request, then you may get incorrect results due to the caching behavior of any intermediate networking equipment. By convention, `GET` requests should not cause changes to any data on the server.

`POST` requests were first used by HTML browsers to send, or post, data for HTML forms using a specific data encoding, specified by a `Content-Type` of `application/x-www-form-urlencoded`. iOS apps typically use `POST` requests to send XML or JSON data to a server. The following code snippet shows the creation of a JSON object and how to supply it as the request body.

```
NSError *error;
NSDictionary *dict =
     [NSDictionary dictionaryWithObjectsAndKeys:
                      @"dog", @"animal",
                      @"fido", @"name",
                      @"20", @"weight", nil];
NSData *jsonData = [NSJSONSerialization
                    dataWithJSONObject:dict
                    options:NSJSONWritingPrettyPrinted
                    error:&error];

if (error != nil) { // encoding succeeded
```

```
    NSLog(@"Error on encoding dictionary");
    return;
}
NSLog(@"Json = %@", [[NSString alloc]
                     initWithData:jsonData
                     encoding:NSUTF8StringEncoding]);
NSMutableURLRequest *request = [NSMutableURLRequest
                                    requestWithURL:url];
[request setHTTPMethod:@"POST"];
[request setHTTPBody:jsonData];
```

The code first creates a simple `NSDictionary` with a few name-value pairs about a fictional animal. It then uses the built-in JSON library to create an `NSData` object that represents the dictionary. The `NSData` object is then supplied to an `NSMutableURLRequest` object as the request body. The JSON produced by this code is shown in the following snippet. Chapter 4 describes the process to build and parse request and response payloads in more detail.

```
{
  "weight" : "20",
  "animal" : "dog",
  "name" : "fido"
}
```

A request using the `HEAD` method is instructing the HTTP server to return only the HTTP header information about the requested resource. `HEAD` requests typically do not have request bodies and expect no response body in return. They are commonly used to validate cached data against the data on a server without retrieving the entire contents of the cached resource.

A `PUT` request is similar to a `POST` because it should always have a request body but differs semantically in one key way: A `PUT` request is used to add a new resource to the server, whereas a `POST` is used only to update a resource on the server. This semantic difference is important when interfacing with RESTful services, as described in Chapter 4.

Cookie Manipulation

Cookies are an important component of the HTTP protocol that were added after the initial version. Cookies provide the capability for the server to track the state of the conversation without maintaining a network connection between the client and server. In a browser client, the value of the cookie is provided by the server on a request and then included on subsequent requests. Because cookies are designed to track session state, they are typically quite small, usually tens to hundreds of bytes.

A cookie sent from a server has several properties that determine the value of the cookie, when it is returned to the server, and how long the client should retain the cookie. Those properties include the following:

- **Name —** The name of the cookie, which should be unique across all cookies returned from that DNS domain. The name and the value are the only two properties supplied to the server on subsequent requests.
- **Value —** The value to be returned on the next request to the server.

- **Domain —** The DNS domain for which subsequent requests should contain the cookie. For example, a cookie with a domain value of `.domain1.com` should not be returned to `.domain2.com`. If omitted, the client should use the hostname of the URL as the domain. If the domain is prepended with a period (.) the cookie should be returned for any request to that domain or any of its subdomains. If the leading period is absent, then the cookie is included only in requests to that domain and not to its subdomains.
- **Path —** The path constrains the delivery of a cookie to requests that are destined for the specified URL path. When combined with the DNS domain, the path property can restrict the delivery of a cookie to a limited and precise set of URLs on a server.
- **Expiration Date —** The date and time at which the cookie should no longer be supplied in requests and at which the cookie should be removed from the client's storage.
- **Session Only —** Indicates whether the cookie should be returned only for the duration of the current browser session or until the expiration date, whichever comes first. In an iOS app the session is the lifespan of the app between the OS loading the app and the OS terminating the app.
- **Secure —** Indicates that the cookie should by supplied only over HTTPS connections and not over HTTP connections.
- **Comment —** A comment value that is intended to explain to the user the purpose of the cookie.
- **Comment URL —** A URL value that provides to the user an HTML document explaining the purpose of the cookie.
- **HTTP Only —** An indicator instructing the client to not share the cookie with JavaScript applications to prevent cross-site scripting.
- **Version —** The version of the HTTP Cookie specification to which the cookie conforms.

Although it is not a browser, an iOS app can still use cookies advantageously in HTTP connections. The URL loading framework does much of the heavy lifting necessary to leverage this feature of the protocol. There are three things that apps typically need to do with cookies: retrieve the value of a cookie, explicitly delete a cookie, and artificially add a cookie to a request.

The URL loading facility automatically handles cookies for all HTTP and HTTPS requests. It saves cookies returned in responses and automatically adds them to subsequent requests per the rules for cookie handling. Cookies are returned only to hosts in the DNS domain supplied in the domain property of the cookie. Conveniently, the value of nonsession cookies is persisted across launches of the app. So, if your app retrieves a cookie and then is terminated, the retrieved cookie will still be present in subsequent invocations of the app. The URL loading system sends cookies only when they are unexpired and valid for the target domain.

The URL loading system provides two significant objects for the management of cookies: `NSHTTPCookie` and `NSHTTPCookieStorage`. `NSHTTPCookie` contains the representation of a cookie with all its required and optional properties. `NSHTTPCookieStorage` is a singleton object that manages the cookies for your application. Note that `NSHTTPCookieStorage` cookies are sandboxed like all other application data and cannot be shared across applications.

`NSHTTPCookieStorage` provides the capability to control which cookies are retained, but by default it stores any cookies returned in a response, regardless of whether the cookie's domain matches the domain of the request. Using the cookie acceptance policy, your code can control how eagerly it saves cookies. The values of the storage policy are as follows:

- **NSHTTPCookieAcceptPolicyAlways —** This is the default value and indicates that any returned cookie should be retained.
- **NSHTTPCookieAcceptPolicyNever —** This value indicates that no cookies should be stored.
- **NSHTTPCookieAcceptPolicyOnlyFromMainDocumentDomain —** This policy instructs the `NSHTTPCookieStorage` object to retain only cookies whose domain value matches the domain of the request.

The following snippet would prevent an application from retaining cookies.

```
[[NSHTTPCookieStorage sharedHTTPCookieStorage]
     setCookieAcceptPolicy:NSHTTPCookieAcceptPolicyNever];
```

Your code can also stop the automatic handling of cookies on a request-by-request basis by calling the `setHTTPShouldHandleCookies:` with a value of `NO` on an `NSMutableURLRequest` before submitting it. This prevents the URL loading system from processing the returned requests.

Retrieving Cookies from a Response

It's a common task to retrieve the cookies from a response and look for a specific cookie by name. The following code snippet shows how to perform a request and then find the session ID returned by a JavaEE web server.

```
NSMutableURLRequest *req = [NSMutableURLRequest
                            requestWithURL:url];

NSHTTPURLResponse *response;
NSError *error;

// make a request
NSData *data = [NSURLConnection
                 sendSynchronousRequest:req
                      returningResponse:&response
                                  error:&error];

// get the full set of headers and display them
NSDictionary *headers = [response allHeaderFields];
NSLog(@"Headers = %@", headers);

// extract the set-cookie headers and split them into cookies
NSArray *cookies = [NSHTTPCookie cookiesWithResponseHeaderFields:headers
                     forURL:url];

// look for the cookie with the name JSESSIONID
for(NSHTTPCookie *cookie in cookies) {
```

```
        NSLog(@"Cookie: %@", cookie);
        if ([[cookie name] isEqualToString:@"JSESSIONID"]) {
            NSLog(@"Found the session id");
        }
    }
```

The request pattern is the same as other requests. The `response` object is declared as an `NSHTTPURLResponse` and does not require a cast later. The `allHeaderFields` method returns a dictionary containing all the header names and values. If a header appears multiple times in the response, then the values are concatenated together with separating commas. The `cookiesWithResponseHeaderFields` returns an array of `NSHTTPCookie` objects, each containing a cookie found in a `SetCookie:` header. Remember that this set of cookies may not exactly match the cookies that will be supplied on the next request. Some of the cookies may expire before the next request, and `NSHTTPCookieStorage` may already contain additional cookies that will be added to the next request.

Deleting Cookies

Occasionally, you may need to delete a cookie from the persistent cookie storage so that it is no longer added to outgoing requests. Because cookies are added to the request after your code starts the connection via the asynchronous `start` or the `sendSynchronousRequest` method, you cannot prevent a cookie that already exists in the cookie storage from being added to the request. Instead, your code must first remove the cookie from the apps `NSHTTPCookieStorage` object prior to starting the request.

The following snippet shows how to clear all the cookies from the cookie storage.

```
- (void)deleteAllCookies
{
    // get the shared cookie jar
    NSHTTPCookieStorage *jar = [NSHTTPCookieStorage
                                sharedHTTPCookieStorage];

    // get all the cookies
    NSArray *storedCookies = [jar cookies];

    // delete them all
    for(NSHTTPCookie *cookie in storedCookies) {
        [jar deleteCookie:cookie];
    }
    [[NSUserDefaults standardUserDefaults] synchronize];
}
```

The first two lines of the method retrieve the singleton cookie storage object for the application. The next line retrieves an `NSArray` of all the cookies stored. The loop that follows iterates over each of the cookies in the array and blindly deletes each one.

Your code can selectively delete a cookie by using any combination of cookie properties. The following snippet deletes a cookie by name for a specific URL. Remember that cookie names are only unique to the target domain.

```
- (void)deleteCookie:(NSString *)cookieName url:(NSURL *)url {
    // get the shared cookie storage object
    NSHTTPCookieStorage *jar = [NSHTTPCookieStorage
                                sharedHTTPCookieStorage];

    // get the cookies for the supplied URL
    NSArray *storedCookies = [jar cookiesForURL:url];

    // iterate over the list of cookies
    for(NSHTTPCookie *cookie in storedCookies) {

        // if the cookie name matches the target, delete it
        if ([cookieName isEqualToString:[cookie name]]) {
            NSLog(@"Deleting cookie %@", cookie);
            [jar deleteCookie:cookie];
        }
    }
}
```

This code uses the supplied URL to filter the set of cookies returned from the cookie storage. After the list of cookies is returned, the method looks through the list for a cookie with a matching name, and if found, that cookie is deleted.

The `deleteCookie:url:` method takes two arguments, the name of the cookie to delete and a URL to which that cookie would be returned if a request were made to it. The rules for determining which cookies to return to a domain allow for global cookies for a domain and cookies for a specific host. Table 3-2 illustrates the application of the domain mapping rules across a domain and subdomain.

TABLE 3-2: Cookie Domain Mapping

COOKIE DOMAIN	COOKIE SUBMITTED IN REQUEST TO HTTP://WWW.HOST.COM	COOKIE SUBMITTED IN REQUEST TO HTTP://HOST.COM
`.host.com`	Yes	Yes
`.www.host.com`	Yes	No
`host.com`	No	Yes
`.www.nothost.com`	No	No

If a request is made to `http://www.host.com`, then a cookie with a domain of `.host.com` or `.www.host.com` will be added to the request. If a request is made to `http://host.com`, then a cookie with a domain of `.host.com` will be added, but a cookie with a domain of `.www.host.com` will not be added.

Creating Cookies

Cookies can be created and programmatically added to requests or the cookie storage. This might be necessary if you want to artificially create cookies to conform to a web application's unique protocol. It may also be necessary to receive a cookie for one domain and create an identical cookie to send to another domain. This section shows two methods to accomplish this task.

The first snippet creates a cookie and adds it to an individual request.

```
// create a dictionary of properties
NSDictionary *properties = [NSDictionary
                            dictionaryWithObjectsAndKeys:
                            @"FOO",NSHTTPCookieName,
                            @"This is foo", NSHTTPCookieValue,
                            @"/", NSHTTPCookiePath,
                            url, NSHTTPCookieOriginURL,
                            nil];
// using the properties make a cookie
NSHTTPCookie *cookie = [NSHTTPCookie
                        cookieWithProperties:properties];

// create a mutable request
NSMutableURLRequest req = [NSMutableURLRequest
                           requestWithURL:url];

// make an array of 1 cookie
NSArray *newCookies = [NSArray arrayWithObject:cookie];

// make a dictionary of the headers required to send the cookie
NSDictionary *newHeaders = [NSHTTPCookie
                      requestHeaderFieldsWithCookies:newCookies];
// make the cookie headers the headers of the request
[req setAllHTTPHeaderFields:newHeaders];

// send the request
[NSURLConnection sendSynchronousRequest:req
                      returningResponse:&response
                                  error:&error];
```

The first segment of the code creates a dictionary containing the properties of the new cookie using a helpful set of `NSHTTP` constants for the property names. Using that dictionary, called `properties` in the snippet, a new `NSHTTPCookie` is instantiated. An `NSMutableURLRequest` object is then created using the default properties.

The next segment goes through a little dance to make the correct structure to add to the request. It first makes an `NSArray` containing the single new cookie. Then using that array it creates an `NSDictionary` object with the headers to put into the HTTP request to represent the cookies. It then replaces the default header contents with the new headers containing the cookie information. If your code needed to add other headers, it would do so by adding them to the `newHeaders` dictionary before `setAllHTTPHeaderFields:` is invoked.

The second way to add cookies to a request, as shown in the following code snippet, is cleaner but adds the cookie for all subsequent requests.

```
// create the properties for a cookie
NSDictionary *properties = [NSDictionary
                            dictionaryWithObjectsAndKeys:
                            @"FOO",NSHTTPCookieName,
                            @"This is foo", NSHTTPCookieValue,
                            @"/", NSHTTPCookiePath,
                            url, NSHTTPCookieOriginURL,
                            nil];
// create the cookie from the properties
NSHTTPCookie *cookie = [NSHTTPCookie
                        cookieWithProperties:properties];

// add the cookie to the cookie storage
[[NSHTTPCookieStorage sharedHTTPCookieStorage] setCookie:cookie];

// create and issue a request
req = [NSMutableURLRequest requestWithURL:url];
[NSURLConnection sendSynchronousRequest:req
                      returningResponse:&response
                                  error:&error];
```

The code starts out identically to the prior example by creating a cookie, which is then stored in the apps cookie storage. Keep in mind that if the cookie storage acceptance policy has been restricted, the cookie may not be stored. After storing the cookie, the new request automatically pulls it from the cookie storage when it begins executing.

Advanced Headers

In addition to manipulating the body of a request and the cookies sent with a request, it is often necessary to add or remove headers to or from a request and examine headers in a response.

The following sections describe how to manipulate HTTP request and response headers, and why it may be important to closely manage the headers sent from your app.

Adding Request Headers

When your code needs to change request headers, it must create an `NSMutableURLRequest` object rather than an unalterable `NSURLRequest` object. The `NSMutableURLRequest` class provides two ways to add headers to a request: one header at a time or by replacing all headers.

The `setAllHTTPHeaderFields:` method provides a way to replace all the request headers with a single call. This method was used in the cookie examples to add a cookie to the request. It can be helpful if your code needs to add headers based on a dynamic set of data. If the data set is encoded in a dictionary, it can be added to any outgoing request with a single call.

Headers can also be added to a request individually. The following code snippet shows how to add an individual header to a request.

```
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req addValue:@"en" forHTTPHeaderField:@"Content-Language"];
[req addValue:@"da" forHTTPHeaderField:@"Content-Language"];
```

The first line instantiates a request object named `req`. The second line adds a header with a name of `Content-Language` and a value of `en` to the request. The HTTP protocol specifies that header names end with a colon, but this method appends a colon if omitted. If the same header name is added multiple times, the header values concatenate together with a separating comma. In the preceding example, the resulting header was transmitted as follows:

```
Content-Language: en,da
```

Removing Request Headers

Sometimes it is necessary to remove a header from a request. For example, if an app wants to disable compression of the returned data, it can override the default iOS header that indicates supports for `gzip` or `DEFLATE` compression:

```
Accept-Encoding: gzip, deflate
```

This indicates to the server that it can compress the returned response body. The following code overrides the `Accept-Encoding` header from the request:

```
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req setValue:@"" forHTTPHeaderField:@"Accept-Encoding"];
```

Using the URL loading system API, there is not a way to completely delete one of the standard headers that the API includes in the request. The best you can do is to override the value with an empty value.

Examining Response Headers

When an HTTP request completes without errors, it may include zero or more headers. The following code snippet illustrates a request that, upon successful completion of an HTTP request, dumps the headers to the log file and retrieves the `MIME` type of the response.

```
NSHTTPURLResponse *response;
NSError *error;
[NSURLConnection sendSynchronousRequest:req
                      returningResponse:&response
                                  error:&error];

NSDictionary *headers = [response allHeaderFields];
NSLog(@"Headers = %@", headers);

NSString *contentType = [headers objectForKey:@"Content-Type"];
```

This is safe to do if the request URL will always use either the HTTP or HTTPS protocol scheme. The `allHeaderFields` method returns a dictionary of all the response headers, including `Set-Cookie` headers. The last line of the snippet retrieves the value of the `Content-Type` header. If the header was omitted from the response, the returned value will be `nil`. If the response contained

multiple headers of the same type, only one value will be returned. This single value contains all the response values concatenated together with a comma separating each value.

Key Request Headers

Several HTTP headers are more commonly manipulated in requests than others. This section discusses the use cases for these headers.

Many REST server implementations determine the data encoding for the returned payload by examining the `Accept` header. For example, an `Accept` header with a value of `application/xml` would instruct the server to return an XML document, whereas `application/json` would cause a JSON document to be returned.

The `Authorization` header can be prepopulated with authentication credentials to avoid a credentials check response from the server. The following snippet shows how to add a `BASIC` authentication header:

```
NSString *basicBody = [NSString stringWithFormat:@"%@:%@",
                       username, password];
NSData *authData = [basicBody
                       dataUsingEncoding:NSASCIIStringEncoding];

// Base64 encode authData into a string called b64Data
// code omitted to perform the Base64 encoding

NSString *authValue = [NSString stringWithFormat:@"Basic %@",
                    b64Data];
[theRequest setValue:authValue
  forHTTPHeaderField:@"Authorization"];
```

The code creates the body of the authentication data, called `basicBody`, which contains the username and password separated by a colon (`:`). The `basicBody` is then converted into an `NSData` object containing the ASCII values of the string. Those bytes are then Base64-encoded into a string, called `b64Data`. That string is then prepended with the word `Basic` set as the value for the `Authentication` header.

As you can tell `BASIC` authentication provides no more security than Base64-encoding the password, which means you should use only this approach over HTTPS connections. The Base64 code is omitted from the example, but there are numerous libraries and categories available online to Base64-encode data. Chapter 11, "Inter-App Communication," contains additional information about Base64 encoding.

Another header that may be necessary to modify is the `User-Agent` header, which was created to identify the browser type to the HTTP server. In many enterprise networks, the user agent value is used to direct traffic to specific servers based on the browser type. Some HTTP servers modify the content of the response based on the user agent value. By default, the user agent from your app will contain the app product name and version, the network framework and version, and the OS name and version. The `User-Agent` value for the VideoDownloader application is shown here.

```
VideoDownloader/1.0 CFNetwork/548.1.4 Darwin/11.0.0
```

The `User-Agent` header value for your app will differ based on the iOS version of the device and will also be different when run in the iOS Simulator. If your network infrastructure employs user agent-sensitive routing, it must accommodate these differences. There are two ways to accommodate the differences:

- Override the `User-Agent` header to provide a value consistent across platforms.
- Route on a subset of the User-Agent header, such as only the application name. The remainder of the header will change based on the version of the app and the device platform on which it is running.

Your code is not limited to using only standard headers for requests. You can define your own custom headers with almost any header name; however, these custom names must not contain colons or whitespace. Header names are case-sensitive, and you need to coordinate with web service developers to determine precise header names and values. Custom headers are useful for transporting authentication and control data that can be used by the application infrastructure to support the application's business logic. One example is to provide a session identifier via a custom header so that the application server can associate a session context with the request payload before sending it to the application logic.

Custom headers are added to a request in the same way as standard headers, as shown in the following code snippet:

```
NSMutableURLRequest *req = [NSMutableURLRequest requestWithURL:url];
[req setValue:@"myValue" forHTTPHeaderField:@"My-Custom-Header"];
```

SUMMARY

HTTP requests have become the mainstay of app developers wanting to connect to enterprise services. iOS provides a simple yet robust API to perform the vast majority of these requests without the need to delve into lower-level APIs. Although it was originally created for browsers, the HTTP protocol is flexible enough to suit almost any request and response interaction between your app and a remote server.

Using the URL loading system and `NSURLRequest`, `NSURLResponse`, and `NSURLConnection`, you can issue requests with only a couple of lines of code. As the communications become more complex, you can continue using this toolset to issue requests with tight integration into your app code. Using the URL loading system API, you can control both the payload and request headers to completely master the communications between your app and your server.

4

Generating and Digesting Payloads

WHAT'S IN THIS CHAPTER?

- Using web service protocols and styles and understanding their impacts on mobile applications
- Understanding common payload formats
- Digesting XML, HTML, and JSON payloads
- Generating JSON and xml payloads

WROX.COM DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. You can find the code for this chapter in the Chapter 4 download in a single Xcode project with the following major components:

- A newsreader application that parsers XML, HTML, and JSON data and generates JSON and XML payloads. The source code for this application can be found in the `App.zip` archive.
- An archive of CNN articles for use with the example app. The articles were saved the day the application was developed. Article content can be found in the "News Content" directory in the `App.zip` archive.
- A server-side PHP script to process XML and JSON payloads generated in the application. The source code for the server side component is available in the `Server.zip` archive.

As more companies invest in initiatives to mobilize their work force, integrating enterprise services into applications will become more common. Successfully communicating with web services, internal to the organization or not, requires the ability to generate and digest the structured data transmitted. This structured data forms the contract between your application and the web service and enables each to be updated independently as long as the structure does not change.

This chapter compares and contrasts two popular web service implementations and the considerations that you should take with mobile in mind. This chapter also reviews common data interchange formats and covers how to create and interpret payloads using a combination of native iOS APIs. A *payload* is the body of data representing the intent of the transmission being processed.

For situations in which you have control over the design of the service, Chapter 2, "Designing Your Service Architecture"; covers possible web service design patterns. These patterns are meant to optimize the service for mobile devices and maximize the reuse of business logic in the service tier. This leads to a more streamlined payload digestion process (discussed in this chapter), which enables you to deploy applications to multiple channels, for example the web, more quickly.

WEB SERVICE PROTOCOLS AND STYLES

A *protocol* is a set of formats, procedures, and rules to follow when exchanging structured information with another system. Among other things, protocols define the data format to be used during the transmission. This enables receiving systems to properly interpret the structured information and react accordingly. This section compares and contrasts SOAP, a popular protocol within enterprise applications, with REST, an architectural design style that is well suited for web services targeting mobile devices.

Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (*SOAP*) is a lightweight protocol meant for exchanging structured data between systems using Extensible Markup Language (XML). SOAP was originally designed for Microsoft in 1998 to facilitate data interchange in a manner convenient to definitions, methods, and procedures in common programming languages. Version 1.2 of the SOAP specification became the W3C recommendation in June 2003, with enhancements included in "Part 2" following in April 2007.

SOAP messages can form the foundation of a web service stack, and many enterprises have established service tiers with SOAP as the backbone of their Service-Oriented Architectures (SOA) that serve clients inside and outside the firewall. Although messages are typically one-way transmissions between SOAP nodes, the protocol was built with a more conversational mode in mind. A SOAP node is a logical processing unit that can transmit, receive, process, or relay an individual SOAP message.

A SOAP message consists of an envelope that includes a header and a body, as shown in Figure 4-1. The header is optional, as depicted by the dashed objects in Figure 4-1, and contains service-level information, typically authentication and session management data. The body is the primary content of the message. It contains the information to be processed by the ultimate receiver of the message. SOAP was designed to use a variety of transport layers such as HTTP, e-mail, and asynchronous queues. This helped SOAP become a single solution for many interconnectivity problems, but it was designed before the explosion of mobile devices.

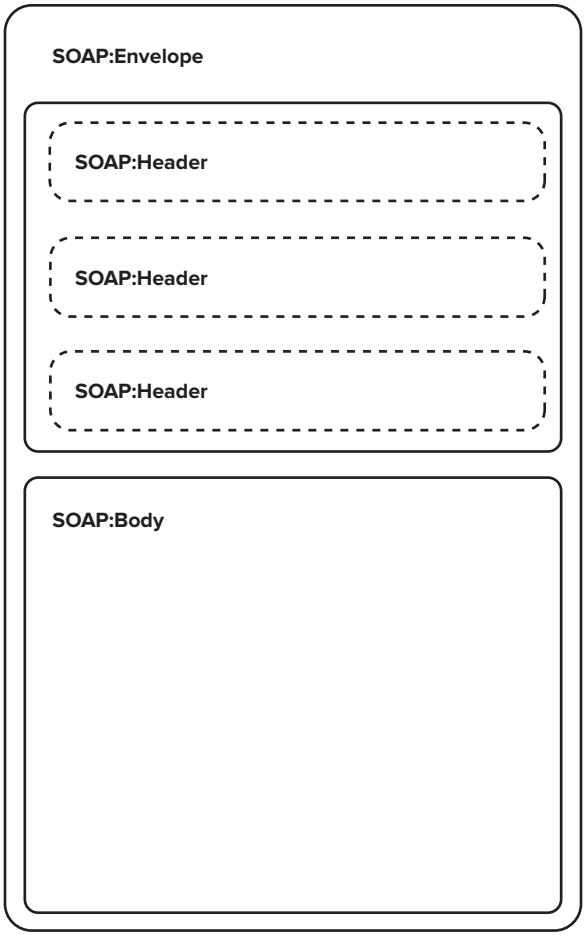


FIGURE 4-1

The content and structure of the message body depends on the receiving system. A body can consist of multiple child elements, each of which can be namespace qualified. Each child element includes the operation that should be performed by the receiver as well as any necessary parameter values. The following code snippet provides a sample SOAP body.

```
<soap:Body>
  <m:GetWeather xmlns:m="http://www.<domain>.com/weather">
    <m:ZipCode>95014</m:ZipCode>
  </m:GetWeather>
</soap:Body>
```

The preceding code includes a SOAP body that instructs the receiver to execute the `GetWeather` operation. This operation is namespace qualified, indicated by the `xmlns:m` attribute. This operation requires that the `ZipCode` parameter is passed to provide an appropriate response.

You can specify the data type for a specific operation parameter by including an additional attribute within the parameters start-tag like so:

```
<m:ItemNumber xsi:type='xsd:int'>95014</m:ItemNumber>
```

The preceding code specifies that the data type associated with the element content, `95014`, is an integer. Table 4-1 outlines some common data types supported by the SOAP protocol and their corresponding attribute names.

TABLE 4-1: Common SOAP Data Types

DATA TYPE	ATTRIBUTE NAME
Integer	`xsd:int`
Boolean	`xsd:boolean`
String	`xsd:string`
Float	`xsd:float`
Double	`xsd:double`
Array or Dictionary	`xsd:struct`

Generating SOAP client-side code from *Web Service Description Language* (*WSDL*) and *XML Schema Definition* (*XSD*) documents can be complex. This complexity is exacerbated by the need to develop for multiple platforms and device families. Another pitfall of SOAP is that changes can be difficult in the mobile environment. When two internally operated servers communicate, controlling changes to those servers is simple. It's an entirely different story when you rely on users to update their applications. In light of these difficulties, there is another web service design style that tends to be more appropriate for services targeting mobile devices.

Representational State Transfer (REST)

Representational state transfer (*REST*) was introduced in 2000 by Roy Fielding, one of the principle authors of the RFC, "Hypertext Transfer Protocol (HTTP)," as part of his doctoral dissertation. Although often referred to as such, REST is not a standard or protocol; REST is an architectural design style that can be applied to web services. Although REST was developed in parallel with HTTP 1.1 and is often associated with the protocol, it is not limited to HTTP as the sole application layer protocol. The largest implementation of the REST design style is the World Wide Web. Services that implement a REST style interface are commonly referred to as *RESTful*.

One central facet of REST is the concept of a *resource*, which has a global identifier. This concept of *uniform resource identifiers* (*URI*s) distinguishes REST from other architecture styles. Resources can be thought of as anything exposed independent of its representation. For example, `/user/accounts` may be an endpoint to retrieve a JSON list of account resources. In addition, `/user/account/123` may be an endpoint to retrieve a specific image representation of an account resource with the number `123`.

REST requires you to put a greater emphasis on schema design, approaching it from the resource perspective versus designing in terms of actions or services like SOAP. You can think of resource identifiers as nearly complete sentences; having both a subject (for example, `/user/account/123`) and a verb (for example, the HTTP method used in the request–`POST`, `GET`, `PUT`, or `DELETE`). This lends itself to being both machine- and human-readable.

A RESTful architecture has two other key attributes: They are both stateless and cacheable. Stateless interaction requires that a request contain all necessary information, which may typically be included in a session, to understand the context of the transmission. The overhead associated with transmitting this information with each request offsets some of the benefit associated with REST response payloads, which are typically lighter-weight than other service patterns. Additionally, clients can easily cache responses because each resource has a unique, global identifier. Static resources, such as images, are also prime candidates to be hosted with a *content delivery network* (*CDN*) because they can be cached on their expansive server network and served quickly upon request. Endpoints within a RESTful service can return different data types. For example, some may return a resource representation as JSON formatted data, whereas others may return an image.

Choosing an Approach

SOAP-based services are still actively deployed in many enterprises, especially those that run some of the more popular packaged software solutions such as enterprise resource planning (ERP) software. SOAP and REST try to solve the same problem in different ways. Although neither protocol is a perfect fit for every situation, REST style service architecture is the best design for mobile optimized service tiers.

One false assumption is that SOAP is more secure than REST. This assumption arises because there are specific security methods included as part of the overall SOAP, namely WS-Security. However, the reason WS-Security was created is largely because the SOAP specification was transport-independent, and no assumptions could be made about the security available on the transport layer. As with any secure application, security must be designed into the architecture and proper discipline must be followed.

Another assumption that doesn't hold true in the mobile world is that the remote device is trustworthy. When SOAP is used to communicate between known application servers within a network, this may be true. For mobile devices however, which can easily be compromised, the opposite is true: You need to assume that the remote device is not trustworthy.

As with any data protocol, security requires good design and discipline throughout the development life cycle; REST is no different. REST security must be designed with the application data in mind. One must carefully consider the data transmitted to ensure that only the minimal amount of data is sent to allow the application to function as required.

Exposure of Personally Identifiable Information (PII) can happen just as easily when using REST as when using HTTP or SOAP. PII should never be sent to a mobile device unless absolutely necessary and where the reward outweighs the risk. Although deviating slightly from the constraints of the REST style, requests from the device should leverage information held in the server session to verify the semantic correctness of any inbound payload.

When implemented properly, RESTful style architecture is the best approach for delivering resources to the mobile channel. Among other benefits, RESTful services offer the best combination of the following:

- Developer familiarity and productivity
- Performance
- Network efficiency
- Opportunity for security
- Robustness
- Interface flexibility

In addition, one best practice is to consolidate all external service calls to a single Mobile Façade built in the REST style and deliver resources in JSON format, which is discussed next in the "Payloads" section. For a more detailed overview of the Mobile Façade architecture pattern, see Chapter 2.

PAYLOADS

Payloads are the essential data exchanged during the service request-response transaction. For example, in a `POST` request, the payload is the request body. Payloads do not include the overhead data such as request headers or the HTTP method being requested; `POST` in the example. If your application is sending or receiving information from a web service you need to have a firm understanding of the payload format for requests and responses.

This section provides an overview of common payload formats including XML, JSON, and HTML. Once you have a firm understanding of those typical data exchange formats, you can practice detailed examples of how to digest web service responses and integrate the received data into your applications. Many applications also need the ability to send structured payload data to a web service, therefore the final topic in this section covers creating XML and JSON output and provides examples of how to structure those requests.

Introducing Payload Data Formats

Inbound and outbound payload data comes in many shapes and sizes. For example, some developers choose to communicate with their web services using raw strings or pipe-delimited data. Although simple, that technique is not extensible, struggles to handle complex data structures, and could lead to numerous downstream issues. This section covers three alternative, standardized formats for sending and receiving structured data: *Extensible Markup Language (XML)*, *JavaScript Object Notation* (*JSON*), and *Hypertext Markup Language* (*HTML*).

XML

XML is a markup language for encoding and structuring data. The XML specification, an extension of the *Standard Generalized Markup Language* (*SGML*), was started in 1996 and is produced by the *World Wide Web Consortium* (*W3C*). The fifth revision was published in November 2008. The initial focus of XML was on documents, but it has been widely adopted as a format to transmit

structured data in web services. XML has been extended to a number of niche markup languages and protocols such as VoiceXML for structuring voice dialogs, Open Financial Exchange (OFX) for exchanging financial data, and *Really Simple Syndication* (*RSS*) for publishing media content. As mentioned in the last section, SOAP is a protocol for exchanging XML structured data.

XML documents contain markup and content. Markup consists of *tags*, *attributes*, and *elements*. There are three types of tags: start-tags (`<person>`), end-tags (`</person>`), and empty-element tags (`<noContact />`). Empty-element tags are also referred to as *self-closing tags*. Attributes are key-value pairs within the start-tag or empty-element tag that provide additional information about the element.

Elements are the components that comprise an XML document. Elements are a collection of tags, attributes, and content. An element consists of both a start-tag and end-tag or an empty-element tag. The data between the start-tag and end-tags is the content. Content can contain markup, including other elements, which enables you to build parent-child relationships into your data structure. The following code snippet provides an example of an XML element.

```
<person>
    <firstName>Nathan</firstName>
    <lastName>Jones</lastName>
    <emailAddress primary='true'>email@domain.com</emailAddress>
    <noContact medium='email' />
</person>
```

The preceding code outlines a `person` element that contains several child elements: `firstName`, `lastName`, `emailAddress`, and `noContact`. The `emailAddress` element contains an attribute indicating that the element's content is this person's primary e-mail address. The `noContact` element, which indicates that this person has opted-out of being contacted, contains an attribute indicating the medium in which he should not be contacted.

JSON

JSON is a lightweight data format to exchange structured information. It is believed that JSON was conceived and used in 2001; however, RFC4627 (`http://tools.ietf.org/html/rfc4627`), which documents the `application/json` media type, was not published until July 2006. JSON has experienced significant adoption and growth fueled, in part, by the explosion of mobile applications as they sought efficient, easy-to-understand, compact methods for exchanging data over cellular data networks.

JSON has a small, defined set of formatting rules that must be followed when creating payloads. Following are the supported data types and their associated formatting rules:

- **Numbers:** Unquoted.
- **Boolean:** `true` or `false`; unquoted.
- **Strings:** Double-quoted.
- **Arrays:** Comma-separated lists enclosed in square brackets.
- **Objects:** Collection of `key:value` pairs enclosed in curly braces. Objects are represented in Objective-C using `NSDictionary`.
- `null`: Unquoted.

The root type of properly formatted JSON documents is either an array or an object. The following code snippet is the JSON representation of the `person` code example previously outlined in the XML overview.

```
{
    "person": {
        "firstName": "Nathan",
        "lastName": "Jones",
        "email": {
            "emailAddress": "email@domain.com",
            "primary": true
        },
        "noContact": "email"
    }
}
```

The preceding example represents `person` as an object with the keys `firstName`, `lastName`, `email`, and `noContact`. In this code `email` is represented as a sub object to maintain visibility of the `primary` attribute. The JSON has been formatted for human-readability and may not appear to save much space. However, after stripping away all whitespace and newlines, the previous example becomes the following more compact payload that can be just as easily interpreted by a machine.

```
{"person":{"firstName":"Nathan","lastName":"Jones","email":{"
emailAddress":"email@domain.com","primary":true},"noContact":"email"}}
```

HTML

HTML is a markup language standard for structuring data on a web page so that your browser can interpret it. HTML was created in the early-1990s by Tim Berners-Lee as a means for co-workers at CERN to exchange documents. The first proposal for the HTML specification was published in mid-1993 and has since seen five major revisions. As of this writing, the fifth major revision, aptly dubbed HTML5, is in draft status as it works through the specification review and approval process.

HTML document structures are similar to XML documents; they both descend from SGML. However, the new draft version of HTML, HTML5, is not based on SGML like previous versions. HTML documents consist of a *doctype definition* (*DTD*), elements, attributes, data types, and character entity references. One key difference between HTML and XML document structures is that HTML documents have a predefined set of tag and attribute names. This section focuses on data structure and content, so data types and character entity references are not covered. There are a number of types defined in HTML and that number grows significantly as you include types for all predefined attributes such as ID, languages, color, and units of length, to name a few.

The doctype definition is the first line of an HTML document and enables the browser to know which version of the HTML specification the page is written in. HTML elements, although predefined, consist of a start-tag and end-tag (`<html>` and `</html>`) or an empty-element tag (`<br />`). Element attributes are key-value pairs that reside within the start-tag. Attributes are

predefined, such as `id`, `name`, and `class`, but HTML5 includes support for custom attributes. These attributes are prefixed with `data-` and should not contain uppercase characters. Custom attributes are intended to store application-specific data that does not fit an existing attribute. The following snippet is an example of an HTML document.

```
<!DOCTYPE html>
<html>
    <head>
      <title>Person: Nathan Jones</title>
    </head>
    <body>
        <div id='firstName'>Nathan</div>
        <div id='lastName'>Jones</div>
        <div id='emailAddress' data-primary='true'>
            email@domain.com
        </div>
        <div id='noContact' data-medium='email' />
    </body>
</html>
```

The preceding example contains an HTML representation of the person object used in the previous three code examples. The doctype definition instructs the browser that the document was written in HTML5. The root element `html` has two child elements: `head` and `body`. The `head` tag includes metadata about the page such as title, keywords, and page styles. The `body` tag includes the content that displays on the screen. The field names are defined using the `id` attribute, and the value is the content between the start-tags and end-tags. Note the use of HTML5 custom attributes to indicate primary e-mail and the no-contact medium.

Digesting Response Payloads

Web services return structured data in a number of formats, most commonly XML and JSON. It's also feasible that an application must retrieve HTML-structured data. Applications that implement these web services or retrieve HTML documents must interpret and transform that structured data into an object that is meaningful in the context of the application. Chapter 6, "Securing Network Traffic," discusses working with encrypted payloads.

This section covers parsing and transforming response data using native iOS APIs. To reinforce the concepts discussed in this chapter, the sample application you work through is a lightweight RSS reader. The application aggregates article content and Twitter information for articles posted to the CNN Top Stories RSS feed (`http://rss.cnn.com/rss/cnn_topstories.rss`).

> **WARNING** *Major websites often optimize and reorganize their site markup structure. For that reason, the wrox.com download for this chapter includes a copy of the RSS feed and linked articles from the day this chapter was developed.*

XML

Before walking through how to parse XML documents, you must understand the two parsing styles, *Simple API for XML* (*SAX*) and *Document Object Model* (*DOM*). SAX parsers are event-driven and step through elements in an XML document sequentially, processing each element individually. Alternatively, DOM parsers read the entire XML document into memory as a tree of nodes that can be traversed.

iOS ships with two native XML parsers, `NSXMLParser` and `libxml`. `NSXMLParser` is an Objective-C SAX parser that calls a variety of delegate methods as it encounters elements, attributes, CData blocks, comments and document start, and end events. `libxml` is an open source, C-based API that supports SAX and DOM parsing. `libxml` SAX parsing is similar to `NSXMLParser` in that it makes a number of callbacks as it encounters certain events. `libxml` DOM reads the entire XML document into a tree of nodes that can be traversed or queried using *XML Path Language* (*XPath*). The examples in this section use `NSXMLParser`, however, `libxml` is used in the next section on parsing HTML.

A number of third-party XML libraries are also available, most notably:

- TBXML (`https://github.com/71squared/TBXML`)
- TouchXML (`https://github.com/TouchCode/TouchXML`)
- KissXML (`https://github.com/robbiehanson/KissXML`)
- GDataXML (`http://code.google.com/p/gdata-objectivec-client/source/browse/trunk/Source/XMLSupport/`)

Each library, as well as any native XML parser, has benefits and drawbacks. For example, some libraries are more efficient for speed and memory consumption but lack the capability to create XML documents, and some are quick but consume larger portions of memory. Anticipated XML document size also plays into the decision of which parser to choose; some parsers do well with small documents, whereas others are intended to support large XML documents. Another factor to consider for native parsers is that they are delivered and supported by Apple. This means that they will be thoroughly tested with each future release of the iOS operating system to ensure backward compatibility. Each of these should be considered as you evaluate which parser to use in your application.

The purpose of the newsreader application is to display a list of articles with the capability to navigate to the full text. Like many media outlets, CNN does not publish story content in its RSS feed. That means fetching the entire article dataset requires two steps. First, you need to fetch the RSS feed that contains post-metadata, including links to the full stories, which enable you to build a shell for each post. Second, you need to fetch the actual story content and additional metadata from the articles HTML `meta` tags.

Before you can create your XML parser, you need to know what data you intend to capture and from where it will be retrieved. Listing 4-1 defines the interface for the `Post` object.

> **NOTE** *For clarity, the data source, whether it is a property that was fetched from the RSS feed or HTML story content, has been listed next to the property.*

LISTING 4-1: Post Object Interface Definition (/Application/topstories/topstories/Post.h)

```
@interface Post : NSObject

@property(nonatomic,strong) NSString        *title;             //rss
@property(nonatomic,strong) NSString        *postDescription;   //rss
@property(nonatomic,strong) NSString        *content;           //html
@property(nonatomic,strong) NSString        *author;            //html
@property(nonatomic,strong) NSString        *section;           //html
@property(nonatomic,strong) NSString        *contentURL;        //rss
@property(nonatomic,strong) NSDate          *pubDate;           //rss
@property(nonatomic,strong) NSMutableArray  *keywords;          //html
@property(nonatomic,strong) NSMutableArray  *tweets;
@property(nonatomic,assign) BOOL            contentFetched;
@property(nonatomic,assign) BOOL            tweetsLoading;

- (NSDictionary*)dictionaryRepresentation;

@end
```

With the `Post` object defined, you are ready to create the RSS parser. Listing 4-2 outlines the interface definition for the parser. Two things to note are the custom delegate that returns an array of the `Post` objects that were fetched and that the parser conforms to the `NSXMLParserDelegate`.

LISTING 4-2: Top Stories RSS Parser Interface Definition (/Application/topstories/topstories/TopStoriesParser.h)

```
#import "Post.h"

@protocol TopStoriesDelegate <NSObject>
@required
-(void)topStoriesParsedWithResult:(NSMutableArray*)posts;
@end

@interface TopStoriesParser : NSObject <NSXMLParserDelegate>

@property(nonatomic,strong) NSData                  *feedData;
@property(nonatomic,strong) NSMutableArray          *posts;

@property(assign)           id<TopStoriesDelegate>  delegate;

- (id)initWithFeedData:(NSData*)data;
- (void)parseTopStoriesFeed;

@end
```

Listing 4-3 covers the implementation of the RSS parser. `NSXMLParser` is a SAX parser; as such, it receives a number of delegate messages as certain parser events occur.

LISTING 4-3: Top Stories Parser Implementation (/Application/topstories/topstories/TopStoriesParser.m)

```
#import "TopStoriesParser.h"
#import "Utils.h"

@interface TopStoriesParser () {
    Post            *post;
    NSMutableString *currentValue;
    BOOL            parsingItem;
}

@end

@implementation TopStoriesParser

@synthesize posts = _posts;
@synthesize feedData = _feedData;
@synthesize delegate = _delegate;

- (id)initWithFeedData:(NSData*)data {
    self = [super init];
    if (self != nil) {
        self.feedData = data;
    }
    return self;
}

- (void)parseTopStoriesFeed {
    // create and start parser
    NSXMLParser *parser = [[NSXMLParser alloc]
                           initWithData:_feedData];
    parser.delegate = self;
    [parser parse];
}

#pragma mark - NSXMLParserDelegate
- (void)parserDidStartDocument:(NSXMLParser *)parser {
   _posts = [[NSMutableArray alloc] init];
}

- (void)parserDidEndDocument:(NSXMLParser *)parser {
    if ([_delegate respondsToSelector:
        @selector(topStoriesParsedWithResult:)]) {

        [_delegate topStoriesParsedWithResult:_posts];
    }
}

- (void)parser:(NSXMLParser *)parser
didStartElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName
    attributes:(NSDictionary *)attributeDict {

    // if you were expecting an attribute, it would be
```

```
    // handled here in the attributeDict by using
    // objectForKey: using the attribute name

    // started a new post, create a fresh object
    if ([elementName isEqualToString:@"item"]) {
        post = [[Post alloc] init];
        parsingItem = YES;
    }
}

- (void)parser:(NSXMLParser *)parser
foundCharacters:(NSString *)string {
    // capture the current element value
    NSString *tmpValue =
        [string stringByTrimmingCharactersInSet:
        [NSCharacterSet whitespaceAndNewlineCharacterSet]];
    if (currentValue == nil) {
        currentValue = [[NSMutableString alloc] initWithString:tmpValue];
    } else {
        [currentValue appendString:tmpValue];
    }
}

- (void)parser:(NSXMLParser *)parser
 didEndElement:(NSString *)elementName
  namespaceURI:(NSString *)namespaceURI
 qualifiedName:(NSString *)qName {

    // reached the end of a post
    if ([elementName isEqualToString:@"item"]) {
        [_posts addObject:post];
        post = nil;
        parsingItem = NO;
    }

    // make sure we're parsing a post item and not header data
    if (parsingItem == YES) {
        if ([elementName isEqualToString:@"title"]) {
            post.title = currentValue;

        } else if ([elementName isEqualToString:@"description"]) {
            post.postDescription = currentValue;

        } else if ([elementName isEqualToString:@"pubDate"]) {
            post.pubDate = [Utils publicationDateFromString:currentValue];

        } else if ([elementName isEqualToString:@"feedburner:origLink"]) {
            post.contentURL = currentValue;

        }
    }

    // reset the current element value
    currentValue = nil;
}
```

After the parser starts parsing the RSS feed, it calls the `parserDidStartDocument:` method at which point an `NSMutableArray` is initialized to store the processed `Post` objects. Likewise, when the parser reaches the end of the document, the `parserDidEndDocument:` method is called. At this point, the parser has a complete list of `Post` objects, so the parser informs its delegate.

The `parser:didStartElement:namespaceURI:qualifiedName:attributes:` method is called when a new element is started. This is where attributes are handled; remember, they are part of the start-tag. `parser:foundCharacters:` is called as content is read from the element and `parser:didEndElement:namespaceURI:qualifiedName:` is called when the element is closed. When elements are closed, it is safe to process and store any accumulated content.

Now that the parser is complete, it can be called as the application receives RSS feed data. Listing 4-4 details how to initialize the parser, begin the parsing process, and handle the delegate method. `FetchTopStoriesOperation` registers as the delegate for the RSS parser. When `topStoriesParsedWithResult:` is called, the operation iterates through each `Post` retrieved and issues a subsequent call to initiate the second step in the process, retrieving the story content.

LISTING 4-4: Fetch Top Stories Operation Implementation (/Application/topstories/topstories/FetchTopStoriesOperation.m)

```
#import "FetchTopStoriesOperation.h"
#import "FetchPostContentOperation.h"
#import "TopStoriesParser.h"

//#define kURL @"file:/<path to folder>/cnn_topstories.rss"
#define kURL @"http://rss.cnn.com/rss/cnn_topstories.rss"
#define kTimeout 30.0

@implementation FetchTopStoriesOperation

- (void)main {

    [self postNotification:kTopStoriesStartNotification];
    [self startNetworkActivityIndicator];

    // create the and issue request
    NSMutableURLRequest *req = [[NSMutableURLRequest alloc]
                                initWithURL:[NSURL URLWithString:kURL]
                                cachePolicy:NSURLCacheStorageAllowed
                                timeoutInterval:kTimeout];

    NSHTTPURLResponse *response = nil;
    NSError *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:req
                                        returningResponse:&response
                                                    error:&error];

    // check response got data and process data accordingly
    if (data != nil) {
        TopStoriesParser *parser = [[TopStoriesParser alloc]
                                    initWithFeedData:data];
        parser.delegate = self;
```

```
            [parser parseTopStoriesFeed];

        // there was an error getting the feed, alert the presses
        } else {
            [self postNotification:kTopStoriesErrorNotification];
        }

        [self stopNetworkActivityIndicator];
    }

    #pragma mark - TopStoriesDelegate
    - (void)topStoriesParsedWithResult:(NSMutableArray *)posts {
        // add the parsed results to the model
        [Model sharedModel].posts = posts;

        // trigger the content fetch (low priority)
        // handled here vs Model.m to adjust priority
        for (Post *post in posts){
            FetchPostContentOperation *op = [[FetchPostContentOperation alloc]
                                             init];
            op.post = post;
            op.queuePriority = NSOperationQueuePriorityLow;
            [op enqueueOperation];
        }

        [self postNotification:kTopStoriesSuccessNotification];
    }

    @end
```

Using the information gathered to this point, the application should resemble Figure 4-2. You can download the entire source from wrox.com for a fully implemented table view.

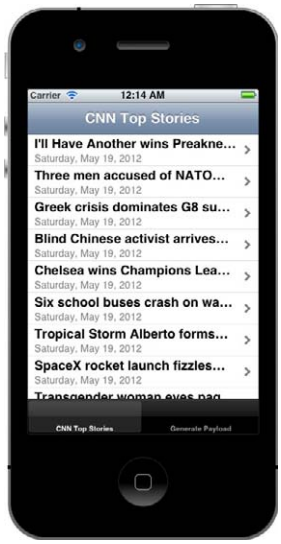


FIGURE 4-2

At this point the application has fetched and parsed the RSS feed and has initiated the story content download process. With that, it's time to parse the story content.

HTML

As discussed during the introduction, HTML documents share a similar structure with XML. However, XML document structures typically come with some sort of service contract between the sender and receiver. HTML documents typically do not have a contract associated with them; they can change frequently and drastically with little notice. One possible solution to help minimize the impacts of constantly changing HTML content to your application is to implement a remote façade, as discussed in Chapter 2. This web service enforces a strict content structure contract with the application. Changes to applications require App Store approval and that the user update the application, but changes to a web service under your control are much easier to deploy and more flexible in adapting to changes in the originating content.

> **WARNING** *Given the propensity for change in HTML documents, you should avoid parsing HTML within applications whenever possible. Changes could dramatically impact an application's ability to function properly.*

This section uses a `libxml` wrapper created by Ben Reeves (`https://github.com/zootreeves/Objective-C-HMTL-Parser`) to parse story content for each of the articles fetched from the RSS feed. To use this wrapper in your application you need to complete the following:

1. Add the `HTMLNode.h`/`.m` and `HTMLParser.h`/`.m` files to the project.
2. Add the `libxml2.dylib` to the project.
3. Add `$(SDKROOT)/usr/include/libxml2` to the Header Search Paths field in the targets' Build Settings.
4. Disable ARC for the `HTMLNode` and `HTMLParser` files. Within the target Build Phases, add the compiler flag `-fno-objc-arc` to `HTMLNode.m` and `HTMLParser.m`. As of this writing, this wrapper has not been converted for ARC support.

Listing 4-5 details how the application generates and processes the story content request. When the network call finishes, the custom method `processContentData:` is called. This method creates a new `HTMLParser` that processes the `head` and `body` tags, retrieving the pertinent meta data and article content. The logic to retrieve story content is particularly interesting because CNN uses a specific class to denote that a paragraph tag is part of the story.

LISTING 4-5: Fetch Story Content Implementation (/Application/topstories/topstories/FetchPostContentOperation.m)

```
#import "FetchPostContentOperation.h"
#import "HTMLParser.h"

#define kTimeout 30.0

@interface FetchPostContentOperation ()
- (void)processContentData:(NSData*)content;
@end

@implementation FetchPostContentOperation

@synthesize post = _post;

- (void)main {

    [self postNotification:kPostContentStartNotification];
    [self startNetworkActivityIndicator];

    NSURL *url = [NSURL URLWithString_post.contentURL];
    NSMutableURLRequest *req = [[NSMutableURLRequest alloc]
                                initWithURL:url
```

```
                                cachePolicy:NSURLCacheStorageAllowed
                                timeoutInterval:kTimeout];
    NSHTTPURLResponse *response = nil;
    NSError *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:req
                                         returningResponse:&response
                                                     error:&error];

    // check response got data and process data accordingly
    if (data != nil) {
        [self processContentData:data];
        _post.contentFetched = YES;

        [self postNotification:kPostContentSuccessNotification];

    // there was an error getting the post content, alert the presses
    } else {
        [self postNotification:kPostContentErrorNotification];

    }

    [self stopNetworkActivityIndicator];
}

#pragma mark - Private Methods
- (void)processContentData:(NSData*)content {

    NSError *error = nil;
    HTMLParser *parser = [[HTMLParser alloc]
                          initWithData:content error:&error];
    if (error) {
        return;
    }

    // get html doc head and meta tags
    HTMLNode *head = [parser head];
    NSArray *metaTags = [head findChildTags:@"meta"];

    // retrieve article meta data and add to the post
    for (HTMLNode *meta in metaTags) {
        NSString *name = [meta getAttributeNamed:@"name"];

        // keywords
        if ([name isEqualToString:@"keywords"]) {
            NSString *keywordContent = [meta getAttributeNamed:@"content"];
            NSMutableArray *keywords = (NSMutableArray*)
            [keywordContent
             componentsSeparatedByString:@","];

            if ([keywords count]>0) {
                _post.keywords = keywords;
            }

        // author name
        } else if ([name isEqualToString:@"author"]) {
```

continues

LISTING 4-5 *(continued)*

```
            NSString *author = [meta getAttributeNamed:@"content"];
            if (author.length > 0) {
                _post.author = author;
            }

        // article section
        } else if ([name isEqualToString:@"section"]) {
            NSString *section = [meta getAttributeNamed:@"content"];
            if (section.length > 0) {
                _post.section = section;
            }
        }
    }

    // get html doc body and paragraph tags
    HTMLNode *body = [parser body];
    NSArray *paragraphTags = [body findChildTags:@"p"];

    // iterate through all paragraphs saving the appropriate story content
    NSMutableString *storyContent = [[NSMutableString alloc] init];
    for (HTMLNode *para in paragraphTags) {

        // only save the 'story paragraphs' - class=cnn_storypgraphtxt
        NSString *class = [para getAttributeNamed:@"class"];
        NSRange storyParaTest = [[class lowercaseString]
                                 rangeOfString:@"cnn_storypgraphtxt"];
        if ((storyParaTest.location != NSNotFound) && (class != nil)) {
            [storyContent appendString:[para rawContents]];
        }
    }

    _post.content = storyContent;
}

@end
```

As you can see, parsing HTML can be extremely fragile. There are a number of points throughout this process that could break, each of which could render the application unusable. A simple change to the class name for story content would cause the application not to receive any story content and display a blank view.

However, if all goes well the story content would be retrieved and you could drill into an individual article to read the entire story. Although it is a simple layout, Figure 4-3 shows what the story content view may resemble. The application shown in this figure has now fetched all CNN-related content.

The next feature is to add light Twitter search integration to search for story keywords retrieved from the `meta` tags.

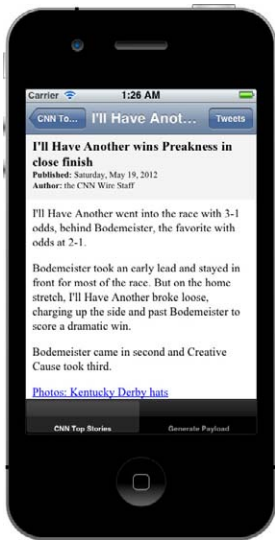


FIGURE 4-3

JSON

As of iOS 5 Apple provides native JSON parsing support via the `NSJSONSerialization` class. Parsing JSON data prior to iOS 5's native support required the use of a third-party library such as JSON framework (`https://github.com/stig/json-framework`) or JSON-Kit (`https://github.com/johnezang/JSONKit`). Although these libraries were well supported and were simple to use, the native, Apple-supported API was a welcome addition.

`NSJSONSerialization` provides two methods for parsing JSON data: `JSONObjectWithData:options:error:` and `JSONObjectWithStream:options:error`. `JSONObjectWithData:options:error:` creates a Foundation object from the JSON data passed. `JSONObjectWithStream:options:error:` behaves similarly to `JSONObjectWithData:options:error:` with the exception that it accepts a JSON data feed as its source. Both methods have an `options` parameter that accepts any combination of the following values to configure how the parser interprets input.

- `NSJSONReadingAllowFragments`: Instructs the parser to enable top-level objects that are neither an `NSArray` nor `NSDictionary`. This option can be used to allow simple JSON structures such as `{"user": null}` to be successfully converted.
- `NSJSONReadingMutableContainers`: Instructs the parser to generate `NSMutableArray` and `NSMutableDictionary` objects. Mutable objects mean that you can modify them using methods, such as `addObject:` for `NSMutableArray` and `setObject:forKey:` for `NSMutableDictionary`. This could be used in situations with a primary and secondary result set in which you need to add a value from the secondary result to the primary result before further processing.
- `NSJSONReadingMutableLeaves`: Instructs the parser to generate `NSMutableString` objects. You could use this option if you need to manipulate a particular field value within the parsed response prior to performing additional processing.

One feature of the sample application is that it can fetch related tweets based on the stories keywords. This is accomplished by using Twitter's search API (`http://search.twitter.com/search.json?q=<query>`), which returns JSON-encoded search results. Listing 4-6 outlines the interface definition for the `Tweet` object. The `Tweet` object uses a small subset of fields returned from Twitter. Each `Post` maintains an array of related tweets as they are retrieved.

LISTING 4-6: Tweet Object Interface Definition (/Application/topstories/topstories/Tweet.h)

```
@interface Tweet : NSObject

@property(nonatomic,strong) NSString *identifier;
@property(nonatomic,strong) NSString *fromUser;
@property(nonatomic,strong) NSString *fromUserDisplay;
@property(nonatomic,strong) NSString *profileImageURL;
@property(nonatomic,strong) NSString *text;
@property(nonatomic,strong) NSDate   *createdDate;
@property(nonatomic,strong) UIImage  *profileImage;

- (id)initWithDictionary:(NSDictionary*)tweetData;

@end
```

> **NOTE** *Just after beginning development on this sample application, CNN stopped populating the keywords* `meta` *tag in its stories. Given the change, the author created the keyword meta data in the story files available on wrox.com. Although this change did not materially impact the application, it underscores the potential risks associated with parsing HTML content in applications.*

With the `Tweet` object defined, the application can search Twitter and begin parsing the JSON response. Listing 4-7 details how to create the search request and parse the resulting response. When the response is received, the operation calls `JSONObjectWithData:options:error:` to create an `NSDictionary` representation of the search results. The operation then iterates through the tweet data creating `Tweet` objects for each result and adding it to the `Post`.

LISTING 4-7: Retrieve Related Tweets (/Application/topstories/topstories/FetchPostTweetsOperation.m)

```
#define kTimeout 30.0

@implementation FetchPostTweetsOperation

@synthesize post = _post;

- (void)main {

    [self postNotification:kTweetsStartNotification];
    [self startNetworkActivityIndicator];
    _post.tweetsLoading = YES;

    // create the twitter query
    NSMutableString *query = [[NSMutableString alloc] init];
    for (int i=0; i<[_post.keywords count]; i++) {
        // prepend comma for all but first keyword
        if (i != 0) {
            [query appendString:@","];
        }

        [query appendString:[_post.keywords objectAtIndex:i]];
    }

    // create the and issue request - separate variables for line size
    NSString *searchEndpoint = @"http://search.twitter.com/search.json";
    NSString *querystring = [NSString
                             stringWithFormat:@"q=%@&rpp=15",
                             [Utils urlEncode:query]];

    NSString *url = [NSString stringWithFormat:@"%@?%@",
                     searchEndpoint,
```

```
                    querystring];
    NSMutableURLRequest *req = [[NSMutableURLRequest alloc]
                                initWithURL:[NSURL URLWithString:url]
                                cachePolicy:NSURLCacheStorageAllowed
                                timeoutInterval:kTimeout];

    NSHTTPURLResponse *response = nil;
    NSError *error = nil;
    NSData *data = [NSURLConnection sendSynchronousRequest:req
                                        returningResponse:&response
                                                    error:&error];

    // check response got data and process data accordingly
    if (data != nil) {

        // convert response to JSON
        NSError *error = nil;
        NSDictionary *searchResults = [NSJSONSerialization
                                       JSONObjectWithData:data
                                       options:NSJSONReadingAllowFragments
                                       error:&error];

        // create tweets
        NSMutableArray *tweets = [[NSMutableArray alloc] init];
        NSArray *results = [searchResults objectForKey:@"results"];
        for (NSDictionary *tweetData in results) {
            Tweet *tweet = [[Tweet alloc] initWithDictionary:tweetData];
            [tweets addObject:tweet];
        }

        _post.tweetsLoading = NO;
        if ([tweets count] > 0) {
            _post.tweets = tweets;
            [self postNotification:kTweetsSuccessNotification];

        // no tweets were retrieved
        } else {
            [self postNotification:kTweetsErrorNotification];
        }

    // there was an error getting the post content, alert the presses
    } else {
        _post.tweetsLoading = NO;
        [self postNotification:kTweetsErrorNotification];

    }

    [self stopNetworkActivityIndicator];
}

@end
```

When the `TweetsTableViewController` has been hooked up (see the wrox.com downloads for this chapter) and the operation finishes, you should see a related tweet view similar to Figure 4-4.

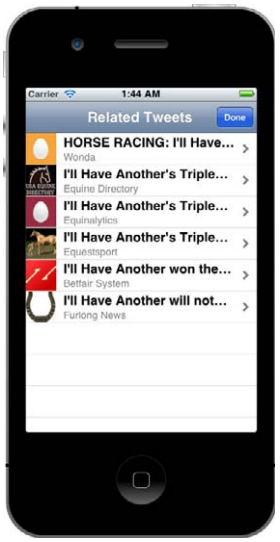


FIGURE 4-4

Generating Request Payloads

Integrating sophisticated web services into your applications often requires you to transmit payloads in a structured format. These formats are most typically JSON or XML, or some variation of XML such as SOAP. This section covers how to create each of these common exchange formats from the Foundation objects used within your applications. This section builds on the newsreader sample application by adding the ability to transmit the aggregated story content to a simple, server-side script for evaluation. The script is included with the wrox.com downloads for this chapter.

JSON

Generating JSON data is as easy as parsing it. Apple released `NSJSONSerialization` with iOS 5, which provides a native API for creating JSON data from Foundation objects. `NSJSONSerialization` exposes two methods to create JSON data, `dataWithJSONObject:options:error:` and `writeJSONObject:toStream:options:error:`. Each method contains an `options` parameter to configure the output of the method. As of this writing there is only a single option, `NSJSONWritingPrettyPrinted`, which instructs the method to generate JSON that is more readable by adding whitespace. Not specifying this option generates the most compact JSON possible.

`NSJSONSerialization` also provides `isValidJSONObject:` to validate whether the Foundation object you attempt to convert is convertible. For objects to be converted to JSON, they must conform to the following rules:

- Top-level object that is an `NSArray` or `NSDictionary`.
- All objects must be `NSString`, `NSNumber`, `NSArray`, `NSDictionary`, or `NSNull`.
- All `NSDictionary` keys must be `NSStrings`.
- `NSNumbers` must not be NaN or infinity.

Because the newsreader application stores articles in the form of a custom class, `Post`, a little legwork is required to use `NSJSONSerialization`. This additional legwork is required because the `Post` class does not meet the conversion rules mentioned. One approach that enables the application to convert articles into JSON is to implement a method on `Post` that returns an `NSDictionary`, as shown in Listing 4-8. Only a subset of the class's properties is included in an attempt to minimize the example payload size.

LISTING 4-8: Generating a Post NSDictionary Representation (/Application/topstories/topstories/Post.m)

```
...
- (NSDictionary*)dictionaryRepresentation {
    NSString *pubDateString =
    [NSString stringWithFormat:@"%@", self.pubDate];

    // content, tweets, and keywords were left off to limit size
    return [NSDictionary dictionaryWithObjectsAndKeys:
            [Utils urlEncode:self.title], @"title",
            [Utils urlEncode:self.postDescription], @"description",
            [Utils urlEncode:self.author], @"author",
            [Utils urlEncode:self.section], @"section",
            [Utils urlEncode:self.contentURL], @"contentURL",
            [Utils urlEncode:pubDateString], @"pubDate", nil];
}
```

Now that each `Post` has an `NSDictionary` representation, they can be converted to JSON and transmitted to the server-side script, as shown in Listing 4-9.

LISTING 4-9: JSON Generation and Transmission (/Application/topstories/topstories/ShareArticlesOperationJSON.m)

```
#import "ShareArticlesOperationJSON.h"

@implementation ShareArticlesOperationJSON

@synthesize posts, shareType;

- (void)main {

    [self startNetworkActivityIndicator];

    // create url and issue request
    NSString *urlString =
    [NSString
     stringWithFormat:@"<server>/parse.php?parseMethod=%d",shareType];
    NSURL *url = [NSURL URLWithString:urlString];

    NSMutableURLRequest *req =
    [[NSMutableURLRequest alloc] initWithURL:url
                                 cachePolicy:NSURLCacheStorageAllowed
                             timeoutInterval:30.0];

    [req setHTTPMethod:@"POST"];
    [req setValue:@"application/json" forHTTPHeaderField:@"Accept"];

    // convert array of POST objects to array of dictionary
    // objects so NSJSONSerialization can handle it
    NSMutableArray *articles = [[NSMutableArray alloc] init];
    for (Post *post in posts) {
```

continues

LISTING 4-9 *(continued)*

```
    // dictionaryRepresentation is a custom method
    // that creates an NSDictionary of a few key fields
    [articles addObject:post.dictionaryRepresentation];
}
NSDictionary *articleData =
[NSDictionary dictionaryWithObject:articles forKey:@"articles"];

// validate object
if ([NSJSONSerialization isValidJSONObject:articleData] == NO) {
    [self postNotification:kShareArticleErrorNotification];
    [self stopNetworkActivityIndicator];
    return;
}

// convert dictionary to JSON data and set the body
NSError *jsonWriteError = nil;
NSData *payload =
[NSJSONSerialization dataWithJSONObject:articleData
                                options:NSJSONWritingPrettyPrinted
                                  error:&jsonWriteError];

[req setHTTPBody:payload];


NSHTTPURLResponse *response = nil;
NSError *error = nil;
NSData *data = [NSURLConnection sendSynchronousRequest:req
                                     returningResponse:&response
                                                 error:&error];

// check response got data and process data accordingly
// you would also typically check the status code here too
if (data != nil) {
    NSError *jsonParseError = nil;
    NSDictionary *responseDict =
    [NSJSONSerialization JSONObjectWithData:data
                                    options:0
                                      error:&jsonParseError];

    // successfully transmitted articles
    if ([responseDict objectForKey:@"articleCount"] != nil) {
        [self postNotification:kShareArticleStartNotification];

        // tell the user how many articles were sent
        NSInteger articleCount =
            [[responseDict objectForKey:@"articleCount"] intValue];

        NSString *msg =
            [NSString stringWithFormat:@"%d articles shared via JSON.",
             articleCount];

        dispatch_async(dispatch_get_main_queue(), ^{
            [[[UIAlertView alloc] initWithTitle:@"Great Success"
```

```
                                                    message:msg
                                                   delegate:nil
                                          cancelButtonTitle:@"OK"
                                          otherButtonTitles:nil] show];
                });

            // server was not able to properly interpret the content
            } else {
                [self postNotification:kShareArticleErrorNotification];
            }

        }

        [self stopNetworkActivityIndicator];
    }
```

The application creates an array of `NSDictionary` objects (representing each `Post`) and then creates an `NSDictionary` with one key-value pair of `articles`. This is done so that the top-level of the resulting JSON is a collection.

Attempting to convert invalid types to JSON causes your applications to crash. Therefore, it is a best practice to call `isValidJSONObject:` prior to calling `dataWithJSONObject:options:error:` to handle potential errors gracefully.

After executing `ShareArticlesOperationJSON` the user should see an alert that resembles Figure 4-5.

FIGURE 4-5

XML

There are a number of approaches to create XML documents, including string formatting, using one of the third-party libraries that supports writing XML mentioned earlier in the previous "XML" section, and `libxml`. `libxml` is the only native API for writing XML that ships with iOS. `libxml` is a C-based API, which means that it may be cumbersome for users that have not worked with C functions before.

The examples in this section use `libxml`, specifically the `xmlwriter` interface. The approach you ultimately choose depends on your specific requirements. If you are in a situation in which all the service communication is done via JSON except for a single, third-party call that requires a two-field XML document, using `libxml` may be too much overhead. However, more advanced requirements may benefit from the use of a third-party library or even a more flexible wrapper around `libxml` than is presented in this section.

Building on the steps outlined in the XML parsing section earlier in this chapter, one additional step is required to use `libxml` to create XML documents; `<libxml/xmlwriter.h>` must be imported into each class creating XML. Within the newsreader sample application, XML generation for articles has been consolidated to the `postXMLDataFromDictionary:` method in `Utils`, as shown in Listing 4-10.

LISTING 4-10: Article XML Generation (/Application/topstories/topstories/Utils.m)

```
#import "Utils.h"
#import <libxml/xmlwriter.h>

@implementation Utils
...
+ (NSData*)postXMLDataFromDictionary:(NSDictionary*)dictionary {

    xmlTextWriterPtr    _writer;
    xmlBufferPtr        _buffer;
    xmlChar             *_elementName;
    xmlChar             *_elementValue;

    _buffer = xmlBufferCreate();
    _writer = xmlNewTextWriterMemory(_buffer, 0);

    xmlTextWriterStartDocument(_writer, "1.0", "UTF-8", NULL);
    xmlTextWriterStartElement(_writer, BAD_CAST "articles");

    NSArray *posts = [dictionary objectForKey:@"articles"];
    for (NSDictionary *post in posts) {

        // start the post element
        xmlTextWriterStartElement(_writer, BAD_CAST "article");

        // create elements for each post property
        NSArray *keys = [post allKeys];
        for (NSString *key in keys) {
            // you could optionally check class types here
            // and do additional processing, however, the
            // types being processed can be cast as xmlChar*

            // xmlChar pointer to element name and value
            _elementName = (xmlChar*)[key UTF8String];
            _elementValue = (xmlChar*)[[post objectForKey:key]
                                       UTF8String];

            // write the element
            xmlTextWriterStartElement(_writer, _elementName);
            xmlTextWriterWriteString(_writer, _elementValue);
            xmlTextWriterEndElement(_writer); // </_elementName>
        }

        xmlTextWriterEndElement(_writer); // </article>
    }

    xmlTextWriterEndElement(_writer); // </articles>
    xmlTextWriterEndDocument(_writer);
    xmlFreeTextWriter(_writer);

    // convert buffer to NSData and cleanup
    NSData *_xmlData = [NSData dataWithBytes:(_buffer->content)
```

```
                                        length:(_buffer->use)];
    xmlBufferFree(_buffer);

    return _xmlData;
}

@end
```

Calling `xmlTextWriterStartDocument()` adds the XML version and encoding definition to the document. When that is added you can begin calling `xmlTextWriterStartElement()`, `xmlTextWriterWriteString()`, and `xmlTextWriterEndElement()` as needed to create your XML structure. `xmlTextWriterEndElement()` does not require the element name to close; it keeps track of that for you.

Two important function sets of `libxml` not required by this example are the ability to add comments and element attributes to the XML document. Similar to elements, comments can be added by calling the `xmlTextWriterStartComment()`, `xmlTextWriterWriteComment()`, and `xmlTextWriterEndComment()` series of functions. Attributes are handled differently; there is a convenience function, `xmlTextWriterWriteAttribute()`, that accepts the attribute name and value and handles the start and end portion of the process for you.

When the method iterates through all the articles, it calls `xmlTextWriterEndElement()` one final time to close the parent element (`articles` in this case) and then calls `xmlTextWriterEndDocument()` to complete the process. When complete, the application converts the XML buffer to `NSData` so that it can be transmitted in the requests post body.

Now that the application can generate the necessary XML, Listing 4-11 outlines how to transmit that data to the server.

LISTING 4-11: XML Request Creation and Transmission (/Application/topstories/topstories/ShareArticlesOperationXML.m)

```
#import "ShareArticlesOperationXML.h"

@implementation ShareArticlesOperationXML

@synthesize posts, shareType;

- (void)main {

    [self startNetworkActivityIndicator];

    // create the and issue request
    ...

    // convert array of POST objects to array of dictionary
    // objects so the XML writer can handle it
    NSMutableArray *articles = [[NSMutableArray alloc] init];
    for (Post *post in posts) {
        [articles addObject:post.dictionaryRepresentation];
    }
    NSDictionary *articleData =
```

continues

LISTING 4-11 *(continued)*

```
        [NSDictionary dictionaryWithObject:articles forKey:@"articles"];

        // convert dictionary to XML data and set body
        NSData *payload = [Utils postXMLDataFromDictionary:articleData];
        [req setHTTPBody:payload];

        // issue network request
        NSHTTPURLResponse *response = nil;
        NSError *error = nil;
        NSData *data = [NSURLConnection sendSynchronousRequest:req
                                             returningResponse:&response
                                                         error:&error];

        // check response got data and process data accordingly
        // you would also typically check the status code here too
        if (data != nil) {
            ...
        }

        [self stopNetworkActivityIndicator];
    }

    @end
```

A significant portion of Listing 4-11 should look familiar; it resembles Listing 4-9. It has been condensed for brevity but follows a similar process. To simplify the XML generation logic, the array of `Post` objects is converted to an `NSDictionary` following the same steps defined in Listing 4-9. After the XML document has been generated, it is transmitted to the server.

After executing `ShareArticlesOperationXML` the user should see an alert Figure 4-6.

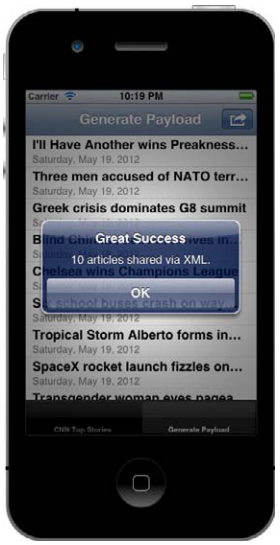


FIGURE 4-6

SUMMARY

There are several factors to consider when designing how your application communicates with web services. When implemented properly, the best architecture to deliver data to the mobile channel is REST. The optimal data interchange format for iOS applications is JSON. Although XML is also natively supported, JSON payloads are easier to work with, map more appropriately to Foundation types, and are more cellular network-friendly.

In the next chapter, you gain an understanding of where errors occur during the networking communication process and how to handle them gracefully.

5

Handling Errors

WHAT'S IN THIS CHAPTER?

- Sources of networking errors in iOS applications
- Detecting reachability of the network
- Rules of thumb for handling errors
- A design pattern for handling network errors

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code is in the Chapter 5 download and individually named according to the names throughout the chapter.

Thus far, you have learned about networking iPhones with other systems under the implied assumption that things just work. In this chapter you discard that assumption and dive into the real world of networking. In this real world, things go wrong, sometimes extremely wrong: Phones move on and off networks; packets get lost or delayed; network infrastructure fails; and there is the occasional user error. Writing a networked iOS app would be much easier if things just worked, but unfortunately that isn't the case. This chapter reviews some of the things that can cause networked operations to fail. It discusses how the system informs the app of the failure, and how the app should gracefully inform the user. A software pattern that is helpful in handling errors in a clean and consistent manner without requiring error handling code in the application logic is also described.

UNDERSTANDING ERROR SOURCES

In the early days of iOS there was a weather app from a reputable source. It worked well on Wi-Fi or a clean cellular network, but if the network was anything less than perfect, this

weather app seemed to catch a cold and crash to the Home screen. Dozens of other apps respond poorly to network errors with a blizzard of UIAlertViews frantically informing the user that there is a "404 Error on Server X" or something similar. Other apps have interfaces that become unresponsive if the network is slow. Each of these is an example of poor understanding of network failure modes and anticipation of possible network degradation or failures. If you want to avoid these kinds of errors and adequately handle networking errors, you must first understand their origin.

When you consider what a byte must do to get from a device to a remote server and back, all within a couple of hundred milliseconds, it is a miracle that networked devices work at all. The complexity of device networking and internetworking led to the development of layered networks. *Layered networks* divide this complex environment into more manageable chunks. Although this helps the programmer, networking errors like those mentioned previously can occur as data moves between layers. Figure 5-1 illustrates the layering in the Internet Protocol stack.

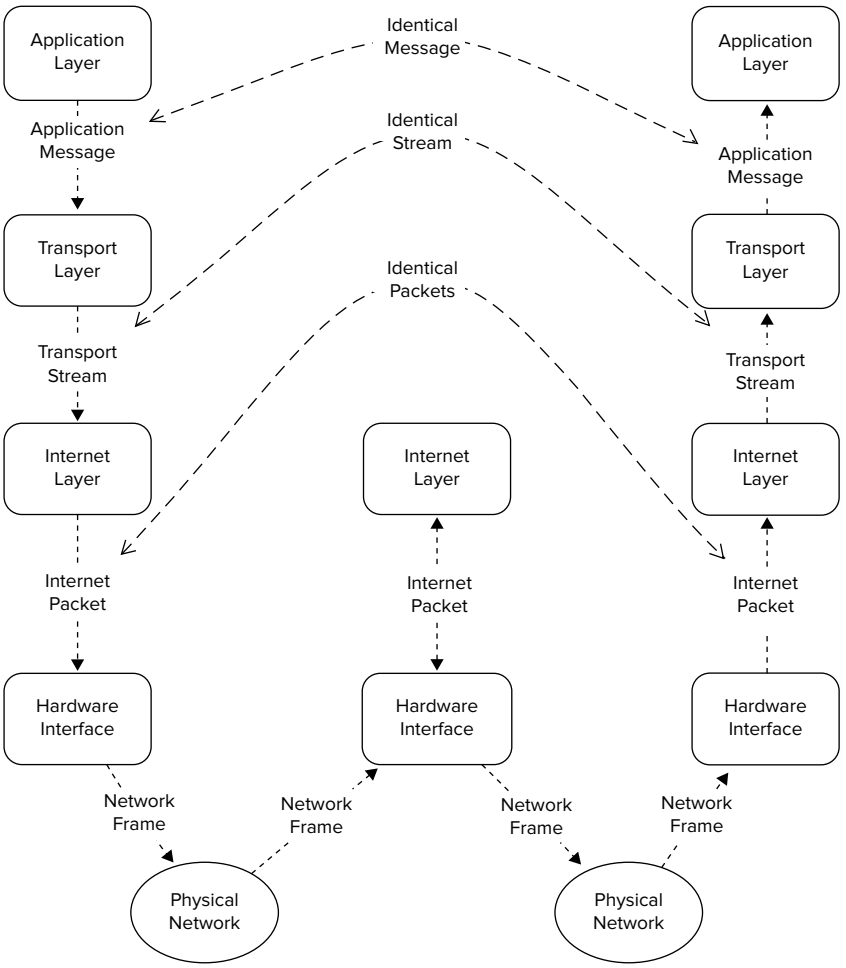


FIGURE 5-1

Every layer performs some sort of error checking, which could be mathematical, logical, or something else altogether. For example, when the network interface layer receives a frame, it first validates the contents against the error correcting code, and if it doesn't match, an error occurs. If the frame never even arrives, a timeout or connection reset may occur. Error checks occur at every step up the stack on the way to the application layer, where the message is checked both syntactically and semantically.

Although there is an almost infinite number of ways that a connection between a phone and a server may fail, when using the URL loading system in iOS, you can group these causes into three categories of errors: operating system errors, HTTP errors, and application errors. These categories of errors correlate to steps in the sequence of operations necessary to make an HTTP request. Figure 5-2 is a simplified sequence diagram of an HTTP request to an application server to provide some data from an enterprise network. Each of the shaded zones represents an error domain for each of the three types of errors. Typically, operating system errors are caused by problems reaching the HTTP server. HTTP errors are caused by problems within the HTTP server or application server. Application errors are caused by problems with either the data delivered in the request or with any other systems the application server queries.

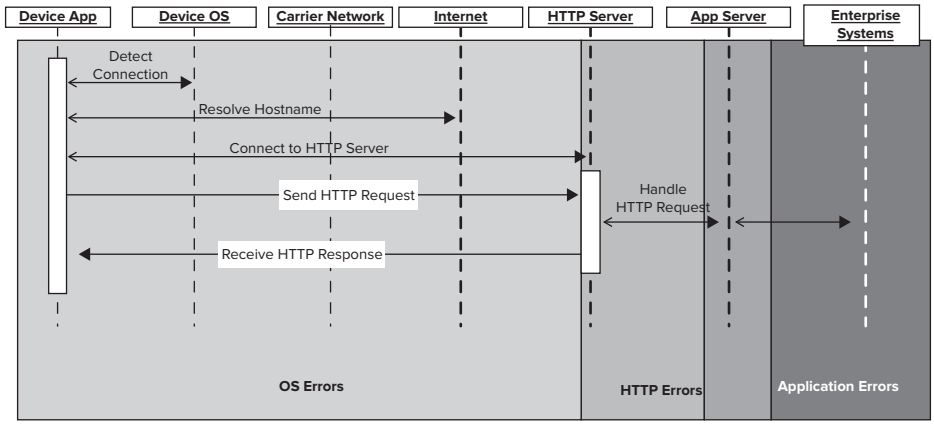


FIGURE 5-2

The steps in the sequence become much more complicated if the request is a secure HTTPS request or if the HTTP server redirects the client. For many of these steps, there are numerous substeps, such as the sequence of SYN and SYN-ACK packets involved in establishing a TCP connection. The following sections describe each error category in greater detail.

Operating System Errors

Operating system (OS) errors are errors caused by a data packet not reaching its intended target. That data packet may be part of establishing a connection or occur somewhere in the middle of the connection. OS errors can be caused by several conditions:

- **Lack of a network —** If the device doesn't have a data network connection, the connection attempt is rejected quickly or fails midstream. These types of errors can be anticipated using the Reachability framework provided by Apple, which is covered later in this section.

- **Inability to route to the intended host —** The device may have a network connection, but the intended target of the connection may be on a segregated network or offline. These errors can sometimes be detected quickly by the operating system but could potentially result in a connection timeout.
- **No application listening on the target port —** After the request arrives at the target host, the packet is delivered to the port number specified in the request. If no server is listening on that port or if too many connection requests are queued, then the connection request will be rejected.
- **Inability to resolve the target hostname —** If the name of the target host cannot be resolved, then the URL loading system returns an error. Often these errors can be due to configuration mistakes or an attempt to access a host on a segregated network with no external name resolution.

In the URL loading system of iOS, operating system errors are reported to the application in an `NSError` object. iOS uses `NSError` to communicate errors between software components. The key benefit of `NSError` when compared to a simple error code is that `NSError` objects contain an error domain property.

The use of `NSError` objects is not limited to the operating system though. Your app can create its own `NSError` objects and use them to propagate error messages around the app. The following snippet illustrates an application method that uses `NSError` to communicate a failure back to the calling view controller.

```
- (id)fetchMyStuff:(NSURL*)url error:(NSError**)error
{
    BOOL errorOccurred = NO;

    // some code that makes a call and may fail

    if (errorOccurred) //some kind of error
    {
        NSMutableDictionary *errorDict = [NSMutableDictionary dictionary];
        [errorDict setValue:@"Failed to fetch my stuff"
                     forKey:NSLocalizedDescriptionKey];
        *error = [NSError errorWithDomain:@"myDomain"
                                     code:kSomeErrorCode
                                 userInfo:errorDict];
        return nil;
    } else {
       return stuff
    }

}
```

The domain property segregates error numbers based on the library or framework that produced them. Using domains, framework developers do not need to worry about overlapping error codes because the domain property defines which framework generated the error. For example, frameworks A and B can both have an error code 1, but the two are distinguished by the unique domain values provided by each framework. Consequently, if your code needs to distinguish between unique `NSError` values, it must compare both the code and domain properties of the `NSError` object.

An NSError object has three primary properties:

- **Code —** An NSInteger value that indicates which error occurred. This number is unique to the error domain that instantiates the error.
- **Domain —** An NSString pointer that specifies the domain of the error. Example domains include NSPOSIXErrorDomain, NSOSStatusErrorDomain, and NSMachErrorDomain.
- **User Info —** An NSDictionary pointer containing values specific to the error that occurred.

Many of the errors that occur within the URL loading system come from the NSURLErrorDomain domain, and the code values are frequently drawn from the error codes defined in CFNetworkErrors.h. As with any constant value provided by iOS, your code should rely on the defined constant name for the error, not the actual error code value. For example, the error code if the client cannot connect to the host is -1004 with a defined constant of kCFURLErrorCannotConnectToHost. Your code should never directly reference -1004 because this value may change in future revisions of the OS; instead it should use the kCFURLError provided enumeration name.

The following code example illustrates making an HTTP request using the URL loading system.

```
NSHTTPURLResponse *response=nil;
NSError *error=nil;
NSData *myData=[NSURLConnection sendSynchronousRequest:request
                                     returningResponse:&response
                                                 error:&error];

if (!error) {
// No OS Errors, keep going in the process
    ...
} else {
    // Something low level broke
}
```

Notice that the NSError object is declared as a pointer to nil. The NSURLConnection object instantiates only the NSError object if an error occurs. The URL loading system owns the NSError object; you should retain the object if your code will need it later. If the NSError pointer still points to nil after the synchronous request completes, then no low-level OS error occurred. At this point your code knows that no OS level has occurred, but an error may have occurred at a higher layer in the protocol stack.

If your application makes an asynchronous request, the NSError object is returned to the delegate class on the method with the following signature:

```
- (void)connection:(NSURLConnection *)connection
  didFailWithError:(NSError *)error
```

This message is the final message delivered to the delegate for the request, and the delegate must discern the cause of the error and react appropriately. In the following example, the delegate displays a UIAlertView to the user:

```
- (void) connection:conn didFailWithError:error {
    UIAlertView *alert = [UIAlertView alloc] initWithTitle:@"Network Error"
```

```
                                                    message:[error description]
                                                   delegate:self
                                          cancelButtonTitle:@"Oh Well"
                                          otherButtonTitles:nil];
        [alert show];
        [alert release];
    }
```

This code reports errors to the user but in an abrupt and unfriendly way. In the iOS Human Interface Guidelines (HiG), Apple recommends against overuse of `UIAlertViews` because it breaks the illusion of a magical device. The Gracefully Handling Network Errors section reviews a pattern for handling errors cleanly and consistently with a pleasing user interface.

Another major cause of communication errors from iOS devices is the inability of the device to reach its target server because it lacks a network connection. You can avoid many OS errors by first checking the network status before attempting a network request. Keep in mind that these devices can rapidly move on and off the network; therefore, it is reasonable to check the network reachability before each call.

iOS provides many ways to determine the status of a device's network connection as part of the `SystemConfiguration` framework. You can find details on the low-level API in `SCNetworkReachability` reference documentation. The API is powerful but also somewhat cryptic. Thankfully, Apple provides an example program called Reachability that implements a simplified, high-level wrapper around `SCNetworkReachability`. Reachability is available in the iOS Developer Library.

The Reachability wrapper provides four major pieces of functionality:

- An indication of whether or not the device has a functional network connection
- An indication of whether a specific host can be reached with the current network connections
- An indication about which networking technology is being used: Wi-Fi, WWAN, or none
- Notifications of any changes in the network state

To use the Reachability API, download the example program from the iOS developer library at `http://developer.apple.com/library/ios/#samplecode/Reachability/Introduction/Intro.html`, and add `Reachability.h` and `Reachability.m` to your app's XCode project. In addition, you need to add the `SystemConfiguration` framework to your XCode project. Adding the `SystemConfiguration` framework to your XCode project entails editing your project configuration. Figure 5-3 shows the steps required to add the SystemConfiguration framework to your XCode project.

After selecting the project target, scroll-down the settings to the Linked Frameworks and Libraries section and press the plus sign to add a framework. The framework selector then appears. Select the `SystemConfiguration` framework and press the add button to add it to your project.

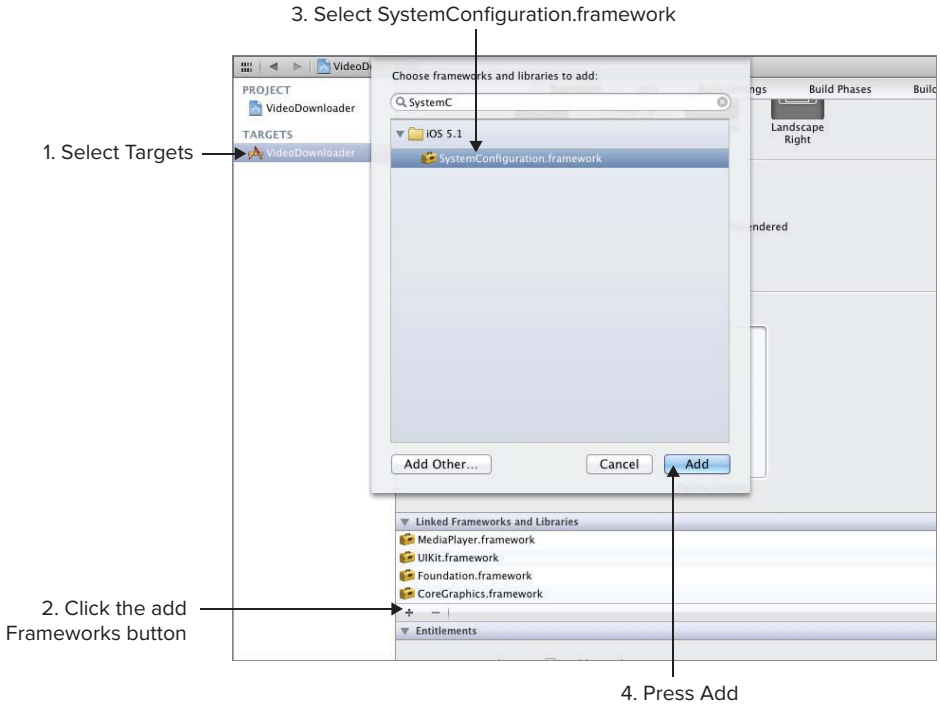


FIGURE 5-3

The following code snippet checks to see if a network connection is available. It does not guarantee that any particular host or IP address is reachable; it just indicates that a network connection exists.

```
#import "Reachability.h"
...
if ([[Reachability reachabilityForInternetConnection]
                          currentReachabilityStatus] == NotReachable) {
    // handle the lack of a network
}
```

In some situations you may want to change certain actions, disable UI elements, or change timeout values if the device is on a limited network. If your application needs to know the connection type it is currently using, use the following code:

```
#import "Reachability.h"
...
NetworkStatus reach = [[Reachability reachabilityForInternetConnection]
                                                 currentReachabilityStatus];
 if (reach == ReachableViaWWAN) {
    // Network Is reachable via WWAN (aka. carrier network)
} else if (reach == ReachableViaWiFi) {
    // Network is reachable via WiFi
}
```

It is also useful to know if the reachability status of the device changes so that you can modify application behavior proactively. The following code snippet initiates monitoring of network status:

```
#import "Reachability.h"
...
[[NSNotificationCenter defaultCenter]
                                    addObserver:self
                                       selector:@selector(networkChanged:)
                                           name:kReachabilityChangedNotification
                                         object:nil];
Reachability *reachability;
reachability = [[Reachability reachabilityForInternetConnection] retain];
[reachability startNotifier];
```

This code registers the current object as an observer for notifications with the name `kReachabilityChangedNotification`. `NSNotificationCenter` calls the method named `networkChanged:` on the current object. It passes into that object an `NSNotification` with the new reachability status when it changes. The following example demonstrates the notification listener:

```
- (void) networkChanged: (NSNotification* )notification
{
    Reachability* reachability = [notification object];
    if (reachability == ReachableViaWWAN) {
        // Network Is reachable via WWAN (a.k.a. carrier network)
    } else if (reachability == ReachableViaWiFi) {
        // Network is reachable via WiFi
    } else if (reachability == NotReachable) {
        // No Network available
    }
}
```

Reachability can also determine if a specific host is reachable on the current network. You can use this feature to alter the behavior of an enterprise app based on whether the app is on an internal segregated network or on the open Internet. The following code sample illustrates this feature:

```
Reachability *reach = [Reachability
                reachabilityWithHostName:@"www.captechconsulting.com"];
if (reachability == NotReachable) {
    // The target host is not reachable available
}
```

Keep in mind that this feature requires a round trip to the target host. If used for each request, it can add significant network overhead and latency to the application. Apple recommends that host reachability detection not be performed on the main thread. There is a possibility that the attempt to reach the host may block the main thread, which will cause the UI to freeze.

OS Errors are your first indication that something has failed in your request. App developers sometimes ignore them, but if you ignore them it is at the peril of your app. Because HTTP leverages layered networking, there is another layer of potential failures that may occur at the HTTP layer or at the application layer.

HTTP Errors

HTTP Errors are caused by problems with the HTTP request, HTTP server, or application server. HTTP errors are delivered to the requesting client via a status code in the HTTP response.

A 404 status is a common example of an HTTP error. It indicates that the resource specified in the URL cannot be found. The HTTP header shown in the following code snippet is an example of the raw output from an HTTP server when it cannot find the requested resource.

```
HTTP/1.1 404 Not Found
Date: Sat, 04 Feb 2012 18:32:25 GMT
Server: Apache/2.2.14 (Ubuntu)
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 248
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

You can find the status code in the first line of the response. An HTTP response may have an associated message body containing friendly human readable content describing what happened. You should not use the presence or absence of a response body as an indicator as to whether the HTTP request succeeded.

There are five categories of HTTP errors:

- **Informational 100-level —** Solely informational from the HTTP server and indicates that the processing of the request will continue but with a caveat.
- **Successful 200-level —** The server processed the request. Each 200-level status indicates a different result of the successful request. For example, a 204 indicates that the request was successful, but no payload was returned to the client.
- **Redirection needed 300-level —** Indicates that the client must perform some action to continue the request because the desired resource has moved. The synchronous request methods of the URL loading system handle redirects automatically without your code being notified. If your application needs to do custom handling with redirects, it should use asynchronous requests.
- **Client Errors 400-level —** Indicates that the client has sent erroneous data that the server cannot correctly handle. For example, an unknown URL or a malformed HTTP header causes errors in this range.
- **Downstream errors 500-level —** Indicates that an error occurred between the HTTP server and any downstream application servers. For example, if the web server calls a JavaEE application server and the servlet fails with a `NullPointerException`, then the client receives a 500-level error.

The URL loading system in iOS handles the parsing of HTTP headers and makes it easy to retrieve the HTTP status. If your code makes a synchronous call using an HTTP or HTTPS URL, then the returned response object will be an instance of `NSHTTPURLResponse`. The `NSHTTPURLResponse` object has a `statusCode` property that returns the numeric HTTP status of the request. The

following code demonstrates the validation of both the `NSError` object and the return of a successful status from the HTTP server.

```
NSHTTPURLResponse *response=nil;
NSError *error=nil;
NSData *myData  = [NSURLConnection sendSynchronousRequest:request
                                        returningResponse:&response
                                                    error:&error];

// Check the return
if ((!error) && ([response statusCode] == 200)) {
    // looks like things worked
} else {
    // things broke, again.
}
```

If the URL of the request could be something other than HTTP, the application should validate that the response object is actually an `NSHTTPURLResponse`. The preferred method to validate the type of the object is using the `isKindOfClass:` method on the returned object, as shown in the following code:

```
if ([response isKindOfClass:[NSHTTPURLResponse class]]) {
    // It is a HTTP response, so we can check the status code
   ...
```

For definitive information on HTTP status codes see W3 RFC 2616 at `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

Application Errors

This section discusses errors generated by the next layer in the network protocol stack: the application layer. Application errors differ from OS or HTTP errors because there is no standard set of values or causes for these errors. These errors are caused by the business logic and application running in the service tier. In some situations the errors may be code failures, such as exceptions, but in others the errors may be semantic errors, such as an invalid account number supplied to the service. In the former situation it is advisable to generate an HTTP 500-level error, but the latter scenario should return an error code in the application payload.

For example, an application error would be reported in a mobile banking application if the user tried to transfer more funds from an account than were available for transfer. If such a request were made, the OS would report that the request was successfully sent and a response received. The HTTP server would report that the request was received and a response sent, but the application layer must report that the transaction failed.

The best practice for reporting application errors is to wrap all application payload data in a standard envelope that contains a consistent location for application errors. In the funds transfer example, the business payload of a successful transfer response may look like:

```
{ "transferResponse":{
    "fromAccount":1,
```

```
        "toAccount":5,
        "amount":500.00,
        "confirmation":232348844
      }
}
```

The response contains the source and destination accounts, the amount transferred, and a confirmation number. Including any error codes and messages directly into the `transferResponse` object would make locating the error code and message difficult. If each action includes error reporting in its own response object, any error reporting logic could not be reused across the application. Using a packet structure like the one in the following code sample allows the application to quickly determine if an error occurred by checking for the existence of the `"error"` object in the response JSON payload:

```
{"error":{
      "code":900005,
      "messages":"Insufficient Funds to Complete Transfer"
   },
   "data":{
      "fromAccount":1,
      "toAccount":5,
      "amount":500.00
   }
}
```

Any UI code to report errors can easily be reused because the error information is always in the `error` attribute of the response payload. Additionally, handling of the actual transaction payload is simplified because it is always under the same attribute name.

Regardless of the cause of a failed request, either OS, HTTP layer, or application, your application must know how to respond. You should spend time early in development considering all the failure modes of the application and design a consistent method to detect and respond to them.

RULES OF THUMB FOR HANDLING ERRORS

Errors can be caused by a multitude of conditions, and the best way to handle them can vary as greatly as the apps you write. Despite the complexity, some rules of thumb can help you cope with the uncontrolled nature of error conditions.

Include Error Handling In the Interface Contract

When designing a service interface, it is a mistake to specify only the input, outputs, and operations of the service. The interface contract should also specify how errors are communicated to the client. A service interface should leverage industry-standard means to communicate errors wherever possible. For example, the server should not define a new HTTP status value for a server side failure; instead it should use the appropriate 500-level status. If standard values are used, both the client-side and the server-side developers will have the same understanding of how errors will be communicated. Applications should never depend on accidental indications or overloaded property values to determine the presence of an error.

Application developers should also not depend on the behavior of the current server software stack to determine how to handle errors. When an iOS app is deployed, the server software stack may change behavior due to a future upgrade or replacement.

Error Statuses Lie

Mobile networking has one nonobvious behavior that differs dramatically from errors in traditional web applications: ambiguous error reporting. There are three possible outcomes of any network request from a mobile device to a server:

- The device has absolute positive confirmation that the operation succeeded. For example, both the `NSError` and HTTP status values indicate success, and the returned payload contains syntactically and semantically correct information.
- The device has absolute negative confirmation that the operation failed. For example, the returned application payload contained a failure indicator from the server that is specific to the operation attempted.
- The device has ambiguous negative confirmation that the operation failed. For example, the mobile app sends an HTTP request to transfer funds between two accounts. The request is received and successfully processed by the bank systems; however, the reply gets lost due to a network failure, and the `NSURLConnection` reports a timeout. The timeout did occur, but only after the transfer request was successful. If the transfer is retried, it results in a duplicate transfer and possibly overdrawn accounts.

This third scenario is the scenario that can cause unexpected and undetected misbehavior of the app. If the app developers do not know that the third scenario exists, they may make bad assumptions about a failure and inadvertently retry an operation that already succeeded. It is not just enough to know that the full request failed; rather, the developers must consider what could cause the request to fail and whether it is appropriate to automatically retry every failed request.

Validate the Payload

App developers should not impute that the payload is valid based on the fact that no OS or HTTP error was reported. Many scenarios can occur in which the request appears to have succeeded, but the payload is invalid. Any payload transferred between client and server should have a mechanism for validation. JSON and XML are examples of payload formats that have validation mechanisms, but neither comma separated value (CSV) files nor HTML do.

Separate Errors from Normal Business Conditions

The service contract should not report normal business conditions as errors. For example, if you have a user whose account is locked due to potential fraud, the lock status should be reported in the data payload rather than as an error condition. Separating errors from normal business conditions enables your code to maintain proper separation of concerns. Errors should be reserved for situations when things are broken.

Always Check HTTP Status

Always check the HTTP status on HTTP responses, and be explicit about the status values that are successful. This is the case even when making repeated calls to the same service. The status of the server can change at any time, even between juxtaposed calls.

Always Check NSError

Your app code should always check the returned `NSError` value to make sure nothing broke at the OS level. This is true even if you know that the app always runs on a well-run and tightly controlled Wi-Fi network. Things do not always work correctly, and your code needs to be defensive when dealing with the network.

Develop a Consistent Method for Handling Errors

The causes of networking errors are too numerous to enumerate, and the variety and scope of their impact can be overwhelming. When you design your app, don't just focus on consistent user interface patterns or a consistent naming scheme. You should also design a consistent pattern for dealing with network errors. This pattern should consider all the types of errors your app may encounter. Your app cannot consistently communicate errors to the user if it is not handling those errors in a consistent manner internally.

Always Set a Timeout

The default request timeout interval for an HTTP request in iOS is 4 minutes, which is a long time for a mobile application, and most users do not spend 4 consecutive minutes in any application. Developers need to choose a reasonable timeout value by evaluating the probable response times for any network requests and then factor in network delays for the worst-case network scenario. The following example demonstrates creating a request with a 20-second timeout.

```
- (NSMutableURLRequest *) createRequestObject:(NSURL *)url {
    NSMutableURLRequest *request = [[[NSMutableURLRequest alloc]
                                          initWithURL:url
                                          cachePolicy:NSURLCacheStorageAllowed
                                      timeoutInterval:20
                                    autorelease];
    return request;
}
```

GRACEFULLY HANDLING NETWORK ERRORS

iOS makes network communications relatively easy, but responding to all the types of errors and edge conditions that can occur is not easy. It is all too common to hook up networking code to see results quickly and plan on handling all the error conditions later. For nonmobile applications you can usually get away with this approach because the network connectivity from a workstation is predictable. If the network was there when the application loaded, it is almost always still there when the user loads the next page. In the rare case that it is not, the developer can depend on the browser to take care of displaying a message to the user. If you delay adding exception handling in a mobile app, you

end up in a situation in which the network code needs significant refactoring as every new error case is encountered.

This section describes a design pattern that creates an elegant and robust framework for exception handling and requires little work to extend it for new errors in the future.

Consider the three major exception cases for mobile communications:

- The remote server is not reachable due to insufficient network connectivity from the device.
- The remote server returns an error response because of an OS error, HTTP error, or application error.
- The device attempts an unauthenticated request to a server that requires authentication.

As the number of potential exception conditions increase linearly, the amount of code required to handle them increases exponentially. If your code attempts to deal with each type of error on each type of request, the complexity and volume of your code also increases exponentially. This pattern attempts to bend that exponential curve back to a more linear curve.

Design Pattern Description

The pattern described in this section uses a Command Dispatch pattern combined with broadcast notifications. This pattern consists of the following object types:

- Controllers
- Command objects
- Exception listeners
- Command queues

The next section describes the behavior at a high level for each type of object.

Object Descriptions

This section describes the attributes and properties of the objects comprising the Command Dispatch pattern.

Controllers

Controllers are typically view controllers that request data and process the results. In this design pattern the controllers do not need to contain any exception handling logic. The only error cases they need to handle are a successful completion or a completely unrecoverable failure of the service. In the unrecoverable failure scenario, the controller would typically pop itself off the view stack because the user has already been informed of the failure by the exception listener objects described next. Controllers create commands and listen for the command's completion.

Command Objects

Command objects correlate to the different network transactions that the application performs. Examples of command object requests include retrieving images, fetching JSON data from a

specific REST endpoint, or POSTing information to a service. A command object is a subclass of `NSOperation`. Because much of the logic in a command object is common to all other command objects, you can create a superclass command object to handle it and let specific commands inherit that logic. A command object has the following attributes:

- **Completion notification name —** In iOS, controllers register themselves as observers for this notification name. When the service call returns successfully, the command object uses `NSNotificationCenter` to broadcast a notification with this name. Although it is usually unique to the command class, in some situations it can be unique to a specific instance if there are multiple controllers issuing the same command types that want to distinguish individual responses.
- **Server error exception notification name —** Special exception handler objects listen for this notification. The command object uses `NSNotificationCenter` to broadcast a message with this name when the server times out or returns an OS or HTTP error not related to authentication. All command classes usually share the same exception names, and therefore the same exception listeners. But different classes of commands may necessitate different exception listeners and have different server error exception names.
- **Reachability exception notification name —** The command object produces a notification of this type when it detects a lack of reachability to the Internet or a target host. Another exception listener may listen for this type of exception. In some apps this type of exception is not needed because the server error exception listener handles the reachability exceptions.
- **Authentication exception notification name —** The command object may produce a notification of this type if it determines the user is not authenticated or the server reports an unauthenticated status. A third exception listener waits for this type of notification to appear. The authentication notification names are typically shared across all notifications in the app.
- **Custom attributes —** These attributes are specific to the request being made. The issuing controller typically supplies these values because they are the specific business data needed for the service call and vary for each one.

Exception Listeners

Each exception listener is typically instantiated by the app delegate and remains in the background waiting for its specific type of notification. In many cases the exception listener displays a modal view controller when it receives a notification, which is described later in the Exception Listener behavior section.

Command Queue

Controllers submit commands to the command queue for processing, and an app may have one or more command queues. In iOS, command queues are subclasses of `NSOperationQueue`. The main queue should not be used as a command queue because its operations run on the user interface thread, which impairs the user experience when executing long-running operations. Using `NSOperationQueues` provides built-in capability for managing the number of active operations and interdependencies between operations.

Object Behaviors

Each of these objects has a distinct part to play in successfully completing a network transaction. The following section describes their respective roles in this pattern.

Controller Behaviors

Controllers are focused on executing UI and business logic. When a controller wants data from a service, it should take the following actions:

1. Create a network command object.
2. Initialize the request for specific attributes of the command object.
3. Register as an observer for the completion of the command.
4. Push the command onto an operation queue for execution.
5. Wait for the `NSNotificationCenter` to deliver a completion notification.

When the operation completes, the controller receives a completion notification and takes the following actions:

1. Checks the status of the operation to see if it was successful.
2. If successful, it processes the received data. The received data is supplied to the controller via the `userInfo` attribute of the `NSNotification` object. `NSOperationQueues` execute `NSOperation` objects on their own thread. When the operation completes it sends an `NSNotification` via the `NSNotificationCenter`. The notification callback methods are called on the thread on which the `NSOperation` runs, which in the example ensures that it arrives on a thread other than the main thread. If the controller manipulates the UI, then it needs to make those changes on the main thread, usually via Grand Central Dispatch (GCD).
3. If unsuccessful, the controller has a number of options depending on the application requirements. For example, it may pop itself off the view stack or update the UI, indicating that the data is not available. It should not ask to retry or show a modal alert because those actions are the responsibility of an exception listener.
4. The controller should unregister itself as an observer for the commands' completion notification. In some cases this is not desirable if the controller wants to monitor for other data arriving from the same command type.

Notice that controllers do not have any logic to handle retries, timeouts, authentication, or reachability; that logic is all done by the commands and exception listeners.

If the controller wants to guarantee that only it receives the returned data, it should alter the completion notification name for that instance of the command object to be a unique value prior to placing it on the queue and listen for notifications of that unique name.

Command Object Behaviors

Command objects are responsible for calling the target service and broadcasting the results of that service call. The steps generally taken by a command object follow:

1. Check for reachability. If the network is not reachable, broadcast a reachability exception notification.
2. Check for authentication status if required. If the user is not yet authenticated, broadcast an authentication exception notification.

3. Build the network request using the custom properties provided by the controller. Usually the endpoint URL is specified as a static attribute of the command object class or loaded from a configuration subsystem.
4. Issue the network request using a synchronous request. See the "Synchronous Requests" section in Chapter 3 for more details on this.
5. Check the status of the request. If the status is an OS or HTTP error, it broadcasts a server exception notification. If the error is an authentication error, it broadcasts an authentication exception notification.
6. Parse the results. See Chapter 4.
7. Broadcast a completion notification with a successful status.

When a command object broadcasts a notification, completion or otherwise, it needs to create a dictionary object that contains a copy of itself, the status of the call, and any data returned as a result of the call. The self-copy is necessary because an instance of an `NSOperation` can be executed only once. As discussed in the next section, a command may be resubmitted when the listener handles the exception.

The synchronous request API is ideally suited to this pattern because the commands are executed on a background thread instead of the main thread. If the request transmits or returns a larger amount of data than you want to squeeze into memory, your application needs to use asynchronous requests. Because the main function of an `NSOperation` is a single method, the operation must implement concurrency locking to block its `main` method until the asynchronous call completes.

Exception Listener Behaviors

Exception listeners are the magic that makes this pattern especially powerful. These objects are usually created by the app delegate and remain in memory listening for notifications. When a notification is received, it is the responsibility of the listener to inform the user and potentially solicit a response from the user (other than throwing the phone through a wall). In the case of an exception, the notification contains a copy of the command that triggered the exception, and after the user has responded, the listener usually resubmits the command back onto the queue to be retried. One interesting caveat for the exception listeners is that because multiple commands may be in-flight there may be multiple exception notifications generated while the user is still responding to the first exception. Because of this, the exception listeners must collect exception notifications and resubmit all the triggering commands after the user responds to the first exception. This collection of errors prevents a common form of app misbehavior where the user is bombarded with `UIAlertViews` triggered by the same fundamental problem.

The flow for a server exception can be as follows:

1. Present a nice looking modal dialog explaining the error and giving the user the option to cancel or retry.
2. Collect any other server exceptions that may be broadcast.
3. If the user selects retry, dismiss the dialog and resubmit all the collected commands.

4. If the user selects cancel, dismiss the dialog. The listener should set the command completion status to failed for all the collected commands and ask each one to broadcast a completion notification.

The flow for a reachability exception may be as follows:

1. Present a nice looking modal dialog informing the users they need to be on a network with connectivity.
2. Collect any other service exceptions that may be broadcast.
3. Listen for reachability changes. When the network is reachable, dismiss the dialog and resubmit the collected commands.

The flow for an authentication exception is a bit more complicated. Keep in mind that commands are independent of one another, and many can be in flight at any one time. The authentication flow does not generate authentication exception notifications. The flow may look like this:

1. Present a login view modally.
2. Continue collecting commands that failed due to authentication errors.
3. If the user cancels, the listener should send a completion notification for the collected commands with a failure status.
4. If the user provides credentials, create a login command, and place it on the command queue.
5. Wait for a completion notification from the login command.
6. If the login didn't succeed due to a username/password mismatch, return to step 2. Otherwise, dismiss the login view controller.
7. If the login command was successful, resubmit the triggering commands to the command queue.
8. If the login command failed, then ask the triggering commands to send a completion notification with a failure status.

Command Queue Behaviors

Command queues are native iOS `NSOperationQueue` objects. By default a command queue operates in first-in-first-out (FIFO) order. When your code adds a command object to an `NSOperationQueue` it performs the following actions:

1. Retain the command object so that its memory will not be released.
2. Wait until an available slot comes open at the head of the queue.
3. When the command object arrives at the head of the queue the `start` method of the command object is invoked.
4. The main method of the command object is invoked.

Refer to iOS API documentation on the `NSOperation` and `NSOperationQueue` objects for detailed information on the interaction between the queue and the command objects.

Command Dispatch Pattern Example

This section provides an example of the Command Dispatch pattern by calling an authenticated service from YouTube. In this type of communication, a number of failure modes need to be considered.

- The user may not provide valid credentials.
- The device may not be on a functional network.
- YouTube may not respond in a timely manner or may fail for some reason.

The application needs to handle each of these conditions in an elegant and reliable manner. This example surveys the major code components and discusses some of the implementation details.

The app in the included project is a simple demonstration app. It is not intended for anything other than demonstrating this pattern.

Prerequisites

The things you need to have to successfully see this app operate are as follows:

- A YouTube account
- At least one video uploaded to your YouTube account (it doesn't need to be public, just uploaded to the account)
- The project zip file from the Wrox companion website

This project was developed using XCode 4.1 and iOS 4.3. The application was developed using the YouTube API as it stood in October 2011. It is under Google's control however and is subject to change.

Major Objects

After you download the project and load it up in XCode, you see the following classes.

Commands

In the `commands` group you'll find the following.

BaseCommand

The `BaseCommand` object is the superclass for all the command objects. It provides many methods needed by every command class. These methods include:

- Methods to send completion, error, and login needed notifications
- A method to help issuing objects listen for completion notifications
- Methods used to support the actual `NSURLRequests`

`BaseCommand` extends `NSOperation` so all the logic of the command is in the *main* method of each subclass of this object.

GetFeed

The main method of this command, shown in Listing 5-1, calls YouTube and loads the list of videos uploaded by the currently logged in user. YouTube determines the identity of the logged in user by a token passed in an HTTP header on the request. Without that header, YouTube returns an HTTP status code of 0 instead of a more standard 4xx HTTP error.

LISTING 5-1: CommandDispathDemo/service-interface/GetFeed.h

```
- (void)main {
    NSLog(@"Starting getFeed operation");
    // Check to see if the user is logged in
    if ([self isUserLoggedIn]) { // only do this if the user is logged in


        // Build the request
        NSString *urlStr =
                 @"https://gdata.youtube.com/feeds/api/users/default/uploads";
        NSLog(@"urlStr=%@",urlStr);
        NSMutableURLRequest *request =
                 [ self createRequestObject:[NSURL URLWithString:urlStr]];

        // Sign the request with the user's auth token
        [self signRequest:request];

        // Send the request
        NSHTTPURLResponse *response=nil;
        NSError *error=nil;
        NSData *myData  = [self sendSynchronousRequest:request
                                          response_p:&response
                                               error:&error];

        // Check to see if the request was successful
        if ([super wasCallSuccessful:response error:error]) {
            [self buildDictionaryAndSendCompletionNotif: myData];
        }
    }
}
```

In this code listing, many of the methods that called on `self` are implemented in the `BaseCommand` superclass. The `GetFeed` command is prototypical of the Command Dispatch pattern. The main method checks to make sure the user is logged in because there's no reason to call the server if you know this call will fail. If the user is logged in, then the code builds the request, adds the authentication header to it, then sends a synchronous request. The final part of the code calls a superclass method to determine if the call succeeded. This method uses both the `NSError` object and the HTTP status code from the `NSHTTPURLResponse` object to determine success. If the call failed, then either an error notification or login needed notification is broadcast.

LoginCommand

This method sends the request to YouTube to authenticate the user. This command is somewhat more involved because it doesn't use several of the helper methods found in the `BaseCommand` object.

It does not use these methods because it should not generate a Needs Authentication failure message if the login fails. It reports only a status of good completion or failed completion.

The login listener handles the errors that come from failed login attempts. For more information on the protocol that YouTube requires, reference `http://code.google.com/apis/youtube/2.0/developers_guide_protocol_understanding_video_feeds.html`.

Exception Listeners

In the *listeners* group you'll find the view controllers that are presented when an error occurs or when the user needs to log in. Both the `NetworkErrorViewController` and the `LoginViewController` extend the `InterstitialViewController`, which provides several common helper methods. Both view controllers are presented as modal view controllers.

- `NetworkErrorViewController`: Provides the user with the choice to retry or abort the failed operations. If the user selects retry, then the failed commands are placed back on the operation queue.
- `LoginViewController`: Solicits a username and password from the user. It stays at the top of the view stack until the user successfully logs in.
- `InterstitialViewController`: As a parent of the other exception listeners, it provides support functionality such as the code to collect multiple error notifications and re-dispatch them upon error resolution.

The key code in the listeners is found in the `viewDidDisappear:` method (see Listing 5-2), which is called when the view has completely disappeared. If the commands are queued before the view has completely disappeared, there is a chance that another error may trigger a repeated presentation of the view, thereby causing a fatal error in the application. iOS 5 has a better capacity to handle this case because users can specify a block of code to execute when the view disappears. The code does not need to determine the cause of the disappearance before handling the triggering commands.

LISTING 5-2: CommandDispatchDemo/NetworkErrorViewController.m

```
- (void) viewDidDisappear:(BOOL)animated {
    if (retryFlag) {
        // re-enqueue all of the failed commands
        [self performSelectorAndClear:@selector(enqueueOperation)];
    } else {
        // just send a failure notification for all failed commands
        [self
           performSelectorAndClear:
                          @selector(sendCompletionFailureNotification)];
    }
    self.displayed = NO;
}
```

The application delegate registers itself as the listener for both network error and login-needed notifications (see Listing 5-3). It collects exception notifications and manages the presentation of the correct view controller when an error occurs.

The code demonstrates the notification handler for the login-needed notification. Because it deals with the user interface, its contents must be executed on the main thread using GCD.

LISTING 5-3: CommandDispatchDemo/CommandDispatchDemoAppDelegate.m

```
/**
 * Handles login needed notifications generated by commands
 **/
- (void) loginNeeded:(NSNotification *)notif {
    // make sure it all occurs on the main thread
    dispatch_async(dispatch_get_main_queue(), ^{
        // make sure only one thread adds a command at a time
        @synchronized(loginViewController) {
            [loginViewController addTriggeringCommand:
                                 [notif object];
            if (!loginViewController.displayed) {
                // if the view is not displayed then display it.
                [[self topOfModalStack:self.window.rootViewController]
                            presentModalViewController:loginViewController
                            animated:YES];
            }
            loginViewController.displayed = YES;
        }
    }); // End of GC Dispatch block
}
```

View Controllers

There is one primary view controller in this simple app. The `RootViewController` (see the following code) extends `UITableViewController`. When this controller loads it creates and enqueues a command to load the user's list of videos (aka the YouTube feed). It patiently waits for the completion of that command by yielding the flow of control back to the main run loop. It is blissfully unaware that it will always fail on the first call because the user is not logged in.

The `requestVideoFeed` method found in `CommandDispatchDemo/RootViewController.m` starts the process to load the video list like so:

```
- (void)requestVideoFeed {
    // create the command
    GetFeed *op = [[GetFeed alloc] init];

    // add the current authentication token to the command
    CommandDispatchDemoAppDelegate *delegate =
            (CommandDispatchDemoAppDelegate *)[[UIApplication
                                        sharedApplication] delegate ];
    op.token = delegate.token;

    // register to hear the completion of the command
```

```
    [op listenForMyCompletion:self selector:@selector(gotFeed:)];

    // put it on the queue for execution
    [op enqueueOperation];
    [op release];
}
```

Notice that the code does not need to check if the user is logged in; the command does that when it executes.

The `gotFeed:` method, shown in the following code, handles the eventual return of data from YouTube. Midway through the example the `requestVideoFeed:` method registers the `gotFeed:` method as the target method for the completion notification. This method loads the data for the table view if the call succeeds. Otherwise it shows a `UIAlertView`.

```
- (void) gotFeed:(NSNotification *)notif {
    NSLog(@"User info = %@", notif.userInfo);
    BaseCommand *op = notif.object;
    if (op.status == kSuccess) {
        self.feed = op.results;

        // if entry is a single item, change it to an array,
        // the XML reader cannot distinguish single entries
        // from arrays with only one element
        id entries = [[feed objectForKey:@"feed"] objectForKey:@"entry"];
        if ([entries isKindOfClass:[NSDictionary class]]) {
            NSArray *entryArray = [NSArray arrayWithObject:entries];
            [[feed objectForKey:@"feed"] setObject:entryArray forKey:@"entry"];
        }
        dispatch_async(dispatch_get_main_queue(), ^{
            [self.tableView reloadData];
        });
    } else {
        dispatch_async(dispatch_get_main_queue(), ^{
            UIAlertView *alert = [[UIAlertView alloc]
                 initWithTitle:@"No Videos"
                       message:@"The login to YouTube failed"
                      delegate:self
             cancelButtonTitle:@"Retry"
             otherButtonTitles:nil];
            [alert show];
            [alert release];
        });
    }
}
```

The class `YouTubeVideoCell` is a `UITableViewCell` subclass that asynchronously loads the thumbnail of a video. It uses the `LoadImageCommand` object to accomplish this.

```
/**
 * Start the process of loading the image via the command queue
 **/
- (void) startImageLoad {
    LoadImageCommand *cmd = [[LoadImageCommand alloc] init];
```

```
        cmd.imageUrl = imageUrl;
        // set the name to something unique
        cmd.completionNotificationName = imageUrl;
        [cmd listenForMyCompletion:self selector:@selector(didReceiveImage:)];
        [cmd enqueueOperation];
        [cmd release];
    }
```

The issuing class changes the completion notification name. It does this so that it, and only it, receives a notification for this particular image. Otherwise it would need to examine the returned notification to see if it were the command that it originally issued.

The beauty of the Command Dispatch pattern is that all the messy exception handling logic and login presentation logic is divorced from the primary view controllers in the app. When a view controller generates a command, it is blissfully ignorant of any exception handling or authentication that occurs to actually complete the request. It simply issues a request, waits for a response, and processes the response. It does not care that it may have taken five retries and a user registration for the request to be completed successfully. In addition, the service request code does not need to know where the request originated or where the results are going; it can simply focus on executing the call and broadcasting the results.

Further benefits are seen when a developer can write initial happy-path code and see demonstrable results, and then in the future add the exception listeners with zero impact to the happy-path code. In addition, if designed properly, all the network service calls can leverage the same base command class resulting in abbreviated command classes.

In a universal app, you could alter the views presented by the exception listeners so that an error presentation on an iPhone is sized suitable for that platform, and error presentation on an iPad is better suited for the larger platform.

This pattern provides a way to rapidly show results, provide excellent separation of concerns between business logic and exception handling, reduce duplicate code, and provide for a better user experience.

SUMMARY

There are many sources of errors that can and will occur when your code uses the network. Understanding the source of the errors can help you quickly diagnose and resolve networking issues. Using the Reachability framework, your code can proactively respond to changing network conditions, thereby avoiding unnecessary network errors. Following a consistent pattern for issuing network requests and handing the successful or unsuccessful outcomes can make your code cleaner and more maintainable.

PART III
Advanced Networking Techniques

- **CHAPTER 6:** Securing Network Traffic
- **CHAPTER 7:** Optimizing Request Performance
- **CHAPTER 8:** Low-Level Networking
- **CHAPTER 9:** Testing and Manipulating Network Traffic
- **CHAPTER 10:** Using Push Notifications

6

Securing Network Traffic

WHAT'S IN THIS CHAPTER?

- How to verify your application is communicating with the correct server
- Authenticating with a service using HTTP and client-side certificates
- How to generate cryptographic hashes and use them to verify payload integrity
- Encrypting and decrypting data within an iOS application
- Tips for storing credentials using the device's keychain

WROX.COM DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code for this chapter is in the Chapter 6 download and is divided into two major sections:

- An Xcode project that includes a rudimentary mobile banking application that communicates with a simple web service
- A set of PHP scripts to serve as a web service for the mobile banking app, which handles authentication, fetching account details, and transferring funds

The average cost of a mobile-related data breach in 2011 was $194 per-record with an average total organizational cost of $5.5 million per incident ("2011 Cost of Data Breach Study." *Ponemon Institute© Research Report*, March 2012, `http://www.symantec.com/content/en/us/about/media/pdfs/b-ponemon-2011-cost-of-data-breach-us.en-us.pdf`). Given the highly networked nature of this world, it is of utmost importance that security be reviewed and addressed at every step of an application's development lifecycle.

To aid development of security-related requirements, Apple provides the Security framework and `CommonCrypto` interfaces to developers for use in their applications. The Security framework is a set of C APIs for managing certificates, trust policies, and access to the device's secure data store. `CommonCrypto` includes a set of interfaces to encrypt and decrypt data, generate common cryptographic hashes (for example, MD5 and SHA1), calculate a message authentication code, and derive password or passphrase-based keys.

This chapter covers how to use the Security framework with `NSURLConnection` to verify client and server identity. It also examines common authentication patterns and provides an example of encrypting transmitted data. Finally, it discusses how to decrypt server responses and securely store credentials using the device keychain.

The sample mobile banking app included in the chapter downloads helps illustrate the various points discussed in this chapter. The app contains a server-side component, which is developed in PHP for simplicity. PHP is relatively straightforward and should be easy to understand even if you lack previous experience with it. Important server-side snippets have been included throughout the chapter and the entire server-side source is available in the Chapter 6 download folder online.

VERIFYING SERVER COMMUNICATION

It is likely that users of your applications will be on the go; these are mobile applications after all, and you can rarely guarantee that any connection to the Internet is secure and rid of prying eyes. Most coffee shops offer free Wi-Fi to their patrons, but these networks are perfect for eavesdropping on one of your unsuspecting users. It is the developer's responsibility to ensure that users communicate only with the server(s) that you intend.

It is a best practice to use `NSURLProtectionSpace` to verify that users of your mobile banking application communicate with your secure banking servers, especially when issuing requests that manipulate data on the back end. `NSURLProtectionSpace` represents a server or realm that requires authentication and is a property of all inbound `NSURLAuthenticationChallenge`s.

The following code snippet illustrates how to create a protection space, which you can compare with the information contained in the challenge:

```
NSURLProtectionSpace *defaultSpace =
    [[NSURLProtectionSpace alloc]
        initWithHost:@"yourbankingdomain.com"
                port:443
            protocol:NSURLProtectionSpaceHTTPS
               realm:@"mobile"
authenticationMethod:NSURLAuthenticationMethodDefault];
```

Notice that you specified port 443, which corresponds to the `NSURLProtectionSpaceHTTPS` protocol. Table 6-1 lists additional supported protocols and their common port values. If you are not sure which port is configured on your server, you can log any properties of the inbound *challenge* to the console.

TABLE 6-1: Supported NSURLProtectionSpace Protocols

PROTOCOL CONSTANT	DEFAULT PORT
`NSURLProtectionSpaceHTTP`	80 or 8080
`NSURLProtectionSpaceHTTPS`	443
`NSURLProtectionSpaceFTP`	21 or 22

The application specifies an authentication method of `NSURLAuthenticationMethodDefault`. The default for the `NSURLProtectionSpaceHTTP` protocol is Basic authentication, so in this case, specifying `nil` or `NSURLAuthenticationMethodHTTPBasic` is the same as specifying `NSURLAuthenticationMethodDefault`. Following is a list of all supported authentication methods:

- `NSURLAuthenticationMethodDefault`
- `NSURLAuthenticationMethodHTTPBasic`
- `NSURLAuthenticationMethodHTTPDigest`
- `NSURLAuthenticationMethodHTMLForm`
- `NSURLAuthenticationMethodNTLM`
- `NSURLAuthenticationMethodNegotiate`
- `NSURLAuthenticationMethodClientCertificate`
- `NSURLAuthenticationMethodServerTrust`

Now that you have created a protection space with your server's attributes, you need to ensure that it is used to verify your connections. When your code requests a resource from the server that requires authentication, the server responds with an HTTP status code of `401 - Access Denied`. `NSURLConnection` receives this response and immediately sends a `willSendRequestForAuthenticationChallenge:` delegate message with a copy of the authentication challenge. Figure 6-1 covers the basic challenge-response process.

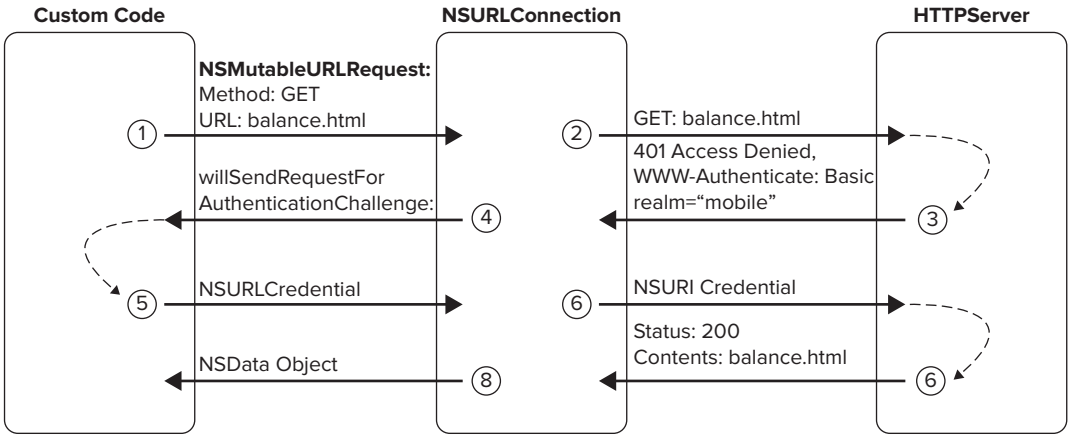


FIGURE 6-1

Implementing `willSendRequestForAuthenticationChallenge:` provides you with an opportunity to examine the challenge, determine if you want to respond to the server's authentication challenge, and issue the appropriate challenge response. Authentication Challenge responses are instances of `NSURLCredential`, which can be created for trusts, username/password combinations, and client certificates, which are discussed in detail in the next section. When creating an `NSURLCredential` for server trust, it's the responsibility of your delegate to evaluate the trust.

The following example outlines one possible implementation of protection space verification within `willSendRequestForAuthenticationChallenge:`.

```
- (void)connection:(NSURLConnection *)connection
      willSendRequestForAuthenticationChallenge:
      (NSURLAuthenticationChallenge *)challenge {

    // create an array of protection spaces for confirmation
    NSURLProtectionSpace *defaultSpace =
        [[NSURLProtectionSpace alloc]
            initWithHost:@"yourbankingdomain.com"
                    port:443
                protocol:NSURLProtectionSpaceHTTPS
                   realm:@"mobilebanking"
    authenticationMethod:NSURLAuthenticationMethodDefault];

    NSURLProtectionSpace *trustSpace =
        [[NSURLProtectionSpace alloc]
            initWithHost:@"yourbankingdomain.com"
                    port:443
                protocol:NSURLProtectionSpaceHTTPS
                   realm:@"mobilebanking"
    authenticationMethod:NSURLAuthenticationMethodClientCertificate];
```

```
    NSArray *validSpaces =
        [NSArray arrayWithObjects:defaultSpace, trustSpace, nil];

    // validate that the authentication challenge
    // came from a whitelisted protection space
    if (![validSpaces containsObject:challenge.protectionSpace]) {

        // dispatch alert view message to the main thread
        NSString *msg =
         @"We're unable to establish a secure connection.
           Please check your network connection and try again.";
        dispatch_async(dispatch_get_main_queue(), ^{
            [[[UIAlertView alloc] initWithTitle:@"Unsecure Connection"
                                        message:msg
                                       delegate:nil
                              cancelButtonTitle:@"OK"
                              otherButtonTitles:nil] show];
        });
        // cancel authentication attempt
        [challenge.sender cancelAuthenticationChallenge:challenge];
    }
    ...
}
```

This section has already covered how to create a protection space, but the preceding code snippet covers adding an additional protection space, which provides some flexibility to the back end. When you have determined the protection spaces to support, create them and add them to an array for comparison against inbound authentication challenges. In practice, you should define valid protection spaces as part of the model layer so that they can be reused across all your network operations. If the protection space from an authentication challenge does not match any of your supported spaces, you should inform the user and cancel the authentication challenge.

The code in the previous code example issues a `UIAlertView` to the main thread via Grand Central Dispatch. This is necessary because each logical unit of network activity that the application performs is created as a subclass of `NSOperation`, which is typically processed on a background thread. However, the Mobile Banking example application issues the request asynchronously on the main thread so that the application may respond to the `willSendRequestForAuthentication Challenge:` delegate method. The use of Grand Central Dispatch in this scenario is a safety precaution. Chapter 7, "Optimizing Request Performance," covers a more appropriate networking pattern and offers an efficient method for notifying view controllers of issues that require user action.

Now that you have implemented your server verification, how does it protect users of the app? This particular verification ensures that the app is communicating only with the servers that you have specified. Should they find themselves on a malicious network where traffic is being rerouted to a third-party's server, for example `yourbankingdomain.phishing.com`, the protection space verification would fail due to mismatched hosts, and further communication would be halted. More importantly, login credentials, bank account numbers, and so on would never be transmitted.

Sophisticated iOS applications often communicate with a web service, and those services may need to change quickly. Unfortunately, consumer iOS application changes require submission and

approval by Apple, which can be unpredictable. Your organization does not want to be in a situation where web service authentication must be changed immediately, possibly rendering your application inoperable. To mitigate this risk, the application should include protection spaces that allow for communication to back up authentication servers or some other alternative. The inclusion of multiple protection spaces allows you a certain amount of flexibility but is ultimately a security decision that each organization needs to evaluate.

Another approach that allows flexibility on the back end is choosing to verify only certain properties of an authentication challenge, such as the host, port, and protocol match a predefined set. For example, one could verify that the challenge was issued from a particular host using SSL over port 443, shown in the following code. If any of the conditions are not satisfied, the code immediately issues an alert to the user indicating that it was unable to establish a secure connection.

```
if (![challenge.protectionSpace.host
        isEqualToString:@"yourbankingdomain.com"] ||
    !challenge.protectionSpace.port == 443 ||
    ![challenge.protectionSpace.protocol
        isEqualToString:NSURLProtectionSpaceHTTPS]) {

    // if ANY of our challenge verifications fail, alert the user
    dispatch_async(dispatch_get_main_queue(), ^{
        NSString *msg = @"We're unable to establish a secure connection."
        [[[UIAlertView alloc] initWithTitle:@"Unsecure Connection"
                message:msg
                                  delegate:nil
                        cancelButtonTitle:@"OK"
                        otherButtonTitles:nil] show];
    });

    // cancel authentication
    [challenge.sender cancelAuthenticationChallenge:challenge];
}
```

Server verification is important, but it alone is not sufficient to protect against all attacks. For example, it does not protect against a man-in-the-middle attack in which someone is eavesdropping on network communication. To ensure the security of your users' data, additional security measures such as message integrity and data encryption must be carefully reviewed and considered.

AUTHENTICATING WITH HTTP

Authentication is the process to confirm the identity of the person trying to access a system. It is paramount that the example mobile banking service tier discerns between a real user and an imposter. This section covers common authentication patterns and how to handle those challenges within your iOS applications.

The banking application has two authentication modes: standard and expedited. Standard authentication simply prompts users to enter their username and password, whereas expedited authentication allows users to register a device and authenticate using a PIN without typing a username and password each time. To maintain security with expedited authentication, if the user chooses to register the device

on a given authentication request, the server response will include an additional attribute, the user's certificate. The application stores this certificate and checks for it on subsequent launches to determine which authentication view should display.

The standard authentication mode of your banking application uses HTTP Basic authentication, whereas expedited authentication uses a client-certificate downloaded from the web service. The following sections discuss each of these approaches.

HTTP Basic, HTTP Digest, and NTLM Authentication

The great thing about Basic, Digest, and NTLM authentication is that they are all username/password-based authentication. This means that you can handle all three authentication challenge types with the same logic. You tend to see Basic and Digest authentication more than NTLM, but NTLM is still used.

HTTP Basic authentication was defined by RFC 1945 (`http://tools.ietf.org/html/rfc1945`) and, as the name suggests, is basic. Username and password information are passed in plaintext making it susceptible to interception and manipulation. However, these weaknesses can be acceptable when paired with SSL, and that combination is a common authentication pattern.

HTTP Digest authentication was originally defined by RFC 2069 (`http://tools.ietf.org/html/rfc2069`) as a more secure form of authentication that is used by applying an MD5 hash to the password before it is transmitted and pairing it with a cryptographic *nonce*. A nonce is a random or pseudo-random number used to sign a message, but each individual value can be used only one time. Because each nonce value is used only once and then marked as expired, it prevents replay attacks that re-send a previously encrypted message. HTTP Basic and HTTP Digest authentication have since been combined into a single standard, RFC 2617 (`http://tools.ietf.org/html/rfc2617`).

NTLM is a Microsoft security protocol that provides authentication, integrity, and confidentiality services. NTLM authentication is a challenge-response protocol similar to HTTP Basic and HTTP Digest authentication. It has largely been supplanted by the Kerberos system but continues to be used to authenticate users remotely over the web. Kerberos is an authentication protocol developed by MIT based on the idea of "tickets" that allow secure identification over nonsecure networks.

Luckily, `NSURLConnection` handles most of the nonce and hash legwork for the various authentication methods, allowing you to simply specify credentials in the form of an `NSURLCredential` object. `NSURLCredential` fits most authentication requirements because it can represent credentials created from username/password combinations, client certificates, and server trusts. Credentials have a variety of persistence options: Do Not Persist, Persist for the Current Session Only, or Persist Permanently. Applications are granted access only to the credentials they create versus those to which the user grants access, as is the case in traditional Mac development.

The response logic is the same for HTTP Basic, HTTP Digest, and NTLM authentication. Listing 6-1 covers the additions required to `willSendRequestForAuthenticationChallenge:` to respond to an authentication challenge with your username and password.

LISTING 6-1: Handling Basic Authentication Challenges (/App/Mobile-Banking/AuthenticateOperation.m)

```
- (void)connection:(NSURLConnection *)connection
        willSendRequestForAuthenticationChallenge:
           (NSURLAuthenticationChallenge *)challenge {

...

    // respond to basic authentication requests
    // DIGEST and NTLM authentication follow this pattern
    if (challenge.protectionSpace.authenticationMethod ==
            NSURLAuthenticationMethodHTTPBasic) {

        // proceed with authentication
        if (challenge.previousFailureCount == 0) {
            NSURLCredential *creds =
                [[NSURLCredential alloc]
                    initWithUser:_username
                        password:_password
                     persistence:NSURLCredentialPersistenceForSession];

            [challenge.sender useCredential:creds
                 forAuthenticationChallenge:challenge];

        // authentication has previously failed.
    // depending on authentication configuration, too
        // many attempts here could lead to a poor user
    // experience via locked accounts

        } else {

            // cancel the authentication attempt
            [[challenge sender] cancelAuthenticationChallenge:challenge];

            // alert the user that his credentials are invalid
            // this would typically be handled in a cleaner
    // manner such as updating the styled login view
    NSString *msg = @"Invalid username / password.";
            dispatch_async(dispatch_get_main_queue(), ^{
                [[[UIAlertView alloc]
                    initWithTitle:@"Invalid Credentials"
                          message:msg
                         delegate:nil
                cancelButtonTitle:@"OK"
                otherButtonTitles:nil] show];
            });
        }
    }
...
}
```

After confirming that the challenge is for HTTP Basic or another supported challenge type, you should ensure that the challenge hasn't previously failed and create your `NSURLCredential` object using the username and password entered. If the challenge previously failed, alert the user and cancel the challenge. This is important because `willSendRequestForAuthenticationChallenge:` can be called multiple times. Depending on your configuration, if the user's credentials are invalid without this check in place, it's possible that an account could be locked after a single invalid credential submission. If the inbound challenge authentication method is not a type the app can handle, do not issue a response. This informs `NSURLConnection` that the application does not handle that particular authentication method.

Client-Certificate Authentication

Now that the user has successfully authenticated, assume the user registered the device during this particular authentication request. During device registration, the application must store the certificate returned from the authentication service. The following provides an example of what a successful service tier response may look like when it includes certificate data.

```
{
    "result": "SUCCESS",
    "additional_info": "Authentication Successful",
    "certificate": "<BASE64 Encoded Certificate>
}
```

Returned certificate data in the previous snippet is encoded in the PKCS #12 (.p12) file format, a commonly used standard published by RSA Laboratories, for exchanging certificate data with client applications. Listing 6-2 and Listing 6-3 outline how to decode Base 64 .p12 data, extract the identity and certificate information, and store credentials for future authentication requests.

LISTING 6-2: Authentication Response Handling with Certificate Data (/App/Mobile-Banking/AuthenticateOperation.m)

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {
    ...

    // unpack service response
    NSError *error = nil;
    NSDictionary *response = [NSJSONSerialization
                              JSONObjectWithData:self.responseData
                                         options:0
                                           error:&error];

    ...

    // create our client certificate if necessary,
    // and store in the credential store
    if (_registerDevice == YES) {
        NSString *certString = [response objectForKey:@"certificate"];
        NSData *certData =
            [NSData dataWithBase64EncodedString:certString];

        // retrieve the identity and certificate for our decoded data
```

continues

LISTING 6-2 *(continued)*

```
            SecIdentityRef identity = NULL;
            SecCertificateRef certificate = NULL;
            [Utils identity:&identity
             andCertificate:&certificate
             fromPKCS12Data:certData
             withPassphrase:@"test"];

            // store the certificate for future authentication challenges
            if (identity != NULL) {

                    // store the certificate and identity in
                    // the keychain as the default
                    NSURLProtectionSpace *certSpace =
                     [[NSURLProtectionSpace alloc]
                        initWithHost:@"yourbankingdomain.com"
                                port:443
                            protocol:NSURLProtectionSpaceHTTPS
                               realm:@"mobilebanking"
                authenticationMethod:
                                 NSURLAuthenticationMethodClientCertificate];

                    NSArray *certArray = [NSArray arrayWithObject:
                                          (__bridge id)certificate];
                    NSURLCredential *credential =
                        [NSURLCredential
                         credentialWithIdentity:identity
                                   certificates:certArray
                                    persistence:
                                          NSURLCredentialPersistencePermanent];

                    [[NSURLCredentialStorage sharedCredentialStorage]
                      setDefaultCredential:credential
                        forProtectionSpace:certSpace];

            }
        }
        ...
}
```

The majority of Listing 6-2 is straightforward with the possible exception of retrieving the identity and certificate from the .p12 certificate data returned by the service. Listing 6-3 details how to use the `SecPKCS12Import()` function within the Security Framework to import an identity and trust and then extract the certificate.

LISTING 6-3: Obtaining Identity and Certificate from .p12 Data (/App/Mobile-Banking/Utils.m)

```
+ (void)identity:(SecIdentityRef*)identity
  andCertificate:(SecCertificateRef*)certificate
  fromPKCS12Data:(NSData*)certData
```

```
withPassphrase:(NSString*)passphrase {

    // bridge the import data to foundation objects
    CFStringRef importPassphrase = (__bridge CFStringRef)passphrase;
    CFDataRef importData = (__bridge CFDataRef)certData;

    // create dictionary of options for the PKCS12 import
    const void *keys[] = { kSecImportExportPassphrase };
    const void *values[] = { importPassphrase };
    CFDictionaryRef importOptions = CFDictionaryCreate(NULL, keys,
                                                       values, 1,
                                                       NULL, NULL);

    // create array to store our import results
    CFArrayRef importResults = CFArrayCreate(NULL, 0, 0, NULL);
    OSStatus pkcs12ImportStatus = errSecSuccess;

    pkcs12ImportStatus = SecPKCS12Import(importData,
                                         importOptions,
                                         &importResults);

    // check if import was successful
    if (pkcs12ImportStatus == errSecSuccess) {
        CFDictionaryRef identityAndTrust =
                                CFArrayGetValueAtIndex (importResults, 0);

        // retrieve the identity from the certificate imported
        const void *tempIdentity = NULL;
        tempIdentity = CFDictionaryGetValue (identityAndTrust,
                                             kSecImportItemIdentity);
        *identity = (SecIdentityRef)tempIdentity;

        // extract the certificate from the identity
        SecCertificateRef tempCertificate = NULL;
        OSStatus certificateStatus = errSecSuccess;
        certificateStatus = SecIdentityCopyCertificate (*identity,
                                                         &tempCertificate);
        *certificate = (SecCertificateRef)tempCertificate;
    }

    // clean up
    if (importOptions) {
        CFRelease(importOptions);
    }
}
```

As part of the SSL handshake between the server and the application, `willSendRequestForAuthenticationChallenge:` will receive multiple callbacks with Server Trust and Client Certificate authentication challenges. You must determine which of these challenges your application needs to handle. The following example expands on Listing 6-1 to determine whether the application should issue a client certificate or standard user credentials for authentication.

```
// if this is a client certificate authentication request AND
// the user has already registered this device, attempt to issue
// the certificate to the service tier
if (challenge.protectionSpace.authenticationMethod ==
      NSURLAuthenticationMethodClientCertificate
      && devicePreviouslyRegistered) {

    // proceed with authentication
    if (challenge.previousFailureCount == 0) {

    // retrieve the default credential specifically for
    // client certificate challenges
    NSURLCredential *credential =
        [[NSURLCredentialStorage sharedCredentialStorage]
           defaultCredentialForProtectionSpace:
            [[Model sharedModel] clientCertificateProtectionSpace]];
        if (credential) {
             [challenge.sender useCredential:credential
                  forAuthenticationChallenge:challenge];
        }

    // authentication has previously failed.
    // depending on authentication configuration, too many attempts
    // here could lead to a poor user experience via locked accounts
    } else {
        // cancel the authentication attempt
        [[challenge sender] cancelAuthenticationChallenge:challenge];

        // alert the user that his credentials are invalid
        // this would typically be handled in a cleaner
        // manner such as updating the styled login view

    }

// either the user has not registered this device or
// this is not a client certificate challenge
} else {
        ...
        // perform authentication based on Listing 6-1
}

// if nothing catches this challenge,
// attempt to connect without credentials
 [challenge.sender
  continueWithoutCredentialForAuthenticationChallenge:challenge];
```

Within your service tier, you can retrieve the attributes of a certificate using the `openssl_x509_parse()` function, as outlined in the following code snippet. After you obtain the certificate attributes, there are a number of service tier authentication options available to you. One option is to verify the Issuer and then look up the private key from a list of known keys for the user. Another option would be to incorporate a PIN mechanism within the application, which is verified prior to issuing the client certificate to the authentication challenge.

```
if (array_key_exists('SSL_CLIENT_CERT', $_SERVER)){
    $clientCertData = openssl_x509_parse($_SERVER['SSL_CLIENT_CERT']);
    // using certificate attributes and encrypted PIN
    // verify identity and issue authentication tokens
} else {
    // issue failed authentication message
}
```

`NSURLConnection` intercepts server responses with untrusted certificates; this includes self-signed SSL certificates. If you test the authentication mechanisms discussed in this section with a self-signed SSL certificate, the majority of the network based code will not be executed. However, emailing your server certificate (.cer file extension) to an e-mail account configured on the device allows you to click and install it. When installed, `NSURLConnection` requests can recognize your server certificate as trusted and proceed with processing.

MESSAGE INTEGRITY WITH HASHING AND ENCRYPTION

Now that the app has verified that it is communicating with the correct server and has successfully authenticated, it can begin issuing service requests on the user's behalf. The app must ensure that the data it transmits is properly secure and unmodified during delivery. This section covers techniques to satisfy both requirements with cryptographic hashes, message authentication codes (MAC), and remove encryption.

These topics are implemented in the funds transfer functionality in the example Mobile Banking application (available for download on the companion website). The funds transfer request utilizes a combination of cryptographic hashes and encryption to ensure that the message is unreadable and delivered in an unaltered state. Although a number of payload samples are generated and discussed in this section, each follows the defined JSON payload structure outlined in the following code example.

```
{
    "mac": "Message Authentication Code",
    "iv": "Initialization Vector",
    "payload": {
        "toAccount": "123123456456",
        "fromAccount": "654654321321",
        "amount": 23.23,
        "transferDate": "2012-02-27",
        "transferNotes": "Book advance to savings."
    }
}
```

The payload property will be the only encrypted element in the request body. For the service tier to properly decrypt the payload, it requires the Initialization Vector (IV) used to encrypt the data within the application. Typically the IV is sent along with the data it was used to encrypt. Although it might seem like this weakens the message's security, it does not because the IV alone is not enough to decrypt the message. After the service tier has decrypted the payload, it generates a MAC using the same predefined set of payload attributes and compares it to the MAC from the inbound request

to verify message integrity. If the message were altered in any way during its journey, the codes will not match and the tainted message results in an error. The attribute set the example application uses in this section's example is the concatenation of the To Account, From Account, Amount, and Transfer Date, as seen later when discussing Message Authentication Codes.

Figure 6-2 contains an overview of the steps taking place during the request-response transaction and how each component is used throughout the encryption and decryption process on each side of the transaction. Important values for encryption and decryption as well as the MAC algorithm are known by both parties in the transaction. Also, the Request Body section aligns with the JSON payload structure defined in the previous example.

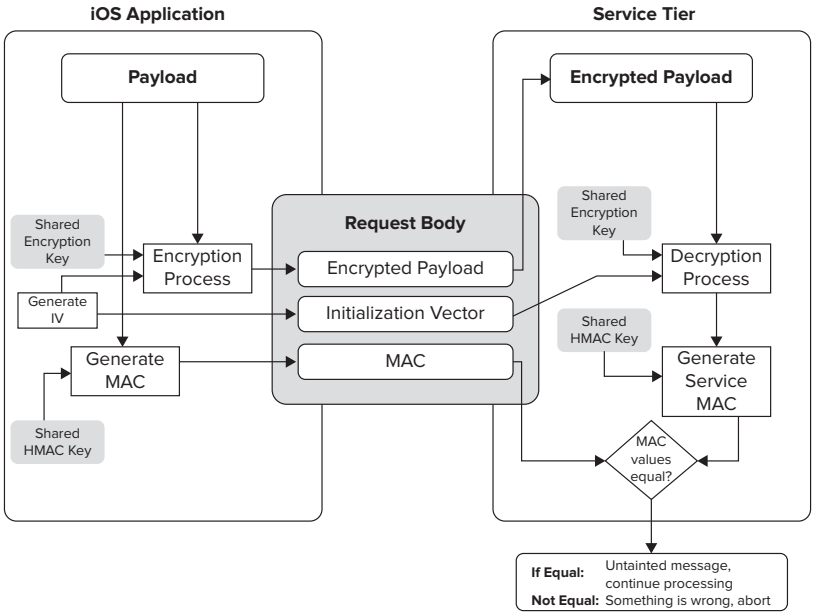


FIGURE 6-2

Hashing

Cryptographic hashes, or digests, generate fixed-size bit sequences for a given block of data. These hash values make comparing and sorting blocks of data easy. A few common uses of hashes include tracking file changes, downloading checksums, obfuscating data for database storage, and a rudimentary method to verify integrity of request data. A more robust approach uses MACs, which is covered in detail later in this section.

The iOS `CommonCrypto` library provides support for MD5, SHA-1, and SHA-256 digests as well as a few other less commonly used routines. The examples covered in this chapter focus only on MD5, SHA-1, and SHA-256; however, the process to generate hash values is the same for all `CommonCrypto` digest routines. Hashes can be created manually through a series of function calls to create a digest context, update the context with the data to be processed, and retrieve the value of the digest calculation, or you can use convenience functions provided for each digest routine.

To simplify practical use within your applications, create a category on `NSString` to implement your hash methods. `NSString+Hashing.h/.m` houses all your hash related category methods. Its interface definition is shown in Listing 6-4.

LISTING 6-4: NSString Hashing Category Definition (/App/Mobile-Banking/NSString+Hashing.h)

```
#import <CommonCrypto/CommonDigest.h>

enum {
    NJHashTypeMD5 = 0,
    NJHashTypeSHA1,
    NJHashTypeSHA256,
}; typedef NSUInteger NJHashType;

@interface NSString (Hashing)

- (NSString*)md5;
- (NSString*)sha1;
- (NSString*)sha256;
- (NSString*)hashWithType:(NJHashType)type;

@end
```

Listing 6-5 outlines the core logic within your hashing category. Each of the convenience methods, `md5`, `sha1`, and `sha256` call `hashWithType:`. Although there are some predefined hash enumerations throughout `CommonCrypto` libraries that the application could potentially use, none of the enumerations are defined within the context of digest calculations. Rather than relying on an enumeration that may change in the future, the application uses this custom one that also provides the added benefit of restricting the digest routines it supports.

LISTING 6-5: Core Digest Calculation Logic Within hashWithType (/App/Mobile-Banking/NSString+Hashing.m)

```
- (NSString*)hashWithType:(NJHashType)type {

    // Create pointer to the string as UTF8 - this is NULL terminated
    const char *ptr = [self UTF8String];

    // Create buffer with length for chosen digest
    NSInteger bufferSize;
    switch (type) {
        case NJHashTypeMD5:
            // 16 bytes
            bufferSize = CC_MD5_DIGEST_LENGTH;
            break;

        case NJHashTypeSHA1:
            // 20 bytes
            bufferSize = CC_SHA1_DIGEST_LENGTH;
```

continues

LISTING 6-5 *(continued)*

```
            break;

        case NJHashTypeSHA256:
            // 32 bytes
            bufferSize = CC_SHA256_DIGEST_LENGTH;
            break;

        default:
            return nil;
            break;
    }

    unsigned char buffer[bufferSize];

    // Perform hash calculation and store in buffer
    switch (type) {
        case NJHashTypeMD5:
            CC_MD5(ptr, strlen(ptr), buffer);
            break;

        case NJHashTypeSHA1:
            CC_SHA1(ptr, strlen(ptr), buffer);
            break;

        case NJHashTypeSHA256:
            CC_SHA256(ptr, strlen(ptr), buffer);
            break;

        default:
            return nil;
            break;
    }

    // Convert buffer value to pretty printed NSString
    // this will match the servers hash calculation
    NSMutableString *hashString = [NSMutableString stringWithCapacity:bufferSize * 2];
    for(int i = 0; i < bufferSize; i++) {
        [hashString appendFormat:@"%02x",buffer[i]];
    }

    return hashString;

}
```

The only portion of the `hashWithType:` implementation that may not be straightforward is the final step. The final step shown in Listing 6-5 loops through the byte output of the digest calculation and converts it to hexadecimal, a readable output. With the core hash logic complete and consolidated to a single method, implementing each convenience method requires only a single line of code each, as shown in Listing 6-6.

LISTING 6-6: Hashing Convenience Method Implementations (/App/Mobile-Banking/NSString+Hashing.m)

```
- (NSString*)md5 {
    return [self hashWithType:NJHashTypeMD5];
}

- (NSString*)sha1 {
    return [self hashWithType:NJHashTypeSHA1];
}

- (NSString*)sha256 {
    return [self hashWithType:NJHashTypeSHA256];
}
```

This approach has the added benefit of being easily extended to support additional digest calculations. The following example demonstrates calling each convenience method and the output it generates. The National Institute of Standards and Technology (NIST) provides test vectors to validate digest calculation output at `http://www.nsrl.nist.gov/testdata/`.

```
NSLog(@"MD5: %@", [@"test string" md5]);
NSLog(@"SHA1: %@", [@"test string" sha1]);
NSLog(@"SHA256: %@", [@"test string" sha256]);

Output:
MD5: 6f8db599de986fab7a21625b7916589c
SHA1: 661295c9cbf9d6b2f6428414504a8deed3020641
SHA256: d5579c46dfcc7f18207013e65b44e4cb4e2c2298f4ac457ba8f82743f31e930b
```

Generating hashes in the service tier is similar because PHP supports each of the digest routines implemented in Listing 6-6, plus a few dozen others. The standard function to generate a hash value is `hash()`, which accepts the algorithm to perform and the value to perform it on. In addition, PHP includes convenience functions for the generation of MD5 and SHA1 hashes. Generating hash values for each of the digests in PHP can be done like so:

```
echo "MD5: ".md5("test string")."</br>";
echo "SHA1: ".sha1("test string")."</br>";
echo "SHA256: ".hash("sha256", "test string")."</br>";

Output:
MD5: 6f8db599de986fab7a21625b7916589c
SHA1: 661295c9cbf9d6b2f6428414504a8deed3020641
SHA256: d5579c46dfcc7f18207013e65b44e4cb4e2c2298f4ac457ba8f82743f31e930b
```

The previous examples covered hashing string objects, but it is also just as easy to generate hashes of `NSData` objects by creating a similar category on `NSData`. However, if you have advanced hashing requirements or intend to compare hash values within your iOS application, you may want to consider creating a custom class, which can be written to optimize initialization and make hash

comparison easier by overriding `isEqualTo:`. Whereas using a hashing algorithm enables you detect content changes, Message Authentication Codes are paired with a key, making them more secure.

Message Authentication Codes

A *message authentication code* (*MAC*) is a mechanism for detecting payload modification and verifying authenticity by creating a hash of the inbound request data (or a pre-arranged subset of the request data) and comparing it with a precomputed MAC that is delivered with the payload. MACs are similar to the hash functions previously discussed but are more secure because they are paired with a secret key. As depicted in Figure 6-2, your application will compute a MAC that is sent with your request. This inbound MAC is then compared with a MAC computed in the service tier using the same key and dataset. If the MAC values are not equal, it is safe to assume that the message has been modified. Another approach would be to generate a MAC of the cipher text. While this accomplishes the same goal, it also allows you to determine if the message has been modified prior to executing a potentially expensive decryption process.

Although there are other MAC algorithms, examples in this section focus on Hash-Based Message Authentication Code (HMAC) because it is natively supported in iOS and most service-tier platforms. Often referred to as a keyed message authentication code, HMAC was defined by RFC2104 (`http://tools.ietf.org/html/rfc2104`). HMAC can use any hash function, typically either MD5 or SHA-1, but its strength depends on the strength of both the underlying hash function and the secret key. Although there are known weaknesses with the MD5 hash algorithm and SHA-1 is considered cryptographically stronger, those weaknesses do not compromise their use in HMACs.

The iOS HMAC implementation supports use of the MD5, SHA-1, SHA-224, SHA-256, SHA-384, and SHA-512 digest algorithms. The HMAC output length is always the same as the digest length of the hashing algorithm used. As with the other hash functions discussed previously, HMACs can be generated manually or using a convenience method, and the following examples demonstrate how to use the convenience method.

The HMAC example builds on the `NSString+Hashing` category you created earlier. To start, the category needs an additional method definition and two additional libraries imported, as outlined in Listing 6-7.

LISTING 6-7: HMAC Hash Addition (/App/Mobile-Banking/NSString+Hashing.h)

```
...
#import <CommonCrypto/CommonHMAC.h>
#import <CommonCrypto/CommonCryptor.h>

@interface NSString (Hashing)
...
- (NSString*)hmacWithKey:(NSString*)key;

@end
```

Importing `CommonCryptor` may seem out of place, but you need access to key size constant `kCCKeySizeAES256` in your implementation of `hmacWithKey:`, as outlined in Listing 6-8.

LISTING 6-8: hmacWithKey: Implementation (/App/Mobile-Banking/NSString+Hashing.m)

```
- (NSString*)hmacWithKey:(NSString*)key {
    // Pointer to UTF8 representations of strings
    const char *ptr = [self UTF8String];
    const char *keyPtr = [key UTF8String];

    // Implemented with SHA256, create appropriate buffer (32 bytes)
    unsigned char buffer[CC_SHA256_DIGEST_LENGTH];

    // Create hash value
    CCHmac(kCCHmacAlgSHA256,            // algorithm
           keyPtr, kCCKeySizeAES256,    // key and key length
           ptr, strlen( ptr ),          // data to hash and length
           buffer);                     // output buffer

    // Convert HMAC buffer value to pretty printed NSString
    NSMutableString *output =
        [NSMutableString
         stringWithCapacity:CC_SHA256_DIGEST_LENGTH * 2];
    for(int i = 0; i < CC_SHA256_DIGEST_LENGTH; i++) {
        [output appendFormat:@"%02x",buffer[i]];
    }

    return output;

}
```

Listing 6-8 includes the implementation of `hmacWithKey:`, which is similar to the `hashWithType:` method covered previously. The only two differences are the addition of a key and the restriction to use only the SHA-256 digest algorithm. UTF8 representations of your key and input data are created and passed to the `CCHmac()` function to create the HMAC. That function populates `buffer` with the HMAC value, and `hmacWithKey:` returns it as a string.

You are probably wondering where the `key` value comes from and how it was chosen. Keys are essential to cryptographic security, and the importance of key strength cannot be overstated! Without a properly secure and randomized key, you are opening the app up to numerous attacks. Depending on the data you attempt to protect, you could also be opening yourself up to potential lawsuits for cases such as data breaches and HIPAA violations. Recall the security overview from Figure 6-2: The encryption (E-Key) and MAC (M-Key) keys are the only two components of each step that are not transmitted with the request. The secrecy of those keys is your only hope at securing the transmitted data!

Because keys are the lynchpin of the security model, you must choose one methodically. The key's final length should match the key length for the encryption algorithm (or HMAC). Anything shorter than the algorithm's key length is NULL-padded until it reaches the full length, and thus weakens its randomness with each character it is short. If you must use user input data as the foundation for a key, salt the value by pre-pending a random or pseudo-random value to the input. If you need the ability to regenerate this key, make sure you store the salt used with your final key value. Lastly,

run the salted value through several thousand hash calculations using one of the routines discussed earlier, such as MD5 or SHA-1, and strip the appropriate number of bytes for use as your key.

If your application requires user input as the basis for your encryption key, consider using the `CCKeyDerivationPBKDF()` function from the `CommonCrypto/CommonKeyDerivation` library. `CCKeyDerivationPBKDF()` returns a key value for the salt, derivation algorithm, rounds of derivation, and output key length you specify. You can also use `SecRandomCopyBytes()` to generate a random array of bytes.

Although it is common for developers to output various processing details to the log, you should *never* print the generated key to the console. Log files can be easily retrieved from the device, which could present a serious security flaw if found by an attacker.

One downside to preshared keys is that the application needs to plan for key versioning. It is inevitable that you will come across a situation where you must modify your shared secret key, which requires you to deploy a new version of your application and then update the service tier. However, iOS users do not always diligently install app updates, and there is no mechanism to force them to do so. How do you handle users of the previous version of your application? You need to ensure that user transactions from your previous version will continue to be properly decrypted, verified, and processed by the service. Key versioning solves this problem without needing to issue an app update; however, it must be included in the app's development from the start.

Following is an example of how to generate an HMAC using the method created in Listing 6-8.

```
#import "NSString+Hashing.h"
...
// create mac input with to account, from account, amount, and transfer date
NSString *macCandidate = [NSString stringWithFormat:@"%@%@%@%@",
                          @"1234",                  // to account
                          @"4321",                  // from account
                          @"2300.00",               // amount
                          @"2012-12-25 00:00:00"];  // transfer date
// generate mac
NSString *mac = [macCandidate hmacWithKey:@"065a62448fb75fce3764dcbe68f9908d"];
...
Output:
51e66ca8fd8eb4bbe02fc6421e0dda1deb94f0c9518996a55bc7a4c242f1c8a9
```

This example uses a subset of the funds transfer payload as the MAC candidate; however, you can also use the entire payload. The concatenation order and attributes for the MAC candidate, which is the value you will hash, must be shared with the service tier to ensure proper decryption. If either the client or service tier operates under different assumptions, then the message integrity check will fail and nothing will be processed.

Now that the MAC is transmitted and the service tier has decrypted the payload, the service must generate the comparison MAC. The following code snippet demonstrates how to generate an HMAC in PHP using the same concatenated input string used on the client side. The `hash_hmac()` PHP function accepts similar input as the `HMac()` function in iOS and allows you to specify the algorithm to use, the content to hash, and the key to be used.

```
echo hash_hmac("sha256",
               "123443212300.002012-12-25 00:00:00",
               "065a62448fb75fce3764dcbe68f9908d");

Output:
51e66ca8fd8eb4bbe02fc6421e0dda1deb94f0c9518996a55bc7a4c242f1c8a9
```

Notice that the output values in the two previous examples match. Because the two values are the same, the service tier can trust that it received an unmodified request and can safely continue with the funds transfer.

Encryption

Now that you have generated the message authentication code, the client is ready to encrypt the request payload and send it to the server for processing. Although this section discusses two specific, symmetric encryption algorithms — the Advanced Encryption Standard and Data Encryption Standard — and their suitability for mobile devices, it is not a comprehensive discussion of encryption theory. However, a good rule of thumb is to trust the experts and use a standardized and vetted encryption algorithm instead of rolling your own.

ASYMMETRIC ENCRYPTION USING PUBLIC KEY CRYPTOGRAPHY

One topic not covered by this chapter is asymmetric encryption using public key cryptography. Using public key cryptography allows an application to download rotating public keys from your web service at a pre-determined interval and use them to encrypt and decrypt communications. The benefit here is that a key does not need to be previously shared for the encryption process to properly function. This option provides more deployment flexibility and long-term maintenance, but does complicate the encryption process, as both the device and web service must maintain a running list of keys and their respective identifiers so that encrypted messages may be properly decrypted.

The Data Encryption Standard (DES) was the United States government's encryption algorithm from 1976 until 1999. Between 1999 and 2002, the standard was upgraded to use a more-secure variant called Triple-DES. In 2002, the Advanced Encryption Standard (AES) was selected by the NIST to officially replace DES and Triple-DES. DES implementations are still around today, but most new implementations opt for Triple-DES. Even though AES is the official standard, Triple-DES has been approved for encryption of sensitive government information through 2030 by NIST.

Triple-DES uses two 56-bit keys and encrypts the data with the first key, encrypts that result with the second key, and then again with the first key. The process of performing three separate encryption passes is resource-intensive, which does not make Triple-DES the ideal candidate for mobile device encryption. Even as processors continue to improve, there are more efficient methods of encryption.

Alternatively, AES was designed for speed, strength, and more efficient use of resources, making it ideal for mobile devices. It requires only a single pass to encrypt data and provides an upper limit

of 256 exabytes (256 billion gigabytes) that can be encrypted in one message by the algorithm. One additional benefit of AES is that, as of iOS 4.3, messages greater than 1024 bytes are hardware accelerated. (Hardware acceleration applies to the SHA-1 digest calculations greater than 4096 bytes, as well.) Given current device storage limitations, AES use in iOS should be well within that limit.

The Mobile Banking application uses the AES encryption algorithm. However, in the interest of being thorough and supporting the previous standard that may still be in use, this book includes additional samples outlining how to use Triple-DES. The approach implemented in this section involves the creation of categories on `NSData` and `NSString`.

Encryption and decryption in Objective-C is similar to generating the cryptographic hashes discussed in the previous section. Cipher methods are packaged within `CommonCryptor`, a library of C functions, to encrypt and decrypt data that supports a number of algorithms as shown in Table 6-2. The library provides a convenience function, `CCCrypt()`, which performs a stateless encryption or decryption operation. This function is robust and should meet most cipher requirements, and is the approach covered in this chapter. The library also provides a suite of functions that allow developers more granular control over each step of the cipher process.

TABLE 6-2: Supported Encryption Algorithms

ENCRYPTION ALGORITHM	ALGORITHM CONSTANT
Advanced Encryption Standard (AES), 128-bit block	`kCCAlgorithmAES128`
Data Encryption Standard (DES)	`kCCAlgorithmDES`
Triple-DES, Three Key, EDE Configuration	`kCCAlgorithm3DES`
CAST	`kCCAlgorithmCAST`
RC4 Stream Cipher	`kCCAlgorithmRC4`
RC2 Block Cipher	`kCCAlgorithmRC2`
Blowfish Block Cipher	`kCCAlgorithmBlowfish`

The full iOS cipher process consists of creating a crypto context, processing your message, retrieving any remaining data, and then releasing the context. Although this chapter does not cover this process, there is one additional step that may be executed between retrieving the final output and releasing the context to improve performance in certain situations. This additional step resets the context, which allows you to process an additional message and even update the initialization vector if needed. If you need to process a large number of messages, this approach should be considered.

As with previous examples, you create a category on `NSString` and `NSData` to hold your cipher methods. Listing 6-9 and Listing 6-10 outline what each of your interface definitions should resemble. The interface definition for `NSData+Encryption` has additional method definitions for Base 64 encoding and decoding.

LISTING 6-9: NSData+Encryption Interface Definition (/App/Mobile-Banking/NSData+Encryption.h)

```
@interface NSData (Encryption)

- (NSData*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSData*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;

- (NSData*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSData*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;

+ (NSData*)dataWithBase64EncodedString:(NSString*)string;
- (id)initWithBase64EncodedString:(NSString*)string;

- (NSString*)base64Encoding;
- (NSString*)base64EncodingWithLineLength:(NSUInteger)lineLength;


@end
```

LISTING 6-10: NSString+Encryption Interface Definition (/Chapter6/App/Mobile-Banking/NSString+Encryption.h)

```
@interface NSString (Encryption)

- (NSString*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSString*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv;

- (NSString*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;
- (NSString*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv;

@end
```

With your interfaces defined, jump to the implementation for `NSString+Encryption`. Listing 6-11 covers the implementation for each of your string encryption and decryption methods.

LISTING 6-11: NSString Encryption and Decryption Method Implementations (/App/Mobile-Banking/NSString+Encryption.m)

```
#import "NSData+Encryption.h"
#import "NSData+Base64.h"
...
- (NSString*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *encrypted =
    [[self dataUsingEncoding:NSUTF8StringEncoding]
      encryptedWithAESUsingKey:key
                         andIV:iv];

    NSString *encryptedString = [encrypted base64Encoding];

    return encryptedString;
}
```

continues

LISTING 6-11 *(continued)*

```
- (NSString*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *encrypted =
    [[self dataUsingEncoding:NSUTF8StringEncoding]
      encryptedWith3DESUsingKey:key
                          andIV:iv];

    NSString *encryptedString = [encrypted base64Encoding];

    return encryptedString;
}

- (NSString*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *decrypted =
    [[NSData dataWithBase64EncodedString:self]
      decryptedWithAESUsingKey:key
                         andIV:iv];

    NSString *decryptedString =
    [[NSString alloc] initWithData:decrypted
                          encoding:NSUTF8StringEncoding];

    return decryptedString;
}

- (NSString*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {
    NSData *decrypted =
    [[NSData dataWithBase64EncodedString:self]
      decryptedWith3DESUsingKey:key
                          andIV:iv];

    NSString *decryptedString =
    [[NSString alloc] initWithData:decrypted
                          encoding:NSUTF8StringEncoding];

    return decryptedString;
}
```

Each of the `NSString` methods is similar; the primary differences are the direction of Base 64 encoding and how the result is encoded prior to being returned. Now, it is time to implement the core of the cipher process in the `NSData+Encryption` category, as outlined in Listing 6-12 and Listing 6-13.

LISTING 6-12: AES Encryption (/App/Mobile-Banking/NSData+Encryption.m)

```
#import <CommonCrypto/CommonCryptor.h>
#import "NSData+Base64.h"
...
- (NSData*)encryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;
```

```
    NSMutableData *encryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSizeAES128];

    CCCryptorStatus status = CCCrypt(kCCEncrypt,
                                     kCCAlgorithmAES128,
                                     kCCOptionPKCS7Padding, //CBC Padding
                                     keyData.bytes,
                                     keyData.length,
                                     iv.bytes,
                                     self.bytes,
                                     self.length,
                                     encryptedData.mutableBytes, //data out
                                     encryptedData.length,
                                     &dataMoved); // total data moved

    if (status == kCCSuccess) {
        encryptedData.length = dataMoved;
        return encryptedData;
    }

    return nil;

}
```

LISTING 6-13: Triple-DES Encryption (/App/Mobile-Banking/NSData+Encryption.m)

```
- (NSData*)encryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;
    NSMutableData *encryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSize3DES];

    CCCryptorStatus result = CCCrypt(kCCEncrypt,
                                     kCCAlgorithm3DES,
                                     kCCOptionPKCS7Padding, // CBC Padding
                                     keyData.bytes,
                                     keyData.length,
                                     iv.bytes,
                                     self.bytes,
                                     self.length,
                                     encryptedData.mutableBytes,// data out
                                     encryptedData.length,
                                     &dataMoved); // total data moved

    if (result == kCCSuccess) {
        encryptedData.length = dataMoved;
        return encryptedData;
    }

    return nil;

}
```

Listing 6-12 and Listing 6-13 cover encryption for AES and Triple-DES, using `CCCrypt()`, respectively. The `CCCrypt()` function is straightforward, but it is important to allocate proper space for your `encryptedData` pointer. The output length of `CCCrypt()` will never be more than the input plus one additional block. Block size is based on the encryption algorithm being implemented.

In the example, if you don't receive a successful result, the method returns `nil`. Depending on your requirements, you may need to implement additional error handling. `CCCrypt()` has three possible error results: `kCCBufferTooSmall`, `kCCAlignmentError`, and `kCCDecodeError`. `kCCBufferTooSmall` or `kCCDecodeError` are the most common. `kCCBufferTooSmall` indicates that the output buffer, `encryptedData` in this example, is not of sufficient size. `kCCDecodeError` can be received only during decryption operations and is most likely related to an invalid key.

With the encryption methods complete, Listing 6-14 covers how the application uses encryption in practice. Listing 6-14 also details how MAC generation fits into the process.

LISTING 6-14: AES Encryption in Practice (/App/Mobile-Banking/FundsTransferOperation.m)

```
...
    // build our transfer data
    NSDictionary *transfer = [NSDictionary dictionaryWithObjectsAndKeys:
                              _toAccount, @"toAccount",
                              _fromAccount, @"fromAccount",
                              date, @"transferDate",
                              _transferNotes, @"transferNotes",
                              amount, @"amount", nil];

    // create the json representation of our transfer data using ios5 API
    NSError *error = nil;
    NSData *transferData = [NSJSONSerialization dataWithJSONObject:transfer
                                                           options:0 error:&error];
    NSString *transferString =
        [[NSString alloc] initWithData:transferData
                              encoding:NSUTF8StringEncoding];

    // generate our initialization vector
    NSData *iv = [Utils blockInitializationVectorOfLength:kCCBlockSizeAES128];

    // because we generate random bytes,
    // it may not be proper UTF8 encoding.
    // because of this, we can't just init a string with data. instead,
    // we encode it for transmission. the IV is then decoded on the service
    // and used in the decryption process
    NSString *ivString = [iv base64Encoding];

    // encrypt our transfer data using AES and a randomly generated
    // IV (this IV needs to be what we send to the service)
    NSString *encryptedString =
        [transferString encryptedWithAESUsingKey:kAESEncryptionKey
                                           andIV:iv];

    // calculate our message authentication code
    NSString *macCandidate = [NSString stringWithFormat:@"%@%@%@%@",
```

```
                                _toAccount,       // to account
                                _fromAccount,     // from account
                                 amount,          // amount
                                _transferDate];   // transfer date
    NSString *mac = [macCandidate hmacWithKey:kMACKey];

    // construct our payload
    NSMutableDictionary *payload = [NSMutableDictionary
                                    dictionaryWithObjectsAndKeys:
                                    ivString, @"iv",
                                    mac, @"mac",
                                    encryptedString, @"payload", nil];

...
```

After you create your funds transfer data structure and generate an initialization vector, you need to encrypt the transfer instructions and calculate a MAC. Although the MAC generation was covered in the last section, it has been included in Listing 6-14 to provide additional context.

Until now, the initialization vector has been provided to you as an input. In Listing 6-14, you are responsible for generating the initialization vector prior to encryption. Listing 6-15 covers how to use the `SecRandomCopyBytes()` function, provided by the Security Framework (discussed in the next section), to create a random array of bytes that are cryptographically secure.

LISTING 6-15: Initialization Vector Creation Using SecRandomCopyBytes() (/App/Mobile-Banking/Utils.m)

```
+ (NSData*)blockInitializationVectorOfLength:(size_t)ivLength {
    // default to AES block size
    if (ivLength == 0) {
        ivLength = kCCBlockSizeAES128;
    }

    NSMutableData *iv = [NSMutableData dataWithLength:ivLength];

    int ivResult = SecRandomCopyBytes(kSecRandomDefault,
                                      ivLength,
                                      iv.mutableBytes);

    if (ivResult == noErr) {
        return iv;
    }

    return nil;
}
```

If you do not specify a vector length, this method uses the AES block size as the default. After generating the vector, ensure you did not receive an error and return the result. The only error is `-1`, which indicates a failure. To use `SecRandomCopyBytes()`, you must include the Security Framework in your project.

When transmitting encrypted data from one system to another for decryption, you must know how the various parameters are utilized. `CCCrypt()` adds `NULL` padding to initialization vectors smaller

than the block size defined by the specified algorithm. This could potentially cause integration issues depending on how the receiving system handles padding.

Listing 6-16 covers decryption and MAC verification in PHP using the encrypted payload generated in Listing 6-14. The decryption process is identical for AES and Triple-DES with the exception of the algorithm and key passed to the `mcrypt_decrypt ()` function call.

LISTING 6-16: AES Decryption Using PHP (/Service/index.php)

```
// retrieve the request payload
$postData = json_decode(@file_get_contents('php://input'));
$inboundMac = $postData->mac;
$iv = $postData->iv;
$payload = $postData->payload;

// decrypt the payload
$decrypted = mcrypt_decrypt(MCRYPT_RIJNDAEL_128,
                            $AES_KEY,
                            base64_decode($payload),
                            MCRYPT_MODE_CBC,
                            $iv);
$decLength = strlen($decrypted);
$padding = ord($decrypted[$decLength-1]);
$decrypted = substr($decrypted, 0, -$padding);

// decode decrypted payload, split into components
$decryptedPayloadJSON = json_decode($decryptedPayload);
$toAccount = $decryptedPayloadJSON->toAccount;
$fromAccount = $decryptedPayloadJSON->fromAccount;
$amount = $decryptedPayloadJSON->amount;
$transferDate = $decryptedPayloadJSON->transferDate;
$transferNotes = $decryptedPayloadJSON->transferNotes;

// grab toAccount, fromAccount, amount, transferDate (in that order)
// and create message auth code (hmac)
$macCandidate = $toAccount.$fromAccount.$amount.$transferDate;
$derivedMac = hash_hmac("sha256", $macCandidate, $HMAC_KEY);

// validate inbound hmac matches your derived value
if ($inboundMac != $derivedMac) {
    sendAPIResponse
        (400,
         json_encode(buildErrorResponse("Message Integrity Error"))
        );
        return;
}

// here you would perform your actual transfer and
// validate it was successful prior to issuing 200

sendAPIResponse(200);
return;
```

The following snippet demonstrates the algorithm change needed to decrypt a Triple-DES encrypted blob.

```
...
// decrypt the payload
$decrypted = mcrypt_decrypt(MCRYPT_3DES,
                            $DES_KEY,
                            base64_decode($payload),
                            MCRYPT_MODE_CBC,
                            $iv);
$decLength = strlen($decrypted);
$padding = ord($decrypted[$decLength-1]);
$decrypted = substr($decrypted, 0, -$padding);

// decode decrypted payload, split into components
...
```

With encryption covered, discussing decryption, response interpretation, and payload decryption should go quickly. Decryption will be covered quickly because it is literally a single line of code, and the following listings are similar to Listing 6-12 and Listing 6-13, respectively. `CCCrypt()` can be used for both encryption and decryption operations and the intent is specified using the first parameter.

The methods outlined in Listing 6-17 and Listing 6-18 should look familiar; they are nearly identical to their encryption counterparts. The only difference is that you instruct `CCCrypt()` to perform a decrypt operation instead of an encrypt operation.

LISTING 6-17: AES Decryption (/App/Mobile-Banking/NSData+Encryption.m)

```
- (NSData*)decryptedWithAESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;
    NSMutableData * decryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSizeAES128];

    CCCryptorStatus result = CCCrypt(kCCDecrypt,
                                     kCCAlgorithmAES128,
                                     kCCOptionPKCS7Padding, // CBC Padding
                                     keyData.bytes,
                                     keyData.length,
                                     iv.bytes,
                                     self.bytes,
                                     self.length,
                                     decryptedData.mutableBytes, //data out
                                     decryptedData.length,
                                     &dataMoved);  // total data moved

    if (result == kCCSuccess) {
        decryptedData.length = dataMoved;
        return decryptedData;
    }

    return nil;

}
```

LISTING 6-18: Triple-DES Decryption (/App/Mobile-Banking/NSData+Encryption.m)

```
- (NSData*)decryptedWith3DESUsingKey:(NSString*)key andIV:(NSData*)iv {

    NSData *keyData = [key dataUsingEncoding:NSUTF8StringEncoding];

    size_t dataMoved;
    NSMutableData *decryptedData =
        [NSMutableData dataWithLength:self.length + kCCBlockSize3DES];

    CCCryptorStatus result = CCCrypt(kCCDecrypt,
                                     kCCAlgorithm3DES,
                                     kCCOptionPKCS7Padding, // CBC Padding
                                     keyData.bytes,
                                     keyData.length,
                                     iv.bytes,
                                     self.bytes,
                                     self.length,
                                     decryptedData.mutableBytes,//data out
                                     decryptedData.length,
                                     &dataMoved);  // total data moved

    if (result == kCCSuccess) {
        decryptedData.length = dataMoved;
        return decryptedData;
    }

    return nil;

}
```

The application implements the list account functionality of the application to illustrate client-side decryption. After a user authenticates successfully, the application immediately issues a request to retrieve the user's accounts. The list of accounts includes name, number, and balance, and transmits to the application in encrypted form. You can decrypt the account list, validate message integrity, and create account objects for display in the view layer.

Listing 6-19 covers creating the service tier response, which includes encrypting the payload using AES and calculating a MAC. The structure created in Listing 6-19 should be familiar to you from the discussion on encryption.

LISTING 6-19: Payload Generation and Encryption Within PHP (/Service/index.php)

```
// create array of bank accounts for the user
$accounts = array();

// fill our accounts array with data
...

// generate the IV
$iv = generateInitVectorOfLength(16);
```

```
// create MAC from accounts data
$mac = hash_hmac("sha256", json_encode($accounts), $hmacKey);

// encrypt our payload
$encryptedPayload = mcrypt_encrypt(MCRYPT_RIJNDAEL_128,
$AES_Key,
addEncryptionPadding(json_encode($accounts)),
MCRYPT_MODE_CBC,
$iv);
// generate service response
$response['iv'] = $iv;
$response['mac'] = $mac;
$response['payload'] = base64_encode($encryptedPayload);
...
```

Although Listing 6-19 covers AES encryption, Triple-DES encryption is similar. You simply need to change the initialization vector length from 16 to 8, update the encryption algorithm from `MCRYPT_RIJNDAEL_128` to `MCRYPT_3DES`, and change the key (`$AES_Key`).

Listing 6-20 details how to interpret and decrypt the response generated in Listing 6-19 within your application.

LISTING 6-20: Handling AES Encrypted Responses (/App/Mobile-Banking/GetAccountsOperation.m)

```
- (void)connectionDidFinishLoading:(NSURLConnection *)connection {

    // unpack service response
    NSError *error = nil;
    NSDictionary *response =
        [NSJSONSerialization JSONObjectWithData:self.responseData
                                        options:0
                                          error:&error];

    // decrypt the payload
    NSString *inboundMAC = [response objectForKey:@"mac"];
    NSData *ivData =
        [[response objectForKey:@"iv"]
            dataUsingEncoding:NSUTF8StringEncoding];

    NSString *encryptedResponse = [response objectForKey:@"payload"];
    NSString *decryptedResponse =
       [encryptedResponse decryptedWithAESUsingKey:kAESEncryptionKey
                                             andIV:ivData];

    if (decryptedResponse != nil) {
        // create JSON array of account info
        NSError *accountError = nil;
        NSArray *accounts =
            [NSJSONSerialization JSONObjectWithData:
                decryptedResponse dataUsingEncoding:NSUTF8StringEncoding]
                                            options:0
```

continues

LISTING 6-20 *(continued)*

```
                                                error:&accountError];

        // validate the MAC
        NSString *generatedMAC = [decryptedResponse hmacWithKey:kMACKey];
        if ([inboundMAC isEqualToString:generatedMAC]) {

            // validation passed, create accounts
            for (NSDictionary *accountData in accounts) {
                Account *account = [[Account alloc] initWithData:accountData];
                [[Model sharedModel].accounts addObject:account];
            }

        } else {
     // post error notification
        }
    } else {
 // post error / unable to decrypt notification
    }
    ...
}
```

This section covered how to encrypt and decrypt data using native Objective-C libraries. Although data encryption plays an important role in your users' security, cryptography can be subject to Federal export controls. These controls come into play during the AppStore submission process. Figure 6-3 depicts the AppStore encryption confirmation screen. For enterprise-deployed applications, you should consult your legal department.

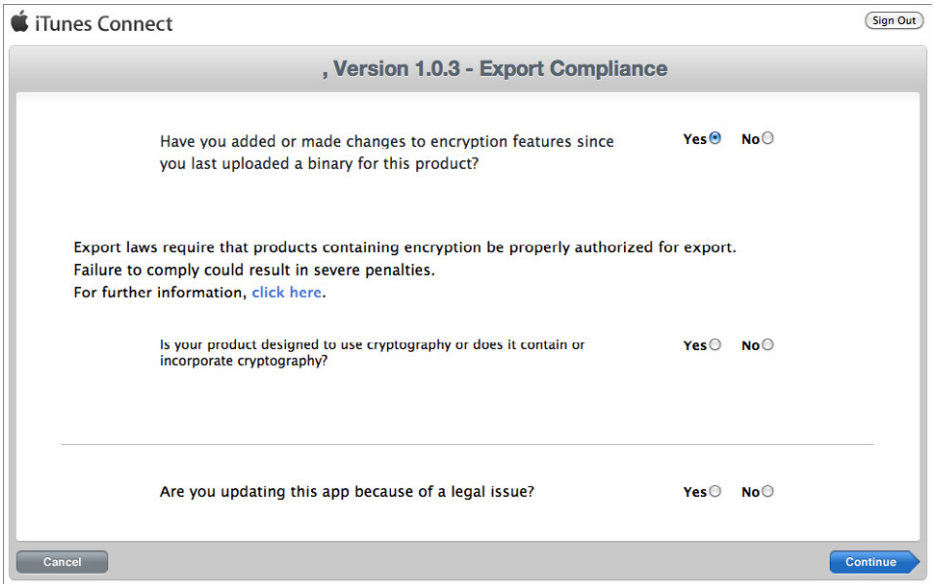


FIGURE 6-3

STORING CREDENTIALS SECURELY ON THE DEVICE

Now that the application can communicate securely with the service tier, it needs to store information securely on the device. Apple provides the Keychain Services API as part of the Security Framework to do just that. Keychain is a mechanism to securely store small amounts of data such as passwords, keys, certificates, and identities on the device. Keychain is not intended for general-purpose encryption and data storage, but items that require protection such as passwords and private keys will be stored in an encrypted manner. Items such as certificates, which do not require that level of protection, will not be encrypted for storage. This section discusses Keychain concepts, and a more in-depth Keychain implementation is covered in Chapter 11, "Inter-App Communication."

In iOS, each application has access to the Keychain items it creates without having to request permission. This differs from traditional Mac development in which applications can access any Keychain items the user approves. Keychain data is technically stored outside of the application sandbox, which allows data to persist through an application deletion event. iOS Keychain rights are dependent on the provisioning profile used to sign the application. As your application progresses through its version lifecycle, it is important to use the same provisioning profile.

An application's Keychain can contain any number of items, each of which consists of data to store and a set of attributes. The attributes for each Keychain item depend on the item class chosen during storage. There are a number of common item attributes between item classes. For a complete list, refer to Apple's documentation, but Table 6-3 outlines the attributes that can be set by the application.

TABLE 6-3: Editable Keychain Item Attributes

ITEM ATTRIBUTE	DESCRIPTION
`kSecAttrAccessible`	Indicates when this item can be accessed; discussed further in the chapter text
`kSecAttrAccessGroup`	Indicates which access group the item belongs to; discussed further in the chapter text
`kSecAttrDescription`	User-visible string describing item
`kSecAttrComment`	User-editable comment for item
`kSecAttrCreator`	Item's creator as unsigned integer representation of four-character code
`kSecAttrType`	Item's type as unsigned integer representation of four-character code
`kSecAttrLabel`	User-visible label for item
`kSecAttrIsInvisible`	Boolean indicating whether item should be displayed
`kSecAttrIsNegative`	Indicates whether there is a valid password for this item
`kSecAttrAccount`	Account name associated with this item; included in Generic and Internet Password classes

continues

TABLE 6-3 *(continued)*

ITEM ATTRIBUTE	DESCRIPTION
`kSecAttrService`	Service name associated with this item; included in Generic Password class
`kSecAttrGeneric`	User-defined attribute included in Generic Password class
`kSecAttrSecurityDomain`	Internet security domain of item; included in Internet Password class
`kSecAttrServer`	Server domain or IP address of item; included in Internet Password class
`kSecAttrProtocol`	Protocol for this item; included in Internet Password class
`kSecAttrAuthenticationType`	Authentication scheme for item; included in Internet Password class
`kSecAttrPort`	Internet port number; included in Internet Password class
`kSecAttrPath`	Path, typically URL path component; included in Internet Password class
`kSecAttrApplicationLabel`	Label for item used to look up key programmatically; included in Key class
`kSecAttrIsPermanent`	Boolean indicating whether key is stored permanently; included in Key class
`kSecAttrKeyType`	Algorithm associated with key; included in Key class
`kSecAttrKeySizeInBits`	Total number of bits in key; included in Key class
`kSecAttrEffectiveKeySize`	Effective number of bits in key; included in Key class
`kSecAttrCanEncrypt`	Boolean indicating whether key can be used for encryption; included in Key class
`kSecAttrCanDecrypt`	Boolean indicating whether key can be used for decryption; included in Key class
`kSecAttrCanDerive`	Boolean indicating whether key can be used to derive another key; included in Key class
`kSecAttrCanSign`	Boolean indicating whether key can be used to create digital signatures; included in Key class
`kSecAttrCanVerify`	Boolean indicating whether key can be used to verify digital signature; included in Key class
`kSecAttrCanWrap`	Boolean indicating whether key can be used to wrap another key; included in Key class
`kSecAttrCanUnwrap`	Boolean indicating whether key can be used to unwrap another key; included in Key class

Two important attributes to note during the creation process, which are common to *all* classes, are `kSecAttrAccessible` and `kSecAttrAccessGroup`. `kSecAttrAccessible` allows you to indicate when the application can access the Keychain item. You should use the most restrictive option that still allows your application to meet its stated purpose. Table 6-4 lists all possible values for the `kSecAttrAccessible` attribute. At a minimum, you should consider setting `kSecAttrAccessible` to a value ending with `ThisDeviceOnly`, which restricts the transfer of the Keychain item to a new device. `kSecAttrAccessGroup` indicates which access group a Keychain item belongs to. Applications can belong to multiple access groups as defined in the `Entitlements.plist` file from Chapter 11. Multiple access groups can be used to further compartmentalize Keychain data. An access group can also be used to share data between applications. The example in Chapter 11 touches on this in more detail, including the `Entitlements.plist` file.

TABLE 6-4: kSecAttrAccessible Attribute Values

POSSIBLE VALUE	DESCRIPTION
`kSecAttrAccessibleWhenUnlocked`	Item data can be accessed whenever the device is unlocked. Recommended when items are needed while application is in the foreground.
`kSecAttrAccessibleAfterFirstUnlock`	Item data can be accessed after the first unlock following a restart.
`kSecAttrAccessibleAlways`	Item data is always accessible. It is not recommended that developers use this attribute; it is intended for system use.
`kSecAttrAccessibleWhenUnlockedThisDeviceOnly`	Similar to `kSecAttrAccessibleWhenUnlocked` except that item data cannot be migrated to a new device.
`kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly`	Similar to `kSecAttrAccessibleAfterFirstUnlock` except that item data cannot be migrated to a new device.
`kSecAttrAccessibleAlwaysThisDeviceOnly`	Item data is always accessible. Similar to `kSecAttrAccessibleAlways` it is not recommended because it's intended for system use. Items with this value cannot be migrated to a new device.

Given how easy it is to implement and its flexibility for storage, Generic Password tends to be the most common for developers to start with while they learn the Keychain concepts. The Generic Password class is an ideal location for securely storing non-Internet passwords, such as authentication tokens like those used in this chapter's service tier. The Generic Password class is also where you may choose to store an indicator to detect previous installations of an application.

Interacting with Keychain Services requires an understanding of how to structure Keychain searches. The most important step is specifying the appropriate item class value for the `kSecClass` attribute. This is set during the creation of a Keychain item and compartmentalizes searches. `kSecAttrAccessGroup` is also important to the search structure. As previously stated, an application can contain multiple access groups. If your application specifies the wrong access group to search, it will not be able to locate the Keychain item you are seeking.

When creating an entry in Keychain, it is a best practice to first determine whether the item already exists and respond accordingly with an add or update. The remaining actions (retrieve, update, and delete) accept a `query` parameter, which is an instance of `CFDictionaryRef`. The action is performed on each Keychain item that matches the query provided. The `query` parameter can include a combination of any number of item attributes (a subset of which is outlined in Table 6-3) and the search attributes defined in Table 6-5.

TABLE 6-5: Predefined Keychain Search Attributes

SEARCH ATTRIBUTE	DESCRIPTION
`kSecMatchPolicy`	Restricts certificates and identities must conform to this policy. Value is `SecPolicyRef` object.
`kSecMatchIssuers`	Restricts certificates and identities where their certificate chain contains one or more of the issuers in a `CFArray` of X.509 names.
`kSecMatchEmailAddressIfPresent`	Restricts certificates and identities where they contain the address specified or do not contain an address.
`kSecMatchSubjectContains`	Restricts certificates and identities to those where the subject contains the specified `CFStringRef`.
`kSecMatchCaseInsensitive`	Specifies that search must match case sensitivity to be returned.
`kSecMatchTrustedOnly`	Boolean value that restricts certificates which can be verified back to a trusted anchor. When false, both trusted and untrusted certificates will be returned.
`kSecMatchValidOnDate`	Restricts keys, certificates, and identities that are valid on the specified date. Pass `kCFNull` to use the current date.
`kSecMatchLimit`	Specifies the number of results that can be returned. Default is `kSecMatchLimitOne` with the other predefined constant being `kSecMatchLimitAll`.
`kSecMatchLimitOne`	Restricts results to the first matching item found.
`kSecMatchLimitAll`	Specifies that ALL matching results can be returned.

> **NOTE** *As stated earlier in this section, Chapter 11 covers integrating Keychain storage into your application, including how to share data between multiple applications.*

SUMMARY

Securing network communication is extremely important and can take many forms, such as validating that your users are communicating with the correct servers or authenticating users before granting them access to your systems. You may also choose to encrypt all, or part of, your network traffic using ciphers such as AES and Triple-DES. To ensure your requests have not been manipulated during transmission, you may use a MAC, or cryptographic hash, which requires transmission planning with your service tier. After you have securely communicated the data, you may choose to store it securely using the devices Keychain.

Even though Apple has provided these security libraries, you as a developer must still practice a little common sense when it comes to protecting any valuable personal information stored in your app. This becomes even more essential, and often contractually or legally obligated, when dealing with data from financial institutions, health organizations, or government agencies. Transparency with your users is paramount, and you should always be clear about what information you access and transmit on a user's behalf. Store only the information absolutely necessary for your app to meet its stated purpose, and always transmit that information in an appropriate and secure manner.

The next chapter discusses patterns for request optimization through HTTP caching, compression, and pipelining.

7

Optimizing Request Performance

WHAT'S IN THIS CHAPTER?

- Understanding network bandwidth and latency
- Reducing request bandwidth with compression
- Reducing request latency with pipelining
- Minimizing request bandwidth with caching

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code is in the Chapter 07 download and individually named according to the names throughout the chapter.

As you have seen in earlier chapters, basic networking from an iOS device is powerful and relatively easy. iOS has a full-featured API that supports many industry-standard protocols; however, even if you develop a phenomenal app, the network communication may perform suboptimally based on the device's connection to the outside world.

This chapter teaches you the dimensions by which network performance is measured, and you can use that knowledge to improve your app's network communication. You learn best practices that reduce bandwidth consumed by your app, improve the responsiveness, and even prolong the battery life of the device running the app.

MEASURING NETWORK PERFORMANCE

This section reviews at a high level the metrics used to describe network performance. Although there are many metrics used to describe the performance of a wireless network, this section focuses on the three most important ones: bandwidth, latency, and power consumption.

Network Bandwidth

The most common metric used to describe wireless network performance is bandwidth. *Network bandwidth* in digital wireless communications is best described as the number of bits per second that a communications channel may carry between two endpoints. Modern wireless networks boast impressive theoretical bandwidth capabilities, but keep in mind that the bandwidth numbers quoted by carriers and network equipment providers are usually the theoretical maximums for the technology, and the real-world bandwidth seen by a networked device can vary greatly from that maximum. Figure 7-1 shows the theoretical capacity of current wireless technologies on a logarithmic scale.

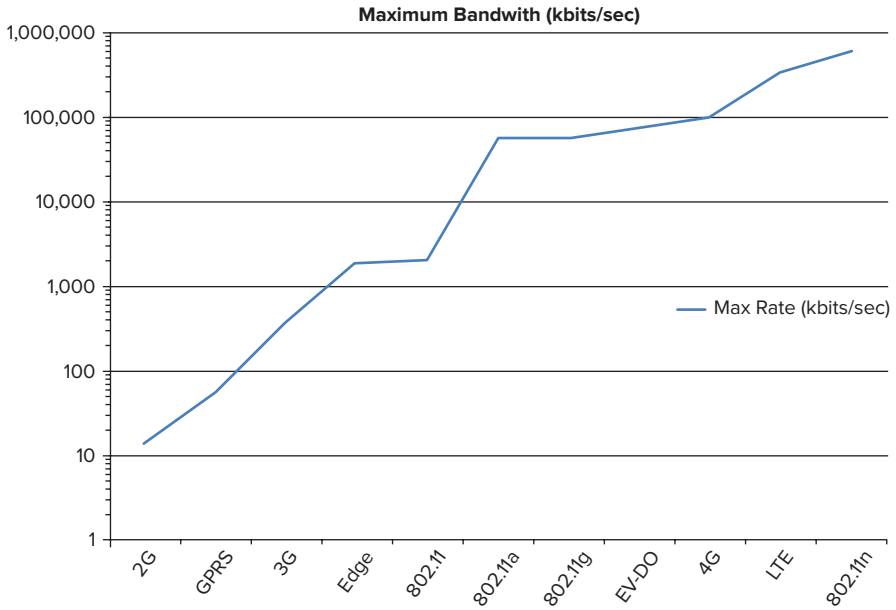


FIGURE 7-1

There are many reasons why the bandwidth your device experiences differs so much from the theoretical maximum. Chief among them is the fact that wireless network bandwidth is a shared commodity. Even if a wireless technology boasts a high bandwidth, that channel may be shared by thousands of wireless devices each competing for airtime. Consequently, the bandwidth consumption and transmission activity of each of these devices can impact how much bandwidth is available to your device. Figure 7-2 illustrates the route that a request from your app must take to arrive at the destination server. Along every step of the route, your request must contend for bandwidth with thousands of other data packets.

Consider a situation in which your app has sole use of an LTE base station in the middle of Kansas, but that base station may have a small back haul circuit that connects it to the Internet at large. You

may get an excellent data rate to the tower, but the back haul restricts the usable bandwidth. The maximum speed of any network connection is the speed of the slowest link along the communication path. Even if there is sufficient back haul bandwidth, carrier network equipment applies a quality of service (QoS) priority value to every data packet traversing their network. That equipment usually deprioritizes simple data packets below voice packets; therefore, your network request must wait behind more time-sensitive voice data.

Other factors such as distance to the base station, atmospheric conditions, and network interference can also impact the bandwidth seen by the device. These myriad factors are unavoidable, and the best response you can take is to acknowledge that your app will receive a miniscule fraction of available bandwidth, and it needs to make the best use of what it is given. Some suggestions for how to do this are discussed later in the chapter in the "Reducing Request Bandwidth" section.

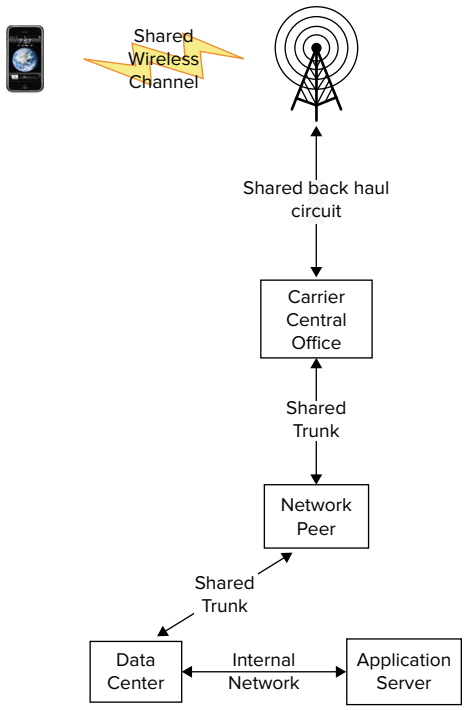


FIGURE 7-2

Network Latency

A second measure of network performance is *network latency*, which is the amount of time it takes for a network packet to make a round trip between endpoints. Wireless carriers rarely quote latency figures for their networks, but latency can have a dramatic effect on the perceived performance of your application. Like bandwidth, there are many factors that impact the latency experienced by your application. The primary factor is the inherent latency of the wireless network technology used to connect the device to the rest of the world. Figure 7-3 shows the best-case latency figures for common network standards.

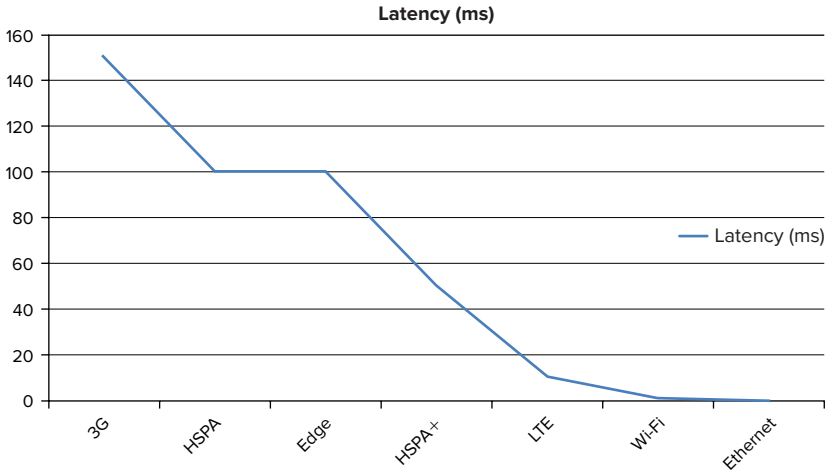


FIGURE 7-3

Latency is also dependent on the specific route taken by the packet as it moves between the device and the destination server. Refer to Figure 7-2 to see an example route. Many other factors have a noticeable impact on influencing network latency and are outlined in the following list:

- Other devices sharing the same base station may cause your packets to be delayed
- QoS prioritization of network packets may make your packets wait for other higher priority packets
- Latency added by back haul channels can increase the latency experienced by your application
- Server response times may add significant time to your packet latency

There are many tools available for iOS devices to measure latency between the device and a network host, such as SpeedTest.net's app. If you use this or a similar tool, you immediately notice the wide variance in latency times between individual requests on the same network, even those made back-to-back. All these factors create a situation that is difficult to measure consistently and accurately.

The effect of latency on your application is primarily in the area of responsiveness. The amount of time a user must wait for a response to an action may vary between milliseconds to seconds depending on the network. This means that you must test your application on a variety of networks early and often in the development process. Chapter 9, "Testing and Manipulating Network Traffic," includes information on using Max OS X's Network Link Conditioner to simulate a variety of network conditions with the iOS Simulator. The impact of latency is inversely proportional to the size of the requests generated by the application. (The exception to this guideline is for requests less than 1,500 bytes where the impact of network latency is identical down to 0 bytes.) If your application sends requests with smaller payloads, then latency will have a greater effect. However, if your application is streaming video content or some other activity that requires large payloads, then latency will be of less concern than overall bandwidth.

You'll see what measures you can take to reduce latency periods later in the chapter in the "Reducing Request Latency" section.

Device Power

iOS devices are usually run on battery power, and every action taken by your app consumes power. The primary consumers of power on an iOS device are as follows:

- Display and display backlight
- Location services
- Wi-Fi radio
- WWAN (cellular) radio
- Graphics processor
- CPU

- Bluetooth Radio
- Audio processor

With the exception of the display, your application has control over most of these variables. As you optimize your program for better network performance, you have a secondary benefit to improve the power consumption profile of your application. If your application is minimizing the amount of data that it transmits, optimizing the use of existing TCP connections, and avoiding unnecessary requests, it can reduce the amount of time the device must keep one or both of its radios powered on.

Another way to reduce battery usage by your application is to use encryption intelligently. A recent best practice in web applications has been to encrypt all data, and this is good advice on applications that use desktop or laptop machines that are usually tethered to a power source. However, on mobile devices with limited power, you should consider what requests must be encrypted. Images and other publically available assets may not require encryption when in transit. The use of encryption increases the CPU load and may activate hardware resources that would normally be powered down.

OPTIMIZING NETWORK OPERATIONS

Admitting that there may be problems with network performance is the first step in addressing difficulties with network performance. If you don't take steps to address network performance, you may leave your users with an unpleasant experience. This section covers techniques you can utilize to reduce the bandwidth used by your application by compressing the content of HTTP bodies, measures you can take to minimize the impact of high-latency connections, and methods you can try to avoid using the network altogether by employing response caching.

Reducing Request Bandwidth

In the early days of computing, it was not uncommon to interconnect computers at 300 bits per second. At this low speed, every bit transmitted was chosen with great care, and computer scientists devised compact and sparse protocols that preserved bandwidth but were difficult to implement, validate, and support.

As modern network speeds increased into the millions of bits per second, each individual bit became almost irrelevant, and the protocols used over these connections became more verbose. These verbose protocols were easier to maintain and manage. Protocols such as SOAP consume excessive amounts of bandwidth but provide enough information to make each message almost self-describing.

With the advent of shared wireless networking such as GSM or CDMA, engineers have to reconsider the bandwidth used by the protocols they select for an application and apply methods to reduce the bandwidth used without sacrificing reliability or maintainability of the application. There are several ways to reduce the amount of bandwidth consumed by an app:

- **Use an efficient data interchange format —** Select an efficient encoding for data between the client and server. Chapter 4, "Generating and Digesting Payloads," addresses the decision criteria for selecting the right payload format.

- **Use precompressed data where possible —** Compress or scale media assets such as audio, video, and images with special purpose algorithms to suit the channel and device.
- **Compress each request or response payload —** Compress text payloads to provide significant bandwidth savings with little impact on server or client code.

You can actually enable payload compression for nonmedia payloads by compressing server responses or client requests. The method for compressing a response is significantly different from compressing a request. The next two sections describe this method of compression for both responses and requests.

USING JSON AND XML FOR REQUEST AND RESPONSE BODIES

JSON and XML are common data encodings used for request or response bodies. The efficiency of an encoding scheme is highly dependent upon the data being encoded, but JSON is usually a more efficient scheme. Methods exist, such as CJSON and EXI, to perform encoding specific compression on either JSON or XML. These compressed encoding schemes require custom server and client modules to compress and decompress the data and may prevent reuse of the service. For example, if you develop a service that uses CJSON you may be able to consume the data on an iPhone, but you will not be able to reuse the service for a mobile web application.

NOTE *You can find official specifications for HTTP response and request formats and compression in the following RFC documents:*

- *RFC 2616 – Hypertext Transfer Protocol:* `http://www.ietf.org/rfc/rfc2616.txt`
- *RFC 1951 – DEFLATE Compressed Data Format Specification:* `http://www.ietf.org/rfc/rfc1951.txt`
- *RFC 1952 – GZIP file format specification:* `http://www.ietf.org/rfc/rfc1952.txt`
- *RFC 1950 – ZLIB Compressed Data Format Specification:* `http://www.ietf.org/rfc/rfc1950.txt`

Response Compression

Response payload compression is the easiest form of HTTP payload compression to implement. An HTTP response is composed of the headers and body returned to the client in response to a previous HTTP request. Response compression applies a data compression algorithm to the response body and leaves the HTTP headers intact.

Response compression of text data can have a dramatic impact on the size of the data returned to the client. With JSON or XML (see Chapter 4 for details on creating and processing JSON and XML responses) the compressed payload can be less than 10 percent the size of the original payload, but your results may vary depending on the succinctness of the original payload. If the original source uses JSON with one-character field names and has all whitespace removed, you see more limited results compared to XML formatted for printing with lengthy element names. Typically, the larger the payload the greater the compression ratio. Some small payloads may experience a net increase in size due to compression lookup tables.

In iOS HTTP payload, compression is enabled by default for all HTTP `NSURLConnection` requests. The received payload is automatically decompressed and presented to your code in its original format. The computational overhead of decompression is substantially less than the communication overhead of transmitting ten times as many bytes; therefore, activating response compression is almost always beneficial.

`NSURLConnection` adds the following HTTP header to every request by default:

```
Accept-Encoding: gzip, deflate
```

The `Accept-Encoding` header informs the server that the client will accept payloads compressed with either gzip or DEFLATE compression, but it is the server's choice whether to compress its response. Thus, the key to enabling the performance wins with response payload compression is to configure the server terminating the HTTP to support compression.

> **NOTE** *Some browsers do not handle DEFLATE compression correctly so the most common compression used is gzip.*

For example, the process to configure the Apache web server involves loading a compression module and activating an output filter for specific document types. First, your Apache configuration needs to load two modules.

```
LoadModule filter_module library-path/mod_filter.so
LoadModule deflate_module library-path/mod_deflate.so
```

> **NOTE** *The value of the library-path will vary depending on the installation of Apache.*

The `filter_module` is a common module and is probably already loaded. The `deflate_module` is less common but is part of the standard Apache installation for Linux, OS X, and Windows.

> **WARNING** *The name of the* `deflate_module` *is misleading because it supports gzip compression but not DEFLATE compression.*

Next you must define the content to compress. The brute force approach is to add an output filter applied to all content. The following snippet applies a global DEFLATE output filter:

```
SetOutputFilter DEFLATE
```

This is not a recommended approach unless you know that the web server is serving only text data that can benefit from compression. Applying compression to precompressed content, such as images, audio, and video, can consume CPU resources to perform the compression while having little to no positive impact on the size of the payload. A more targeted approach is to add output filters for only the content types that can benefit from compression. The following code applies compression to several common content types:

```
AddOutputFilterByType DEFLATE text/plain
AddOutputFilterByType DEFLATE text/xml
AddOutputFilterByType DEFLATE application/xhtml+xml
AddOutputFilterByType DEFLATE text/css
AddOutputFilterByType DEFLATE application/xml
AddOutputFilterByType DEFLATE application/atom_xml
AddOutputFilterByType DEFLATE application/x-javascript
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE application/json
```

Enabling response compression should be transparent to other applications and browser-based users because the client software, either application or browser, must explicitly indicate that it accepts a compress payload with the `Accept-Encoding` header, which most modern browsers do. The only downside of compression is that it makes the payload difficult to read and debug when using a network sniffer to analyze traffic during development. A recommended approach is to enable compression in the test environments but not in development environments.

Many other HTTP servers, including Microsoft's IIS, support response compression. For information on enabling compression in IIS see: `http://www.iis.net/ConfigReference/system.webServer/httpCompression`. Many load balancer appliances, such as BIG-IP appliances, support HTTP compression augmented by hardware-based compression subsystems.

If you do find a reason to disable payload compression, the app can prevent it by clearing the automatically set `Accept-Encoding` header. The following code is an example of clearing this header.

```
NSMutableURLRequest *request = [[[NSMutableURLRequest alloc]
                                    initWithURL:url
                                    cachePolicy:NSURLCacheStorageAllowed
                                timeoutInterval:20] autorelease];
[request addValue:@"" forHTTPHeaderField:@"Accept-Encoding"];
```

The response to the request cannot be compressed by the server. Response compression is an easy way to optimize your application's use of network bandwidth and requires only minimal changes to the service tier.

Request Compression

Unlike response compression, request compression is much more complicated to implement because it requires significant client-side and server-side implementations. When performing request compression, an HTTP client applies some type of data compression on the request body prior to sending it to the server. Request compression is not well supported in web browsers because the browser cannot know if the destination server can support decompression of the request. If the server does not understand the compression scheme, the request is discarded, and the client app never gets a response.

Because of this intractable problem, request compression requires a prior arrangement between the application and server developers. An alternative is to develop a scheme in which the iOS application queries the server to determine if compression is currently supported and then alter its behavior based on the server response.

Request compression can be of significant benefit to mobile applications because wide area wireless transmission speeds are usually asymmetric, providing greater bandwidth to data sent to the device than to data originating from the device. The asymmetry is implemented because most web traffic is usually asymmetric. If your application defies the standard asymmetric pattern, you should strongly consider request compression. For example, if your application collects data for later upload to a server, it can benefit from compressing that uploaded payload. The sample application written for this chapter includes a simple, moderately sized XML file that is 40 KB in size. Without compression, that file consumes 80 KB of bandwidth when transmitted to the server and echoed back to the device. With both request and response compression enabled, the data consumes only 12 KB for the same round trip.

To set up request compression you must first define an input filter in the web server. This example shows how to define this for Apache web servers.

> **WARNING** *Apache does not decompress data before sending it through a resource filter such as the PHP or* `mod_jk` *modules. Therefore, if you pass compressed data to a web app via a resource filter, the destination web app is responsible for the decompression of the payload.*

> **NOTE** *As with response compression, client apps should not waste CPU time compressing content such as PDFs, encrypted data, images, audio, or video that is already compressed. However, Base64 data representing pre-compressed data will often benefit from request compression. For example, if you must upload a JPEG file in Base64 format, you can compress the Base64 data to achieve approximately a 30 percent reduction in size over the uncompressed Base64 data.*

The same modules used for response compression also perform request compression in Apache. The following configuration snippet loads the required modules.

```
LoadModule filter_module library-path/mod_filter.so
LoadModule deflate_module library-path/mod_deflate.so
```

> **NOTE** *The value of the library-path will vary depending on the installation of Apache.*

The next step in Apache configuration is to define an input filter for the DEFLATE module. The following code snippet defines an input filter and a CGI alias.

```
SetInputFilter DEFLATE
SetOutputFilter DEFLATE
ScriptAlias /cgi/ <html-directory>/cgi-bin/
```

> **NOTE** *The location of the html-directory will vary depending upon the configuration of your host system.*

If a request arrives at the HTTP server with a header of `Content-Encoding: gzip` the HTTP server attempts to decompress the request body and passes it to the next filter in the filter chain. For demonstration purposes, the sample request compression app comes with a simple Perl script (see Listing 7-1) that echoes the received payload back as the response body. The script does not care if the received payload were compressed.

LISTING 7-1: Decomp.cgi

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "Hello, World.\n";
print "Body=\n";
foreach (<>) {
        print;
}
```

Next you add the compression code to your iOS app. It needs to both compress the payload and add a `Content-Encoding` header to the request. The example compression code uses the `libz.dylib` framework, which requires your project to link against that framework before compiling any compression code. The code in Listing 7-2 shows the method used in the example program to compress the body of the HTTP request:

LISTING 7-2: PostCompress/CompressRequest.m

```
- (NSData *)compressNSData:(NSData *)myData {
```

```
    NSMutableData *compressedData = [NSMutableData dataWithLength:16384];

    z_stream compressionStream;
    // setup the compression stream
    compressionStream.next_in=(Bytef *)[myData bytes];
    compressionStream.avail_in = [myData length];
    compressionStream.zalloc = Z_NULL;
    compressionStream.zfree = Z_NULL;
    compressionStream.opaque = Z_NULL;
    compressionStream.total_out = 0;

    // start the compression of the stream using default compression
    if (deflateInit2(&compressionStream,
                     Z_DEFAULT_COMPRESSION,
                     Z_DEFLATED,
                     (15+16),
                     8, Z_DEFAULT_STRATEGY) != Z_OK) {
        // Something failed
        errorOccurred = YES;
        return nil;
    }


    // loop over the input stream writing bytes into
    // the compressedData buffer in 16K chunks
    do {
        // for every 16K of data compress a chunk into
        // the compressedData buffer
        if (compressionStream.total_out >= [compressedData length]) {
            // increase the size of the output data buffer
            [compressedData increaseLengthBy:16386];
        }

        compressionStream.next_out = [compressedData mutableBytes] +
                                          compressionStream.total_out;
        compressionStream.avail_out = [compressedData length] -
                                          compressionStream.total_out;
        // compress the next chunk of data
        deflate(&compressionStream, Z_FINISH);

      // keep going until no more compressed data to copy out
    } while (compressionStream.avail_out == 0);

    // end the compression run
    deflateEnd(&compressionStream);
    // set the actual length of the compressed data object
    // to match the number of bytes
    // returned by the compression stream
    [compressedData setLength: compressionStream.total_out];
    return compressedData;
}
```

Listing 7-3 shows the code to add the `Content-Encoding` header to the request. Without this header the DEFLATE module does not know that the contents of the request are compressed.

LISTING 7-3: PostCompress/CompressRequest.h

```
[request addValue:@"gzip" forHTTPHeaderField:@"Content-Encoding"];
NSData *compressed = [self compressNSData:myData];
[request setHTTPBody:compressed];
reqSize = [compressed length];
```

The example app implements compression and provides a way to perform multiple requests containing the sample payload against a URL of your choice. It displays the resulting size of the payload in addition to the average and total time consumed by the requests. The times include the computational time required to compress the payload. Figure 7-4 shows the response time changes when different combinations of request and response compression are applied. The results here are calculated using the Network Link Conditioner tool, which provides a more consistent performance than an uncontrolled live cellular network.

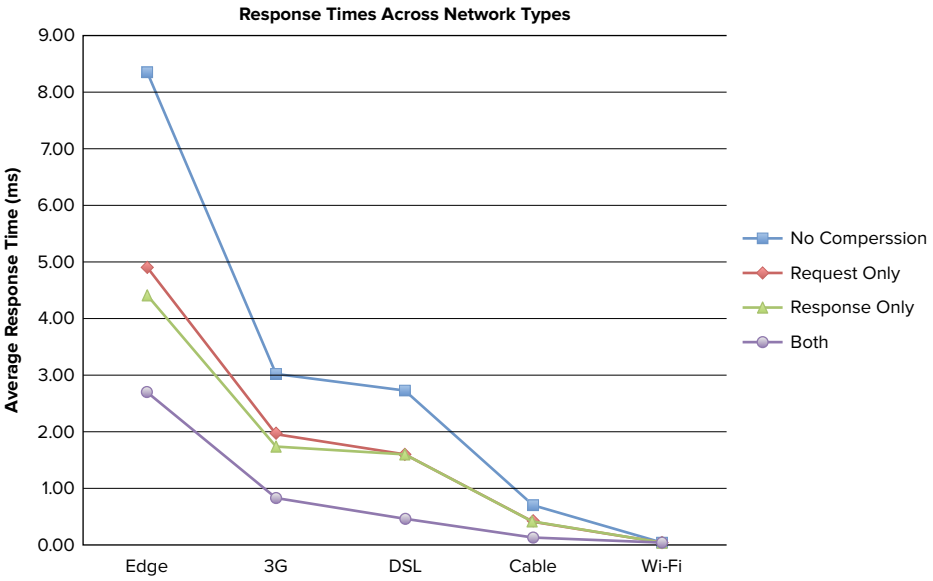


FIGURE 7-4

The benefit is seen by the DSL profile because DSL connections are typically asymmetric for upstream and downstream speeds. Most notably, a request that would have taken almost 10 seconds on an Edge network dropped to just below 3 seconds when both the request and response are compressed.

Reducing Request Latency

Network latency includes the time required to establish a link between the phone and the carrier network, to establish a TCP connection, to possibly negotiate an SSL connection, and to send and

receive an HTTP request. Practically speaking, there is no way on an iOS device to reduce the latency for a single network request, but there are techniques for reducing latency when multiple requests are issued. In this section you learn how to reduce the latent time consumed by your application's network requests.

The techniques discussed in this section provide a means to avoid the repeated consumption of this latent time. Just as it would be foolish to run to the grocery store once for every single item on your shopping list; it is unwise to establish and tear down a TCP connection for each minor piece of data required by your application. There are two best practices to reduce request latency: clustering HTTP requests on a single TCP connection and pipelining HTTP requests to optimize the use of a full-duplex TCP connection.

Your app is probably already using HTTP request clustering because iOS does it by default. After an app finishes using an `NSURLConnection` object, the operating system keeps it open for several seconds, usually about 10, before closing the connection. The technique can be used at a higher level by holding nonessential updates until a sufficient batch has accumulated or some user action requires network activity. The app can then perform all the queued requests in close succession, keeping the same connection active and avoiding the overhead of establishing more than one TCP connection.

An alternative is to architect your service tier with a single service endpoint that proxies requests to other services inside or outside of your organization. This approach can avoid latency by allowing the application to reuse a single connection for disparate activities.

HTTP pipelining is a third way to reuse existing TCP connections. It enables an HTTP client to send a second request on the same TCP socket before a response to the first request is returned. The responses are returned in the same order that the requests were made. Figure 7-5 illustrates the flow of requests and responses for both pipelined and non-pipelined communications. Because POST and PUT commands may modify entities on the server, it is recommended that you do not pipeline those requests.

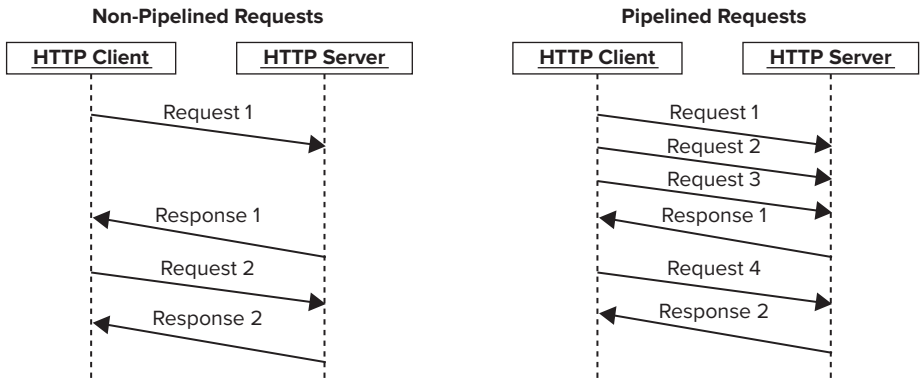


FIGURE 7-5

You can enable pipelining for your `NSURLRequest` easily, as shown in the following code snippet.

```
NSMutableURLRequest *request = [[NSMutableURLRequest alloc]
                               initWithURL:[NSURL URLWithString:url]];
[request setHTTPShouldUsePipelining:YES;
```

> **NOTE** *The request is a mutable request.*

This approach should be used only with extensive testing on the target servers because not all servers support HTTP pipelining. Both Apache and IIS support pipelining without additional configuration.

Avoid Network Requests

Beyond making requests smaller to reduce bandwidth and grouping them to avoid latency, the best way to improve network performance is to avoid the network altogether. In this section you learn the basics of HTTP caching and how you can leverage those rules in an iOS application to cache content locally to avoid unnecessary network traffic.

The IETF has explicitly defined how HTTP caching should work between a web browser and a web server in RFC 2616. You can find this information at `http://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html`. HTTP was designed for browser-to-server communications, and the caching mechanisms are geared toward that usage pattern. iOS provides a mechanism to leverage standard HTTP caching and to also adopt alternative behaviors. Every request issued via `NSURLRequest` goes through a caching component. This component is an instance of an `NSURLCache` object or a subclass of it. This object is the standard mechanism for iOS to manage the caching of responses from a server.

Default Caching Behavior

By default, `NSURLRequest` complies with RFC 2616 for managing the cache. This default behavior specifies that the cache will either return the most current copy of the content, a warning indicating that the returned content might be stale, or an error indicating that the content cannot be returned.

On iOS this means that a returned request is cached in memory when it is retrieved the first time only if the headers of the response indicate that the response can be cached. On subsequent requests to the same URL, iOS sends a request to the server with an `If-Modified-Since` header containing the modified date and time of the cached response. If the server determines that the content has not been modified since the time supplied in the header, it returns an HTTP response with a status of `304` and no response body. From this response iOS determines that the copy it has in cache is the freshest content and returns it with a status code of `200`, effectively hiding the cache activities from the application code. In network configurations where the content comes from a content delivery network (CDN), the source URL may change from request to request, thereby defeating the caching approach defined by the HTTP standards.

These standard caching rules were designed for interaction with a web browser. Mobile applications using HTTP as a transport protocol can bend these default rules to improve performance and still fulfill all application requirements. The URL loading system in iOS provides client applications with

a means to override the default behavior. When overriding the default behavior, you need to take the time to fully understand the edge cases that may cause defects in your application.

You can override the default caching rules by setting the cache policy for a request, as shown in the following code snippet.

```
NSMutableURLRequest *request=[[NSMutableURLRequest alloc]
                              initWithURL:[NSURL URLWithString:url]];
[request setCachePolicy:NSURLRequestUseProtocolCachePolicy];
```

> **NOTE** *Notice that the request is an* `NSMutableURLRequest`. *This enables your code to modify the parameters of the request.*

iOS provides six different settings that enable the developer to control how responses are cached:

- **NSURLRequestUseProtocolCachePolicy —** This setting instructs the system to follow the rules specified in RFC 2616.
- **NSURLRequestReloadIgnoringLocalCacheData —** This setting instructs the request to bypass the local cache and retrieve new contents from the network. If some network appliance, such as a caching network proxy, is between your application and the source of the data and has kept a cached copy of the content, that copy may be returned.
- **NSURLRequestReloadIgnoringLocalAndRemoteCacheData —** This setting instructs the request to bypass the local cache and to add headers to the request, asking intermediate appliances to also bypass their cache to provide the freshest data from the source server.
- **NSURLRequestReturnCacheDataElseLoad —** This setting causes the caching system to return a cached copy of the content without verifying with the server that it has the freshest copy. If a cached copy of the request exists in the cache, it will be returned. If a cached copy does not exist, the content is retrieved via a network request. This setting provides the fastest response time but has the highest possibility of returning stale data. One way to use this setting to your advantage is to use this type of request to provide initial quick responsiveness to the user but also issue a request on a background thread that refreshes the cache with fresh data from the server.
- **NSURLRequestReturnCacheDataDontLoad —** This setting specifies that only content from the cache will be returned. If the content is not in the cache, an error will be returned rather than fetching the content from a server.
- **NSURLRequestReloadRevalidatingCacheData —** This setting always revalidates the data. In some scenarios the cached response may have an expiration time after which the system will check for fresher data. If this setting is used, that expiration time is ignored and the freshness of the content is always validated against the server.

Beyond configuring how each request utilizes the cache, you can specify the amount of data cached by your application by configuring the application's `NSURLCache` object.

Configuring NSURLCache

An instance of `NSURLCache` is created when your application starts making networking requests using any of the standard iOS classes. By default this instance caches data only in RAM, which means that when your program exits, its cached requests will be cleared. The RAM cache will also be cleared whenever the device enters a low-memory state.

iOS does provide a way to redefine the default cache and specify a larger memory capacity and persistent storage to allow the cache to survive app restarts. The following code snippet shows a redefinition of the default cache:

```
NSURLCache *cache = [[NSURLCache alloc]
         initWithMemoryCapacity:1024*1024
                   diskCapacity:1024*1024*20
                       diskPath:@"URLCache"];
[NSURLCache setSharedURLCache:cache];
```

This example creates a 1-MB memory cache and a 20-MB persistent cache. The location of the cache database is in the application's sandbox under the `Library/Caches` directory in a file named `URLCache`. The second line of the example sets the application's cache instance to the one just created in the line above it.

There is an odd behavior in iOS that in some situations system components in the application set the memory capacity of the cache to 0 MB, which effectively disables the cache. One way to defeat this unexplained behavior is to subclass `NSURLCache` with your own implementation that rejects attempts to set the memory cache size to zero. The following code shows a subclass implementation that prevents this behavior.

```
@interface NonZeroingCache : NSURLCache

@end

@implementation NonZeroingCache

-(void)setMemoryCapacity:(NSUInteger)newMemSize
{
    if (newMemSize == 0) {
        NSLog(@"Attempt to set cache size to 0");
        return;
    }
    [super setMemoryCapacity: newMemSize];
}

@end
```

The code in the `setMemoryCapacity:` method validates that the size is not zero and calls the super class to set the new size for any value other than zero.

You can maximize app performance by compressing data and pipelining requests, but the fastest requests are the requests you don't make. By carefully considering your application requirements and the behavior of your server, you can retain data in a cache and refresh that data only when it has changed on the server to avoid making those requests.

SUMMARY

iOS users expect applications that respond immediately to their every request. There is a rule of thumb in the mobile industry that the smaller the screen, the more impatient the user. To provide an application that is enjoyable to use means valuing the user's time as much as you value your time. Optimizing the amount of bandwidth your application uses by compressing responses and requests, avoiding unnecessary latency by pipelining requests, and even avoiding redundant network requests by caching responses can make your application faster and improve the user experience.

8

Low-Level Networking

WHAT'S IN THIS CHAPTER?

- Deciphering BSD Sockets
- Implementing CFNetwork APIs
- Developing with NSStream

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `www.wrox.com/remtitle.cgi?isbn=9781118362402` on the Download Code tab. The code for this chapter is found in one example project: `Low-Level Networking.zip`.

Every complex computer system is built on one or more layers of abstraction, and low-level networking is no exception. At the root of the networking house of cards is the humble Berkley, or BSD, socket. It performs the most basic task of networking: sending and receiving a series of bits. Because it takes a fair amount of code to properly send a single byte and because that same logic is repeated for every single socket, libraries were built to encapsulate that logic so it can be reused by anyone. On iOS the resulting library is called Core Foundation networking, or CFNetwork, which is a lightweight wrapper around raw sockets, but soon became too cumbersome for the most common use cases. Eventually another layer, `NSStream`, was added as a wrapper around CFNetwork and was intended to be the most basic Objective-C networking API. More familiar classes, such as `NSURLConnection` and `UIWebView`, are easy to use and accomplish a lot with minimal code because of the solid foundation these three low-level libraries provide. This chapter gives working examples of each library connecting to the same server and performing the same task, which enables you to compare their relative power and complexity.

BSD SOCKETS

What became the BSD socket API was first implemented in the late 1980s by researchers at the University of California at Berkley. It was eventually standardized as the Portable Operating System Interface (POSIX) sockets API by the Institute of Electrical and Electronics Engineers (IEEE) as the standard for UNIX and all UNIX-like operating systems. Its popularity and relative ease-of-use has also inspired a similar implementation for the Winsock API for Microsoft Windows. At some layer, BSD sockets carry almost all Internet traffic and give application programmers absolute control over any communication to a remote device or server. A single socket is a one-way connection between two endpoints; thus they are typically created in pairs: one for reading and one for writing. Like almost all other resources on UNIX systems, sockets are represented as files and are assigned a file descriptor when created. The six most common socket API calls are summarized in Table 8-1.

TABLE 8-1: BSD SOCKET API CALLS

API CALL	DESCRIPTION
`int socket(int addressFamily, int type, int protocol)`	Creates and initializes a new socket. Returns a file descriptor number on success and `-1` on failure
`int bind(int socketFileDescriptor, sockaddr *addressToBind, int addressStructLength)`	Assigns the socket to the address and port specified in the `addressToBind` struct
`int accept(int socketFileDescriptor, sockaddr *clientAddress, int clientAddressStructLength)`	Accepts a connection request and stores the client's address into `clientAddress`
`int connect(int socketFileDescriptor, sockaddr *serverAddress, int serverAddressLength)`	Connects to the server specified in `serverAddress`
`hostent* gethostbyname(char *hostname)`	Attempts to use DNS to find an IP address corresponding to the provided hostname
`int send(int socketFileDescriptor, char *buffer, int bufferLength, int flags)`	Sends up to `bufferLength` bytes from `buffer` across the socket
`int receive(int socketFileDescriptor, char *buffer, int bufferLength, int flags)`	Reads up to `bufferLength` bytes from the socket into `buffer`
`int sendto(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *destinationAddress, int destinationAddressLength)`	Sends up to `bufferLength` bytes from `buffer` to `destinationAddress`
`int recvfrom(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *fromAddress, int *fromAddressLength)`	Reads up to `bufferLength` bytes from the socket into `buffer` and stores the sender's address into `fromAddress`

BSD sockets are implemented strictly in C and can be used unmodified in Objective-C code. This is convenient if you would like to reuse an existing networking library, use a port code from another platform with little hassle, or if you have previous socket experience and do not want to spend resources learning one of Apple's higher-level frameworks. Apple recommends against this practice, however, because raw sockets don't have access to built-in networking features of the OS like the system wide VPN. Even worse, initiating a socket connection won't automatically turn on the device's Wi-Fi or cellular radios. The radios are intelligently turned off to save battery power, and any communication attempts will fail until some other networking process activates the radio. CFNetwork's wrapper around BSD sockets can activate the device's radio; thus it is recommended over BSD sockets in almost every scenario.

To create a socket, call `socket(int addressFamily, int type, int protocol)` with the desired networking domain, socket type, and protocol enumeration values from `socket.h`. Typically, the `addressFamily` is either IPv4 (`AF_INET`) or IPv6 (`AF_INET6`) for traffic originating from an iOS app; however, you can also open a socket to a local file. The socket type is commonly set to stream (`SOCK_STREAM`) or datagram (`SOCK_DGRAM`). These two values are important because `socket()` is frequently called with a `protocol` value of `0`, which indicates that the system can use the `domain` and `type` values to automatically choose the appropriate protocol. For stream sockets, the automatic value is Transmission Control Protocol (`IPPROTO_TCP`) and for datagram sockets it is User Datagram Protocol (`IPPROTO_UDP`). The semantics of these two protocols are discussed in more detail in Chapter 12, "Device-to-Device Communication with GameKit." If the socket is successfully created, the returned value is the number of the new file descriptor; however, if the call fails for any reason, the return value will be `-1`. At this point, no communication has occurred yet, and the socket has not been designated as an input or output socket (this won't happen until the socket is used for the first time). Clients are now ready to begin connecting to a server; however, a server requires one or more calls before it is ready to communicate.

Configuring a Socket Server

The BSD socket server must associate the socket with a unique address by calling `bind(int socketFileDescriptor, sockaddr *addressToBind, int addressStructLength)`. This takes the socket and assigns, or binds, it to a specific address and port. It returns `0` for a successful bind and `-1` otherwise. After the socket is bound, the next step depends on the type of connection you specified in the `socket()` call, either UDP or TCP:

- For UDP sockets, you are ready to start transmitting data to the world because UDP is a connection-less protocol and doesn't require someone listening on the other end.
- TCP sockets are connection-oriented and require a participant on the other end of the socket. To establish a connection for TCP, you must call `listen(int socketFileDescriptor, int backlogSize)` to set up the data structure for the backlog queue.

The socket passed as the first argument becomes a read-only socket and can't be used to send messages. The `backlogSize` indicates how many pending connections can be queued up while waiting to be acknowledged by your server code. When listening, the server waits for an incoming connection request and calls `accept(int socketFileDescriptor, sockaddr *clientAddress, int clientAddressStructLength)` to accept the request. This removes the pending request from

the backlog queue and populates the `clientAddress` struct with the client's addressing information, most importantly its IP address and port. After the pending request has been accepted, the server is ready to receive messages from a client.

Connecting as a Socket Client

A client's first action depends on the protocol in use by the socket. For TCP sockets, the client must first negotiate the connection to a server with `connect(int socketFileDescriptor, sockaddr *serverAddress, int serverAddressLength)`. The call blocks while the TCP handshake occurs and then returns `0` on success or `-1` on failure. For UDP sockets, `connect()` is optional; however, calling it sets the default address for the socket for all UDP traffic. This makes sending and receiving UDP datagrams convenient. If the device connects to a hostname instead of an IP address, it is probably not clear how to proceed because `sockaddr` struct contains only an IP address. The Domain Name System (DNS) was created to solve this problem of converting a hostname to an IP address. The `hostent* gethostbyname(char *hostname)` function makes a blocking DNS query for the specified hostname. The `hostent` struct contains a list of IP addresses for the host in a format directly compatible with the `sockaddr` struct used through the sockets API. If `hostname` contains an IP address in dot notation, the call simply populates the returned `hostent`'s first result with the given IP address and returns immediately. This behavior makes it easy for the user to provide either an IP address or hostname without needing to branch the connection logic. If a DNS entry cannot be found for the supplied hostname, `gethostbyname()` returns `NULL`.

Once the socket is connected, the device can send or receive messages across it. There are two pairs of socket API calls to use, and the correct pair depends on the type of socket in use. TCP sockets use the `int send(int socketFileDescriptor, char *buffer, int bufferLength, int flags)` and `int receive(int socketFileDescriptor, char *buffer, int bufferLength, int flags)` pair. When sending, the socket described by `socketFileDescriptor` transmits `buffer`'s bytes between 0 and `bufferLength`. If successful, it returns the number of bytes successfully sent and `-1` for any failure. When receiving, `buffer` is populated with a copy of the first `bufferLength` bytes read from the socket. Similarly to `send()`, `receive()` also returns the number of bytes successfully read and `-1` for any failure. UDP sockets that previously used `connect()` to set the default address can also use `send()` and `receive()` in the same manner as TCP sockets. Otherwise, UDP sockets must use the second pair of API calls used specifically for connectionless protocols.

UDP sockets can send to multiple addresses using the same socket connection with the `int sendto(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *destinationAddress, int destinationAddressLength)` API call. This call behaves similarly to `send()` except it has additional arguments for the destination address. Because UDP doesn't make any guarantee about message delivery, it can immediately use the socket to send to another address via another `sendto()` call. The corresponding connectionless receiving call is `int recvfrom(int socketFileDescriptor, char *buffer, int bufferLength, int flags, sockaddr *fromAddress, int *fromAddressLength)` and behaves similarly to `sendto()`. Note one important difference: The last argument is a pointer to an integer that will be populated with the final length of the `fromAddress` struct. Because UDP sockets are not usually connected to a single endpoint, the code receiving a datagram needs to know where it came from, and `recvfrom()` populates `fromAddress` with that information.

Now that all the pieces are in place to connect to other devices and send or receive data, the following example ties everything together. The example app connects to a monitoring server in a warehouse that controls its alarm and climate control systems. The example app must connect using a low-level networking framework because the warehouse server reports its results over the telnet protocol, which isn't directly supported by the higher-level objects such as `NSURLConnection`. Three separate networking controllers each load the same telnet results using a different low-level framework and display the fetched results to the user. The warehouse server responds with a string formatted according to the following snippet of code:

```
84,60,+67,1,1,0,0,0,1
{room temperature},{outlet temperature},{coil temperature},{compressor status},
{air switch status},{auxiliary heat status},{front door status},{system status},
{alarm status}
```

Each controller fetches the data in a background thread, as shown in Listing 8-1, to prevent the user interface from blocking during the network communication.

LISTING 8-1: Fetching Results in a Background Thread (LLNNetworkingController.m)

```
- (void)start {
    NSURL *url = [NSURL URLWithString:[NSString stringWithFormat:@"telnet://%@:%i",
        self.urlString, self.portNumber]];

    NSThread *t = [[NSThread alloc] initWithTarget:self
                                          selector:@selector(loadCurrentStatus:)
                                            object:url];
     [t start];
}
```

The networking controller reports its results to the user interface through two delegate messages: `networkingResultsDidLoad:` takes the networking results as an argument, and `networkingResultsDidFail:` takes a user-readable message that indicates what went wrong. The complete socket implementation to load the warehouse results is shown in Listing 8-2.

LISTING 8-2: Loading Results with BSD Sockets (LLNBSDSocketController.m)

```
- (void)loadCurrentStatus:(NSURL*)url {
    if ([self.delegate respondsToSelector:@selector(networkingResultsDidStart)]) {
        [self.delegate networkingResultsDidStart];
    }

    // create a new Internet stream socket
    socketFileDescriptor = socket(AF_INET, SOCK_STREAM, 0);

    if (socketFileDescriptor == -1) {
        if ([self.delegate respondsToSelector:
                @selector(networkingResultsDidFail:)]) {

            [self.delegate networkingResultsDidFail:
```

continues

LISTING 8-2 *(continued)*

```
                @"Unable to allocate networking resources."];
        }

        return;
    }


    // convert the hostname to an IP address
    struct hostent *remoteHostEnt = gethostbyname([[url host] UTF8String]);

    if (remoteHostEnt == NULL) {
        if ([self.delegate respondsToSelector:
                @selector(networkingResultsDidFail:)]) {

            [self.delegate networkingResultsDidFail:
                @"Unable to resolve the hostname of the warehouse server."];
        }

        return;
    }

    struct in_addr *remoteInAddr = (struct in_addr *)remoteHostEnt->h_addr_list[0];

    // set the socket parameters to open that IP address
    struct sockaddr_in socketParameters;
    socketParameters.sin_family = AF_INET;
    socketParameters.sin_addr = *remoteInAddr;
    socketParameters.sin_port = htons([[url port] intValue]);

    // connect the socket; a return code of -1 indicates an error
    if (connect(socketFileDescriptor, (struct sockaddr *) &socketParameters,
            sizeof(socketParameters)) == -1) {

        NSLog(@"Failed to connect to %@", url);

        if ([self.delegate respondsToSelector:
                @selector(networkingResultsDidFail:)]) {

            [self.delegate networkingResultsDidFail:
                @"Unable to connect to the warehouse server."];
        }

        return;
    }

    NSLog(@"Successfully connected to %@", url);

    NSMutableData *data = [[NSMutableData alloc] init];
    BOOL waitingForData = YES;

    // continually receive data until you reach the end of the data
    while (waitingForData){
```

```
            const char *buffer[1024];
            int length = sizeof(buffer);

            // read a buffer's amount of data from the socket; the number
            // of bytes read is returned
            int result = recv(socketFileDescriptor, &buffer, length, 0);

            // if you got data, append it to the buffer and keep looping
            if (result > 0){
                [data appendBytes:buffer length:result];

            // if you didn't get any data, stop the receive loop
            } else {
                waitingForData = NO;
            }
        }

        // close the stream since you are done reading
        close(socketFileDescriptor);


        NSString *resultsString = [[NSString alloc] initWithData:data
                                                         encoding:NSUTF8StringEncoding];
        NSLog(@"Received string: '%@'", resultsString);

        LLNNetworkingResult *result = [self parseResultString:resultsString];

        if (result != nil) {
            if ([self.delegate respondsToSelector:
                    @selector(networkingResultsDidLoad:)]) {

                [self.delegate networkingResultsDidLoad:result];
            }

        } else {
            if ([self.delegate respondsToSelector:
                    @selector(networkingResultsDidFail:)]) {

                [self.delegate networkingResultsDidFail:
                    @"Unable to parse the response from the warehouse server."];
            }
        }
    }
```

The sockets controller begins by creating a new Internet stream socket with `socket()`. It then takes the server's hostname and uses `gethostbyname()` to get its IP address. If both of these calls are successful, the app is ready to populate the `sockaddr_in` struct with the hostname and port, and then use it to connect to the server. When setting the port, the integer value is converted to network byte order with `htons()` to ensure it can be read properly by both big endian and little endian systems. In the `connect()` call, `sockaddr_in` is cast to a `sockaddr`, and this is possible because both structs have the same layout when `sockaddr_in.sin_family` is `AF_INET`. When the socket is connected, the app is ready to read any available data into an `NSMutableData` for later processing. It reads in 1,024-byte chunks until a read call indicates that no more data is available. After all the

data is downloaded, it is converted to a string, parsed into each individual piece of environment data, and given to the UI for display to the user. Cleaning up a socket is as easy as calling `close()` on the socket file descriptor.

CFNETWORK

CFNetwork is situated one step higher in the framework hierarchy and is a lightweight wrapper around BSD sockets. You can notice many similarities in the callback methods and logic flow that are a consequence of the underlying BSD foundation. As noted earlier, the main benefit of CFNetwork over raw sockets is that CFNetwork integrates into system-level settings and the main run loop. System services such as turning on the antenna when necessary and routing through a system wide VPN are benefits gained when moving up in the framework hierarchy, and there are few significant drawbacks.

Run loop integration is essential to threading and is the basis of the event processing flow in iOS. Every thread can have its own run loop, and the run loop of the main thread is called the main run loop or UI run loop. Each loop schedules the processing of asynchronous events and sleeps the thread if no events have occurred. Examples of these events are keyboard or gesture input, networking events, or timers, and each one needs to be handled by custom application logic. As you can see in the next code example, each thread has its own run loop; however, each secondary thread must explicitly start and stop its own loop. CFNetwork uses a callback system similar to the delegates used in higher-level Objective-C frameworks and is only feasible because of the run loop support in the framework.

C functions are used to create and open the socket, and the callback system is used for all other events originating from it. CFNetwork contains a create convenience function `CFStreamCreatePairWithSocketToHost()` that can create a pair of sockets, one for reading and one for writing, for a given hostname and port. The framework takes care of converting the hostname to an IP address and the port number to network byte order. If you don't need one of the pair, you can simply pass `NULL` as the read or write stream argument to skip its allocation. Just as with raw sockets, the streams must be explicitly opened with `CFReadStreamOpen()` or `CFWriteStreamOpen()` before they can be used. Both calls are asynchronous, and will call the callback function with `kCFStreamEventOpenCompleted` when successfully opened. Listing 8-3 demonstrates creating and opening the stream using the same warehouse feed from the previous example.

LISTING 8-3: Creating a Socket with CFNetwork (LLNCFNetworkController.m)

```
- (void)loadCurrentStatus:(NSURL*)url {
    if ([self.delegate respondsToSelector:@selector(networkingResultsDidStart)]) {
        [self.delegate networkingResultsDidStart];
    }

    // keep a reference to self to use for controller callbacks
    CFStreamClientContext ctx = {0, (__bridge void *)(self), NULL, NULL, NULL};

    // get callbacks for stream data, stream end, and any errors
```

```
CFOptionFlags registeredEvents = (kCFStreamEventHasBytesAvailable |
                                  kCFStreamEventEndEncountered |
                                  kCFStreamEventErrorOccurred);


// create a read-only socket
CFReadStreamRef readStream;
CFStreamCreatePairWithSocketToHost(kCFAllocatorDefault,
                                   (__bridge CFStringRef)[url host],
                                   [[url port] integerValue],
                                   &readStream,
                                   NULL);


// schedule the stream on the run loop to enable callbacks
if (CFReadStreamSetClient(readStream, registeredEvents,
                          socketCallback, &ctx)) {

    CFReadStreamScheduleWithRunLoop(readStream,
                                    CFRunLoopGetCurrent(),
                                    kCFRunLoopCommonModes);

} else {
    NSLog(@"Failed to assign callback method");

    if ([self.delegate respondsToSelector:
            @selector(networkingResultsDidFail:)]) {

        [self.delegate networkingResultsDidFail:
            @"Unable to respond to data from the warehouse server."];
    }

    return;
}


// open the stream for reading
if (CFReadStreamOpen(readStream) == NO) {
    NSLog(@"Failed to open read stream");

    if ([self.delegate respondsToSelector:
            @selector(networkingResultsDidFail:)]) {

        [self.delegate networkingResultsDidFail:
            @"Unable to read data from the warehouse server."];
    }

    return;
}

CFErrorRef error = CFReadStreamCopyError(readStream);

if (error != NULL) {
    if (CFErrorGetCode(error) != 0) {
        NSLog(@"Failed to connect stream; error '%@' (code %ld)",
```

continues

LISTING 8-3 *(continued)*

```
                (__bridge NSString*)CFErrorGetDomain(error),
                CFErrorGetCode(error));
        }

        CFRelease(error);

        if ([self.delegate respondsToSelector:
                @selector(networkingResultsDidFail:)]) {

            [self.delegate networkingResultsDidFail:
                @"Unable to connect to the warehouse server."];
        }

        return;
    }

    NSLog(@"Successfully connected to %@", url);


    // start processing
    CFRunLoopRun();
}
```

The only remaining part of initialization is to register the socket callback function, which requires three distinct steps:

1. First, create a callback function to pass to the stream. Both read and write streams have the same callback signatures and differ only in the type of stream reference passed as the first argument. For example, the callback function can be any function that has the method signature `void socketCallback(CFReadStreamRef stream, CFStreamEventType event, void *info)`. The first argument is the stream that generated the event, and almost every function you would use to process the stream, requires a reference to it. The event that generated the callback is passed as `event` and can be one of the `CFStreamEventType` values, as listed in Table 8-2. The last value is a parameter that you set when registering the context of the stream and can be a pointer to any data that can help you process the stream data.
2. Next, create the previously mentioned context value by populating a `CFStreamClientContext` struct for the stream, which holds a pointer to your own `info` object and callback functions for retaining, releasing, or copying that object. Typically, you pass a reference to `self`, which would point to the class managing the stream, for `info` and `NULL` for each optional callback.
3. Finally, choose the stream events your code wants to receive. Simply store a bitwise OR of the `CFStreamEventType` values described in Table 8-2 that you want to have processed by your callback function.

After these three pieces are in place, call `CFReadStreamSetClient(CFReadStreamRef stream, CFOptionFlags streamEvents, CFReadStreamClientCallBack callbackFunction, CFStreamClientContext *context)` to register the callback with the stream. Listing 8-4 combines these concepts into an example callback function for a read stream.

TABLE 8-2: CFStream Event Types

EVENT CONSTANT	DESCRIPTION
`kCFStreamEventOpenCompleted`	The socket was successfully opened.
`kCFStreamEventHasBytesAvailable`	The socket has bytes available for reading.
`kCFStreamEventCanAcceptBytes`	The socket has space in its buffer to write bytes.
`kCFStreamEventErrorOccurred`	An error occurred with an operation. `CFReadStreamCopyError()` or `CFWriteStreamCopyError()` can provide more details about the error.
`kCFStreamEventEndEncountered`	The socket reached the end of the byte stream.
`kCFStreamEventNone`	This default value represents no event at all.

LISTING 8-4: Example Read Stream Callback (LLNCFNetworkController.m)

```
void socketCallback(CFReadStreamRef stream, CFStreamEventType event, void *myPtr) {
    LLNCFNetworkController *controller = (__bridge LLNCFNetworkController*)myPtr;

    switch(event) {
        case kCFStreamEventHasBytesAvailable:
            // read bytes until there are no more
            while (CFReadStreamHasBytesAvailable(stream)) {
                UInt8 buffer[kBufferSize];
                int numBytesRead = CFReadStreamRead(stream, buffer, kBufferSize);

                [controller didReceiveData:[NSData dataWithBytes:buffer
                                                          length:numBytesRead]];
            }

            break;

        case kCFStreamEventErrorOccurred: {
            CFErrorRef error = CFReadStreamCopyError(stream);

            if (error != NULL) {
                if (CFErrorGetCode(error) != 0) {
                    NSLog(@"Failed while reading stream; error '%@' (code %ld)",
                        (__bridge NSString*)CFErrorGetDomain(error),
                        CFErrorGetCode(error));
                }

                CFRelease(error);
            }

            if ([controller.delegate respondsToSelector:
```

continues

LISTING 8-4 *(continued)*

```
                    @selector(networkingResultsDidFail:)]) {

                [controller.delegate networkingResultsDidFail:
                    @"An unexpected error occurred while reading from
                    the warehouse server."];
            }

            break;
        }

        case kCFStreamEventEndEncountered:
            [controller didFinishReceivingData];

            // clean up the stream
            CFReadStreamClose(stream);

            // stop processing callback methods
            CFReadStreamUnscheduleFromRunLoop(stream,
                                              CFRunLoopGetCurrent(),
                                              kCFRunLoopCommonModes);

            // end the thread's run loop
            CFRunLoopStop(CFRunLoopGetCurrent());

            break;

        default:
            break;
    }
}
```

Even if the callback is successfully registered, you won't see it called until the run loop is properly handled. Thankfully, this requires only two short steps: registering the callback with the run loop and starting the run loop. True to their names, `CFReadStreamScheduleWithRunLoop(CFReadStreamRef stream, CFRunLoopRef runLoop, CFStringRef runLoopMode)` and its write stream equivalent schedule the stream with the given run loop. The `runLoop` is almost always `CFRunLoopGetCurrent()`, which returns the default run loop for the current thread, and `runLoopMode` is almost always `kCFRunLoopCommonModes`, which encompasses the default mode and any other manually added modes for the current thread. When the stream is integrated into the run loop, it's as easy as starting the loop with `CFRunLoopRun()` and waiting for events to happen.

NSSTREAM

Moving up yet another level in the framework hierarchy leads to `NSStream`, which is an Objective-C wrapper around the CFNetwork APIs. It uses a delegate protocol `NSStreamDelegate` that almost exactly mimics the function of the CFNetwork's stream callback function, and all the same run loop requirements apply to `NSStream` as well. `NSStream` has two concrete subclasses, `NSInputStream` and `NSOutputStream`, which perform the same functions as `CFReadStream` and `CFWriteStream`, respectively.

Implementing an `NSStream` for reading or writing is relatively straightforward except for one wrinkle: The iOS SDK doesn't support `NSHost`, which is required to use `NSStream`'s designated initializer, `getStreamsToHost:port:inputStream:outputStream:`. Because this is a pain point for almost every developer, especially those porting code from Mac OS X, Apple has published a technical note (Technical Q&A QA1652, `http://developer.apple.com/library/ios/#qa/qa1652/_index.html`) with an `NSStream` category that mimics the missing functionality, which has been adapted for the example implementation in Listing 8-5. The solution takes advantage of the toll-free bridging between `CFReadStreamRef` and `NSInputStream` and drops down to the CFNetwork framework to create the stream exactly how it was created in Listing 8-3.

TOLL-FREE BRIDGING

If two objects are toll-free bridged, it means that references to the CoreFoundation object are interchangeable with the Objective-C Foundation object. For example, common types such as `NSString`, `NSArray`, `NSDictionary`, and `NSInputStream` are bridged to `CFStringRef`, `CFArrayRef`, `CFDictionaryRef`, and `CFReadStreamRef`, respectively. Although it may seem like magic to interchange bridged types, it is possible because of the way concrete subclasses of the Foundation object are laid out in memory, as well as the brute-force checking in the various CoreFoundation C calls. This complexity is hidden from you; just use whichever bridged type is convenient.

LISTING 8-5: Creating an NSStream to a Hostname without NSHost (NSStream+StreamsToHost.m)

```
+ (void)readStreamFromHostNamed:(NSString *)hostName
                           port:(NSInteger)port
                     readStream:(out NSInputStream **)readStreamPtr {

    assert(hostName != nil);
    assert((port > 0) && (port < 65536));
    assert((readStreamPtr != NULL));

    CFReadStreamRef readStream = NULL;

    CFStreamCreatePairWithSocketToHost(NULL,
                                       (__bridge CFStringRef) hostName,
                                       port,
                                       ((readStreamPtr != NULL) ? &readStream :
                                           NULL),
                                       NULL);

    if (readStreamPtr != NULL) {
        *readStreamPtr  = CFBridgingRelease(readStream);
    }
}
```

After you can easily make a stream, the same basic steps from the previous example implementations are repeated here. Listing 8-6 demonstrates creating a read stream, setting the delegate, scheduling it in the run loop, and opening it. These same steps might have taken a few dozen lines for a BSD socket or CFNetwork implementation; however, here it can be done in just six!

LISTING 8-6: Initializing an NSInputStream (LLNNSStreamController.m)

```
- (void)loadCurrentStatus:(NSURL *)url {
    if ([self.delegate respondsToSelector:
            @selector(networkingResultsDidStart)]) {

        [self.delegate networkingResultsDidStart];
    }

    NSInputStream *readStream;
    [NSStream readStreamFromHostNamed:[url host]
                                 port:[[url port] integerValue]
                           readStream:&readStream];

    [readStream setDelegate:self];
    [readStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                          forMode:NSDefaultRunLoopMode];
    [readStream open];

    [[NSRunLoop currentRunLoop] run];
}
```

Implementing the `NSStreamDelegate` protocol looks similar to the `CFReadStream` callback function and encompasses the exact same events. Listing 8-7 provides an example implementation that reads the warehouse server feed into an `NSMutableData` object and then parses it. The only remotely complex part of the code here is the `read:maxLength:` call for `NSInputStream`; however, the basic concept is similar to the previous examples that read bytes into a buffer. `NSStream` is a lightweight yet powerful object because of the great foundation classes it uses under the hood.

LISTING 8-7: Example NSStreamDelegate Implementation (LLNNSStreamController.m)

```
- (void)stream:(NSStream *)stream handleEvent:(NSStreamEvent)eventCode {
    switch (eventCode) {
        case NSStreamEventHasBytesAvailable: {
            if (receivedData == nil) {
                receivedData = [[NSMutableData alloc] init];
            }

            uint8_t buf[1024];
            int numBytesRead = [(NSInputStream *)stream read:buf maxLength:1024];

            if (numBytesRead > 0) {
                [receivedData appendBytes:(const void *)buf length:numBytesRead];

            } else if (numBytesRead == 0) {
                NSLog(@"End of stream reached");

            } else {
                NSLog(@"Read error occurred");
            }

            break;
        }
```

```
        case NSStreamEventErrorOccurred: {
            NSError *error = [stream streamError];
            NSLog(@"Failed while reading stream; error '%@' (code %d)",
                error.localizedDescription, error.code);

            if ([self.delegate respondsToSelector:
                    @selector(networkingResultsDidFail:)]) {

                [self.delegate networkingResultsDidFail:
                    @"An unexpected error occurred while reading from
                    the warehouse server."];
            }

            [self cleanUpStream:stream];
        }

        case NSStreamEventEndEncountered: {
            NSString *resultsString = [[NSString alloc] initWithData:receivedData
                encoding:NSUTF8StringEncoding];
            NSLog(@"Received string: '%@'", resultsString);

            LLNNetworkingResult *result = [self parseResultString:resultsString];

            if (result != nil) {
                if ([self.delegate respondsToSelector:
                        @selector(networkingResultsDidLoad:)]) {
                    [self.delegate networkingResultsDidLoad:result];
                }

            } else {
                if ([self.delegate respondsToSelector:
                        @selector(networkingResultsDidFail:)]) {

                    [self.delegate networkingResultsDidFail:
                        @"Unable to parse the response from the warehouse
                        server."];
                }
            }

            [self cleanUpStream:stream];

            break;
        }

        default:
            break;
    }
}

- (void)cleanUpStream:(NSStream*)stream {
    [stream removeFromRunLoop:[NSRunLoop currentRunLoop]
                      forMode:NSDefaultRunLoopMode];
    [stream close];

    stream = nil;
}
```

SUMMARY

This chapter has taken you on a tour of the low-level networking frameworks available for iOS and provides pros and cons for each. As you move up the framework hierarchy, code becomes shorter and less complex, but you also lose power as each layer of abstraction hides more of the raw networking sockets that actually perform the communication. The warehouse server examples provide an easy way to determine the proper framework for your app because each implementation can be directly compared in terms of complexity, development time, and ease of use.

9

Testing and Manipulating Network Traffic

WHAT'S IN THIS CHAPTER?

- Observing network traffic
- Manipulating network traffic with proxies
- Simulating real-world network conditions

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code is in the Chapter 9 download and individually named according to the names throughout the chapter.

You make mistakes. This is an inescapable fact of life that everybody deals with every day. Because you and every other person in the world make mistakes, the software you write also has mistakes. Because of this it is essential to diagnose software behavior, find the mistakes, correct them, and try to avoid them in the future.

When writing networked applications, many layers of software and hardware are in between each component of the system. It is often necessary to observe the interaction between these components to get a complete and accurate picture of what happens at a higher level, so you can better discover any existing mistakes.

This chapter describes methods to observe these network interactions, manipulate them, and simulate real-world conditions in them. Observing network interactions gives you an accurate picture of which requests leave the device and what data is received. Manipulating network traffic allows you to create situations in a development lab that occur only when consumers use the app. Simulating network conditions also enables you to validate the behavior of the app under varying network conditions.

OBSERVING NETWORK TRAFFIC

When writing a networked application, you do not have total control over the packets transmitted from or received by the device. Many layers of software and hardware exist between your code on the device and the code on a remote server. Because the interactions between your code and the underlying network layers may be poorly documented or poorly understood, you sometimes must examine the raw data transmitted from the device to know without a doubt what your app is sending. Likewise, similar layers intervene between the server and your app, and you sometimes must know the exact data received from the server. For example, if your app adds HTTP headers to a request, you may need to know exactly how iOS is further manipulating those headers and what it eventually transmits to the server.

The act of observing network traffic is called *sniffing* or *packet analysis*. Packet analyzers have been around since the early days of networking and are available for almost every type of physical interconnect and protocol. This section focuses on sniffing Ethernet and Wi-Fi connections that use TCP transport and HTTP application protocols from a development system running OS X Lion.

Sniffing Hardware

Before you can start capturing network traffic from an iOS device, you must first have a network topology that can facilitate packet capture. The computer running the sniffing software must be on the same network as the Wi-Fi device, or the packets from the Wi-Fi network must be propagated to the sniffing laptop. Therefore, you may need to do some hardware shuffling to capture traffic from a device.

> **NOTE** *No special hardware or network configuration is required to capture packets from the iOS Simulator. If you need to capture these packets, you can use the sniffing software listed in the next section to listen on the loopback device (`lo0`) or the interface used to connect to the network.*

The first suggested configuration, shown in Figure 9-1, uses a Wi-Fi network shared by both the sniffing computer and the iOS device.

In the common Wi-Fi configuration, the iOS device and sniffing computer pair with a Wi-Fi access point. The Wi-Fi adapter on the sniffing computer receives all network packets transmitted over the Wi-Fi network. The sniffing software must be configured to run in monitor mode so that the low-level Wi-Fi drivers propagate all packets to the sniffing software. In some corporate networks with heavy Wi-Fi coverage, the mobile device and the sniffing computer may pair to different access points on different channels. This can occur even when the device and computer are in close proximity because the two systems have different rules controlling which access points to prefer and how aggressively to switch access points. If the device and computer pair to different networks, the computer may not see Wi-Fi traffic from the device.

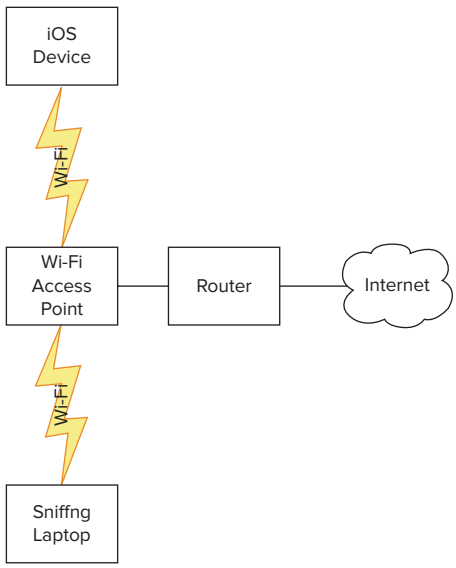


FIGURE 9-1

A second configuration, shown in Figure 9-2, uses the Internet-sharing feature of OS X to make the sniffing laptop act as the Wi-Fi access point.

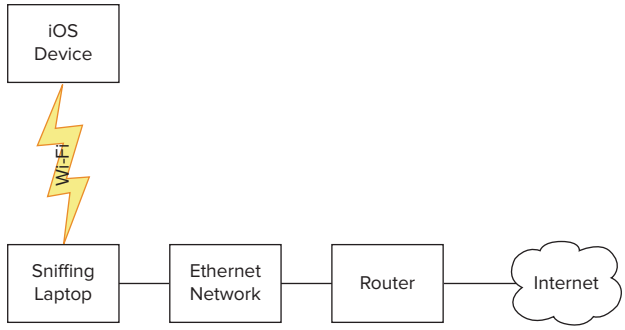


FIGURE 9-2

In this configuration you need to connect the sniffing laptop to a wired Ethernet network and enable Internet sharing in OS X to share the Ethernet network with Wi-Fi users. You then need to configure the iOS device to join the Wi-Fi network created by the sniffing laptop. Each network packet sent from the device to the Internet traverses the Wi-Fi and Ethernet interface of the laptop. Using this configuration you can observe client-server traffic as well as peer-to-peer traffic such as the traffic generated by Game Kit.

Sniffing Software

There are many network sniffers available for OS X; some free like Wireshark (discussed later) and Packet Peeper (see `http://sourceforge.net/projects/packetpeeper/`), as well as some shareware like Cocoa Packet Analyzer (`http://www.tastycocoabytes.com/cpa`).

A command-line packet sniffer, `tcpdump`, is bundled with OS X and is the foundation for most other sniffers available for OS X. `tcpdump` can capture the network traffic from an interface, filter it according to criteria you specify, display the traffic, and save the traffic trace to a log file. The traffic display of `tcpdump` is difficult to decipher, but many packet analyzers leverage the filtering syntax; making it worthwhile to learn.

Capturing with tcpdump

The sniffing needs of network engineers vary greatly from those of app developers. Network engineers need to see every bit transmitted from the device, and this level of detail obscures the HTTP request data most important to app developers. This section looks at the filters commonly used by app developers to analyze networking traffic generated by an app.

If you run `tcpdump` from the command line with no filters, you see a flood of packets. This flood usually obscures the traffic that you want to see: the traffic between your app and its server.

To avoid this you should apply filtering. The most basic filter is filtering by host, which causes `tcpdump` to ignore all traffic except for that with a specific host. The following code snippet shows a filter `tcpdump` command that filters out all traffic except to or from the host at address 192.168.1.50.

```
sudo tcpdump host 192.168.1.50
```

> **NOTE** *The* `sudo` *command will prompt you for a password. You should use the password for the currently logged in account.*

The `host` filter can use either an IP address or a DNS host or domain name. If you specify a DNS domain, any traffic to any host in that domain will be captured. You can also filter traffic from multiple hosts by using logical operators to include other criteria. The following code captures traffic to host 192.168.1.50 and 10.1.2.25.

```
sudo tcpdump host 192.168.1.50 or 10.1.2.25
```

Depending on your network configuration, it may be necessary to filter for traffic destined for an entire subnet. The following code snippet filters traffic to the 192.168.1 subnet.

```
sudo tcpdump net 192.168.1.0/24
```

Another common practice is to filter packets by the TCP or UDP port numbers used by the connection. `tcpdump` uses names defined in `/etc/services` to map a TCP service port name to a port number. The following code shows two equivalent commands to filter for HTTP traffic.

```
sudo tcpdump port http
sudo tcpdump port 80
```

Remember that the value `http` is used to look up the port number, not to detect the protocol used in the connection. Therefore, if you use the HTTP protocol over a TCP connection using a port other than 80, the preceding filters cannot capture that traffic.

The filter criteria can be combined to focus the capture on a specific host and port by using logical operators. The following snippet captures only packets on ports 80 and 8080 for host 192.168.1.50.

```
sudo tcpdump host 192.168.1.50 and \(port 80 or port 443\)
```

Using the `and` and `or` operators along with separating parentheses, you can filter your captured data to eliminate extraneous traffic that might obscure problems. The backslash (`\`) characters in the filter expression prevent the command shell from interpreting the parentheses. When using `tcpdump` filter syntax within a graphical program, these backslash characters must be omitted. Inversely, be careful not to over-filter and remove important data.

`tcpdump` can be used to run an extended capture of network traffic and save it to a capture file for later analysis. The graphical sniffer, Wireshark, described in the next section, "Capturing with Wireshark," can import an existing capture file so that you can drill into the data using a friendlier interface. The following snippet can capture all traffic on port 80 and save it to a file named `http-capture.trace`.

```
sudo tcpdump -s 1514 port 80 -w http-capture.trace
```

The `-s 1514` argument instructs `tcpdump` to capture the first 1514 bytes of the packet. This captures the most common Ethernet packet size; however, if your network uses jumbo-sized packets, you may need to increase this value.

Capturing with Wireshark

Wireshark (`http://www.wireshark.org`) is a free, cross-platform graphical network sniffer available for most major operating systems. It can capture new traces or analyze traces captured earlier with `tcpdump`. On OS X, Wireshark requires that X11 also be installed. The installation process modifies the permissions of a handful of device files so that the installing user can capture data from those devices without being a super-user.

When you start Wireshark you see a helpful start page, as shown in Figure 9-3, where you can start a trace directly against an interface, set trace options, or open an existing trace file.

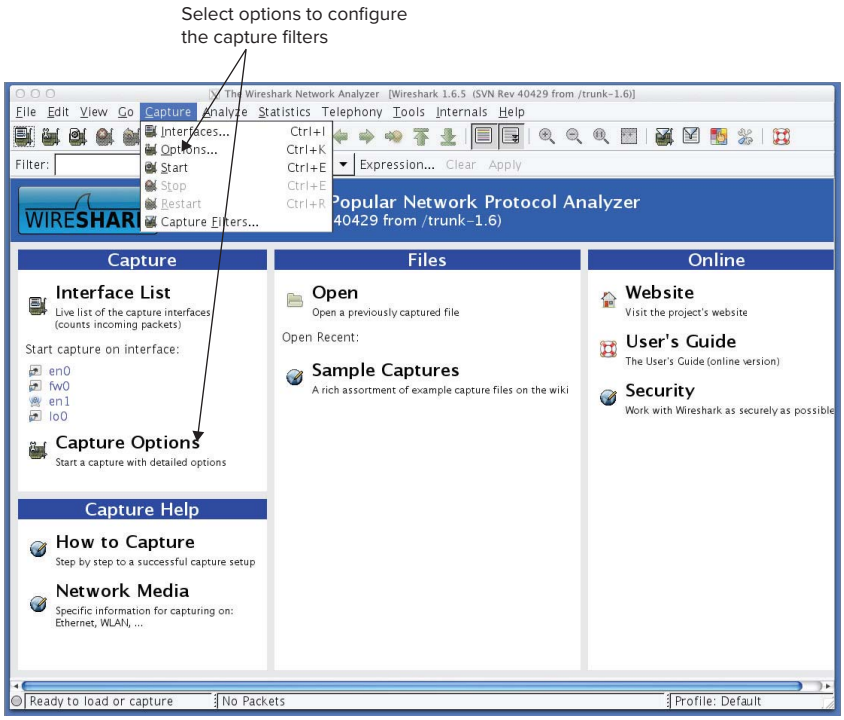


FIGURE 9-3

If you start a trace directly on an interface, you see a flood of packets that are probably not helpful. If you start a trace by using the `Capture->Options` menu entry, you can tune your capture to fit your needs.

> **NOTE** *Because Wireshark is such a powerful tool, it has functions that you will probably never use as a developer; don't let that stop you from experimenting with the application to discover features that may be beneficial. If you configure yourself into a corner where things stop working, close the application and restart it; nothing will be permanently broken.*

Using the Capture Options dialog box, as shown in Figure 9-4, the Interface field allows you to select an interface to capture. Table 9-1 lists three common network interfaces.

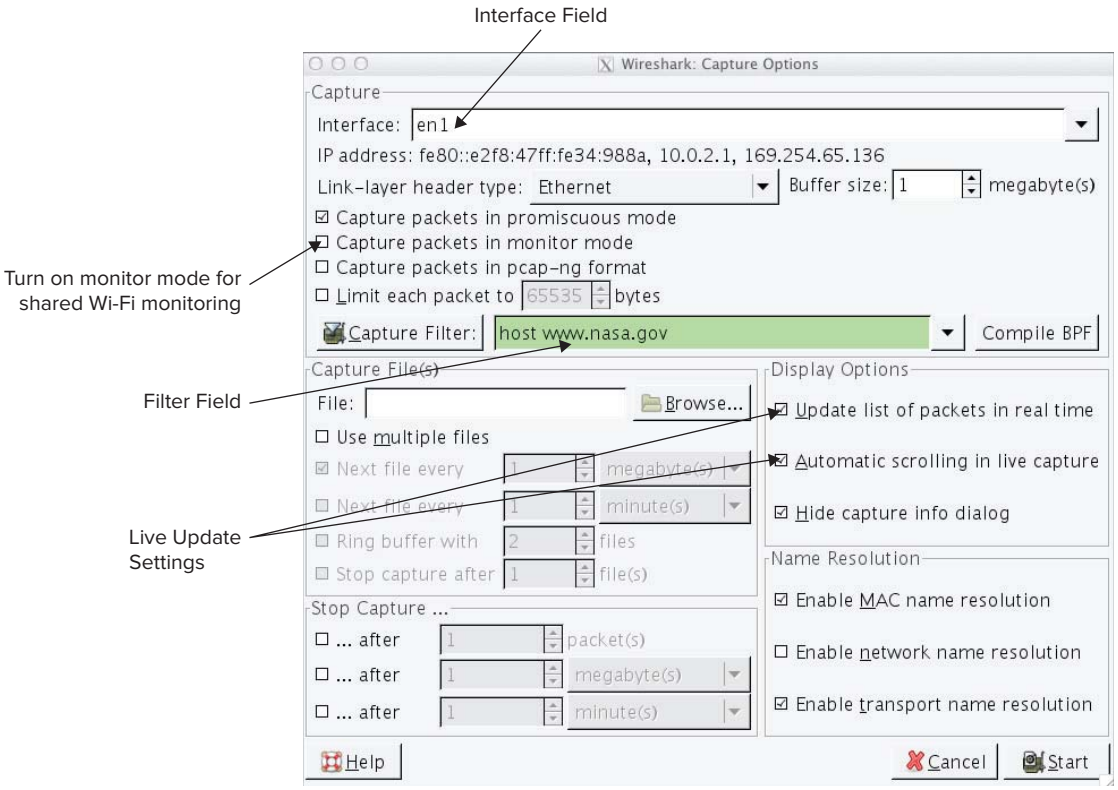


FIGURE 9-4

TABLE 9-1: Network Interface Descriptions

INTERFACE NAME	INTERFACE TYPE	DESCRIPTION
en0	Ethernet	The wired Ethernet port on an OS X laptop
en1	Wi-Fi	The default Airport interface in OS X
lo0	Loopback	The local loopback interface used if a program connects to localhost or 127.0.0.1

Use the capture filter field to specify which packets to capture using the same syntax as the `tcpdump` command. The sniffer captures only packets going to host `www.nasa.gov` (refer to Figure 9-4).

If you are sniffing your app's packets using the common Wi-Fi topology (refer to Figure 9-1) you need to enable monitor mode. Monitor mode configures the Wi-Fi adapter to pass all network

packets received on the Wi-Fi adapter up to the software driver. Without this option enabled, the packets not addressed to the sniffing computer are ignored by the Wi-Fi hardware. Your mileage may vary depending on your wireless network configuration, operating system version, and version of Wireshark. Wireshark is available for OS X, Linux, and Windows. Each platform provides slightly different capabilities so you can use a different OS if necessary to get the perfect configuration for your network.

The live update settings instruct Wireshark to continually update the display as packets are captured. This feature is useful to determine if you have specified the filter correctly and to easily detect when you have captured sufficient data; however, on slower machines it may impact the performance of the application enough to cause it to drop packets. Pressing the Start button on this dialog initiates a capture session.

With live updating enabled, you see packet headers appear in the Captured Packet list as they are captured. Figure 9-5 shows an example capture of the network packets produced when the VideoDownloader app, described in Chapter 3, "Making Requests," was run on an iPhone.

The Packet Decomposition panel exposes the values for each protocol layer for the packet selected in the Captured Packets List. The packet is decomposed into three layers: Ethernet II, Internet Protocol, and Transmission Control Protocol (refer to Figure 9-5).

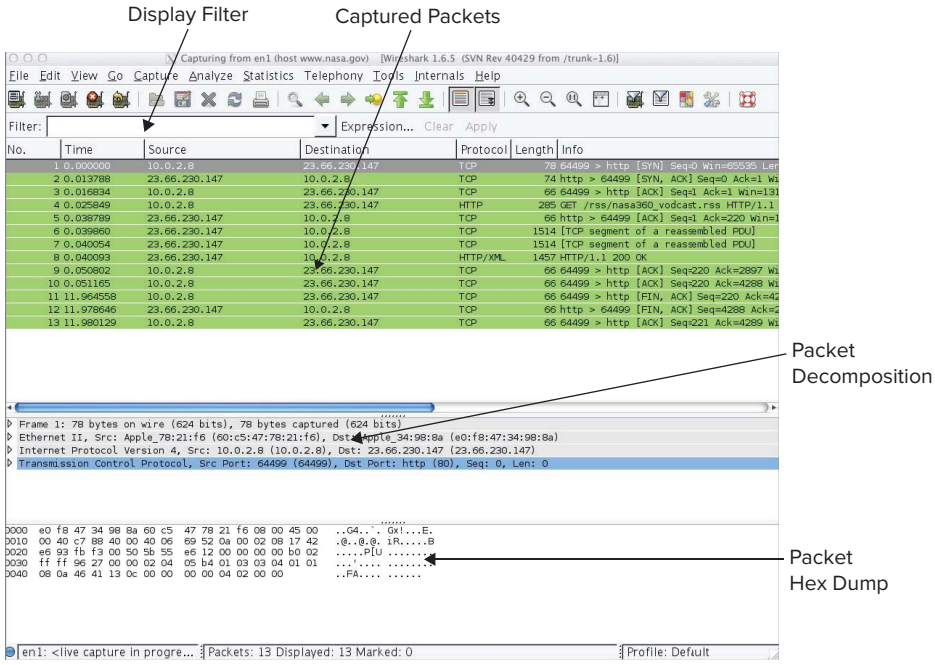


FIGURE 9-5

The Packet Hex Dump area displays the hexadecimal values for each byte in the selected packet and an ASCII conversion of those hex values. If you drill down into a protocol layer in the Packet Decomposition panel the selected portion of the packet is highlighted in the hex dump.

One incredibly helpful feature of Wireshark is the capability to follow a TCP stream. In a typical packet dump on a moderately active machine, you have multiple TCP streams and HTTP conversations occurring concurrently with their packets intermingled. To follow a TCP stream, Ctrl+click on a packet from the stream to activate the packet menu, as shown in Figure 9-6.

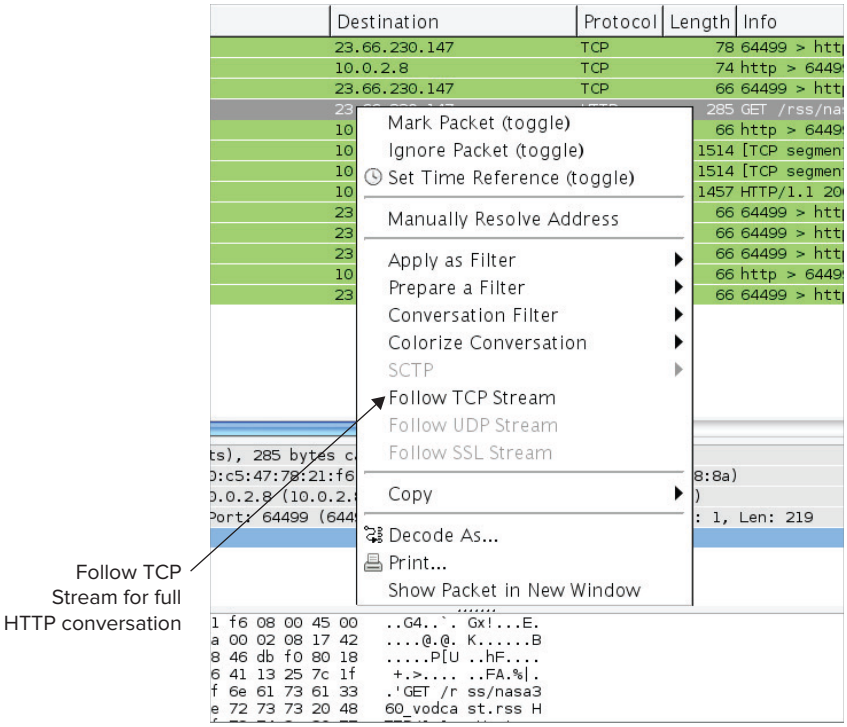


FIGURE 9-6

Select the Follow TCP Stream option of the packet context menu to show the stream trace (see Figure 9-7). The Follow TCP Stream dialog displays the contents of each packet of the stream as one continuous set of data. The outbound and inbound data are shaded differently so that you can easily distinguish the packet direction. In Figure 9-7 the data outbound from the phone has a dark background and the inbound data has a white background. Notice that the response data appears to be gibberish because the `Content-Encoding` of the payload is `gzip`, and it must be decompressed to reveal its actual contents.

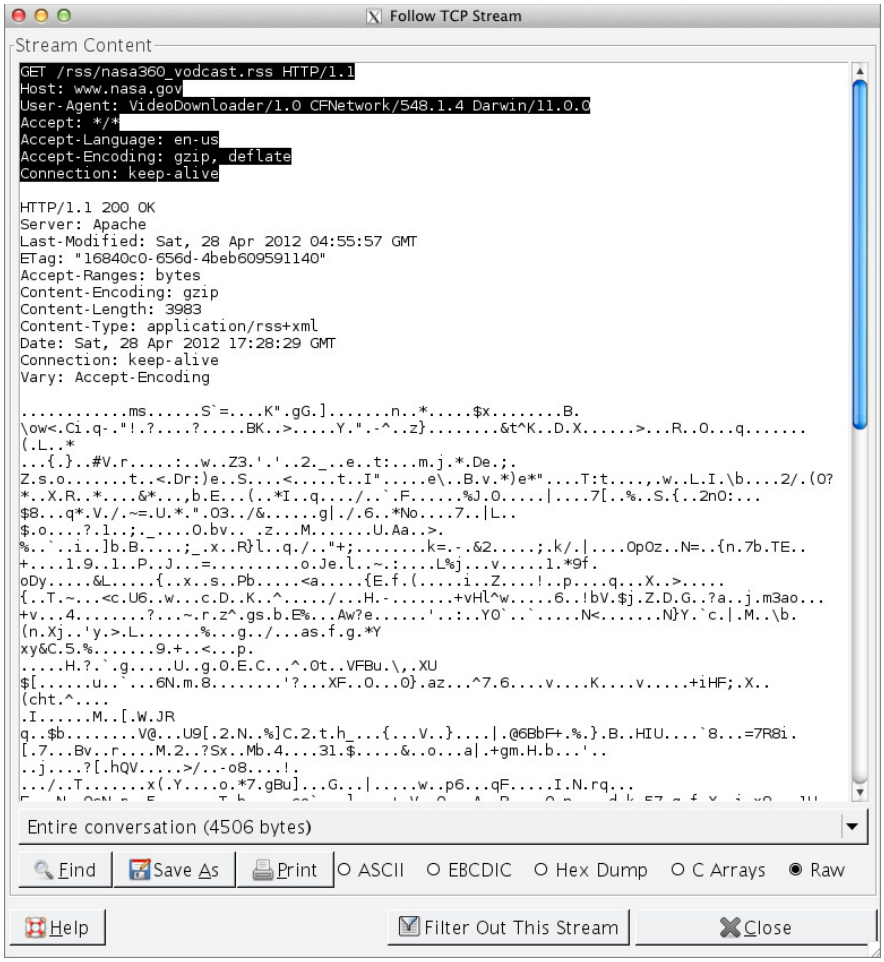


FIGURE 9-7

Wireshark has a solution to the problem of compressed data. Figure 9-8 shows the contents of the eighth packet selected. Wireshark has deduced that this packet is the response to an HTTP request and that it has XML data in the response body. Therefore it provides additional diagnostic data about the entire payload. The Packet Reassembly Views contain the payload of the response pieced together from any subsequent network packets that contain the payload. If the HTTP response were compressed, Wireshark would decompress it. The Uncompressed Entity Body view displays the contents of the response after decompression.

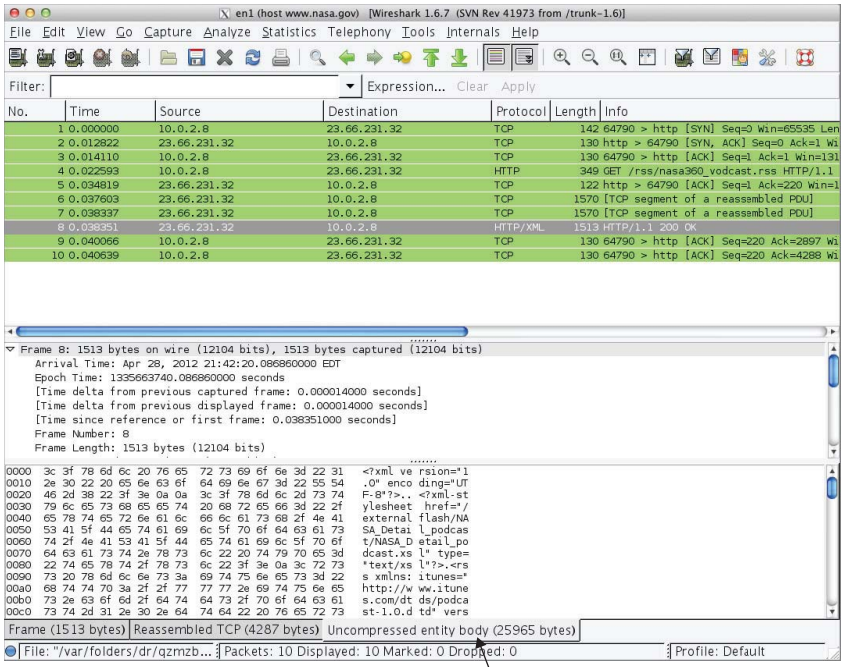


FIGURE 9-8

Wireshark is a powerful tool that should be in every network developer's tool box. It can be extended with modules, called *dissectors*, which parse many industry-standard protocols, but you can also develop your own dissectors to parse your own custom protocols. If built from source, Wireshark can be used to decrypt SSL connections under certain circumstances. By using Wireshark, you can examine every bit and packet transmitted from the device for any networking protocol, which is helpful if you are doing custom protocol development or using non-HTTP based communications. The next section provides a simpler, but less powerful, way to capture and decode network traffic within your development environment.

MANIPULATING NETWORK TRAFFIC

Network packet capture tools provide a way to observe network traffic, but sometimes you need to do more than observe. Luckily there is a common network component, an HTTP proxy, which developers can leverage to manipulate HTTP and HTTPS requests. Some of the uses of HTTP manipulation include

- **Error simulation —** You can intercept a response and change the status code on the response to an error status and optionally change the payload to represent an error. This capability enables you to test how your app responds to error responses from the server without needing to actually invoke an error.

- **Future state simulation —** If you know that a future version of the server will provide different responses from the current server, you can test old versions of the app against the altered protocol.
- **Server validation —** You can modify requests to the server to validate responses based on data that may be difficult to duplicate on a device without writing Objective-C code.
- **Security validation —** Security personnel can validate the behavior of your app or an off-the-shelf app for secure network interactions.
- **Reverse engineering —** You can use a proxy to intercept and decode any HTTP request to discover how an app communicates with its servers.

Network proxies are appliances that usually reside at the perimeter of a secured network and act as a go-between for every request sent to a server. Many large enterprises deploy network proxies to provide content filtering, access control, virus protection, and response caching for personal computers within their network. The HTTP client, typically a browser, must be configured to use the appliance as a proxy; otherwise, it has no access to servers outside of the protected network. Figure 9-9 illustrates the sequence of activities for an HTTP request and response that traverses a proxy server.

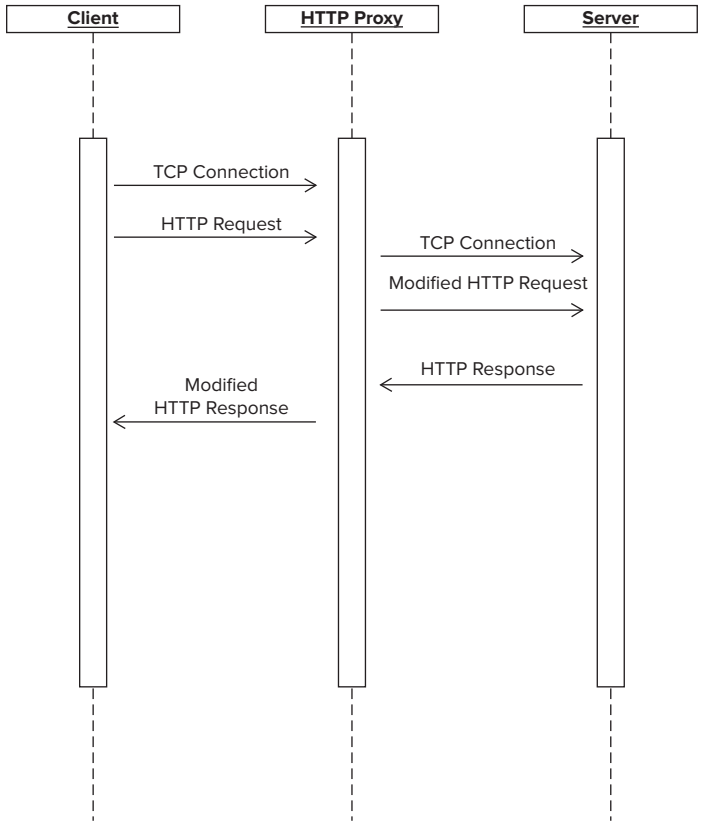


FIGURE 9-9

When a client makes an HTTP request, the client first establishes a TCP connection to the proxy server and sends the HTTP request to the proxy. The proxy applies its processing rules to the request, which may deny the request because it is destined for a blacklisted site or because the request itself has unauthorized content. If the proxy is a caching proxy, it may short-circuit the request and return a response cached from an earlier request to the same destination. If the request is acceptable, the proxy establishes a TCP connection with the true target host specified in the request and then transmits the HTTP request to the target host. If the target host responds, then the HTTP response is received by the proxy, and additional processing rules are applied. For example, the response may be scanned for viruses or cached. If the response is acceptable, it is passed back to the requesting client.

A proxy can handle HTTPS requests in a couple of different ways. First, it may just deny all HTTPS requests, but this behavior is uncommon. Alternatively, some proxy configurations pass HTTPS requests through unaltered and unfiltered. The third possible behavior is that the proxy establishes an HTTPS connection with the client and provides an SSL certificate that appears to be from the target host but is signed with the proxy's Certificate Authority (CA) certificate. The proxy is essentially performing a man-in-the-middle attack on the request, but instead of nefarious purposes, the attack is intended to protect the enterprise. For this approach to work, the client must have the proxies' CA certificate installed in its keychain. If the request is allowed, the proxy establishes an SSL connection with the target host and transmits the request securely.

There are many proxy software and hardware packages available to network engineers. These packages are designed for network engineers and have limited usefulness to app developers. Charles (`http://www.charlesproxy.com`) is a reasonably-priced proxy software package designed for developers. It is a desktop application that performs proxy services for any HTTP client. Charles allows the developer to manually intercept a request and modify the request or response. It can also be configured to automatically perform the same types of modifications on requests.

If you use an HTTP proxy to manipulate network traffic, you can manipulate only HTTP and HTTPS requests. If your app uses another protocol, a proxy will not be of assistance.

The following subsections describe the use of Charles to intercept and manipulate HTTP requests and responses.

Setting Up Charles

To use Charles, the OS X machine and the iOS device must be on the same network. They do not need to be on the same subnet, but one should be able to ping the other. The iOS device must be on a Wi-Fi network because iOS does not apply proxy settings to WWAN connections. To set-up Charles to capture traffic via a proxy, perform the following steps:

1. When you run Charles, it configures the proxy settings on the OS X machine by default. This behavior clutters the data captured from the device, so you need to disable the OS X proxy. To do this, select the Proxy Settings menu, as shown in Figure 9-10.

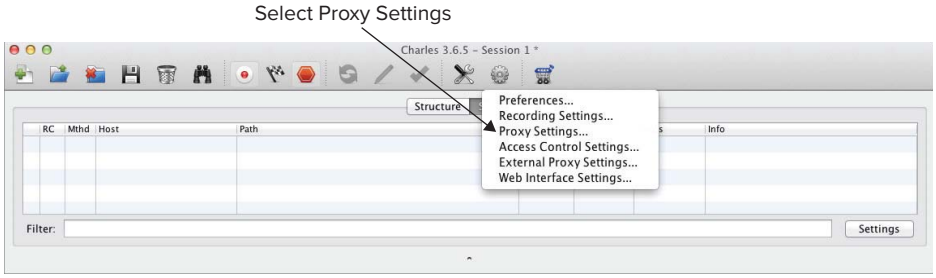


FIGURE 9-10

2. A settings dialog appears, as shown in Figure 9-11. Select the Mac OS X tab, and uncheck all the options on that view; then press OK. Charles then removes the proxy configuration from all active network interfaces.

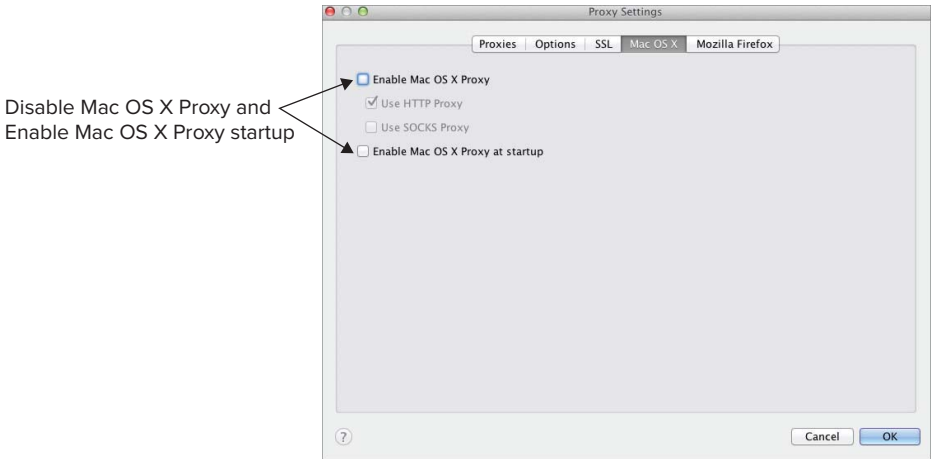


FIGURE 9-11

3. Next you must configure the iOS device to route all its HTTP traffic through the proxy. First, determine the IP address of the machine running Charles. On that machine, open a Terminal window and execute the `ifconfig` command. The following snippet shows part of the output.

```
$ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
…
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether e0:f8:47:34:98:8a
        inet6 fe80::e2f8:47ff:fe34:988a%en1 prefixlen 64 scopeid 0x5
        inet 192.168.1.34 netmask 0xffffff00 broadcast 192.168.1.255
        media: autoselect
        status: active
```

4. Look for the configuration of the `en1` interface. The IP address will be provided in that configuration data; in this case the IP address is `192.168.1.34`. The interface you use depends on your network topology. If the machine running Charles only has an Ethernet interface then you need to use `en0` as the interface.

5. Remembering this value, go to the iPhone or iPad and start the Settings app and the Wi-Fi Settings subsection. Tap on the blue disclosure indicator on the table cell of the active Wi-Fi network. At the bottom of the subsequent detail view, as shown in Figure 9-12, there is a segmented control where you can enable manual HTTP proxy configuration.

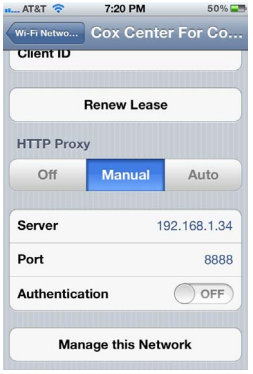
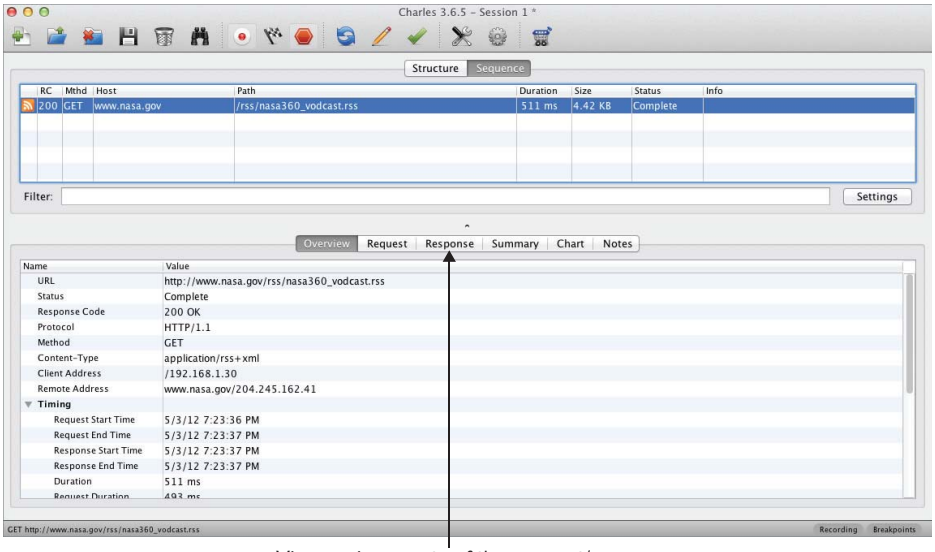


FIGURE 9-12

6. Select the Manual option, and enter the IP address that was previously in the server field. The default port for Charles is port 8888. Press the back button in that view and the proxy settings are applied. If you leave these settings active on the phone, it will block all HTTP traffic unless Charles is running at the specified address, so don't forget to reverse this setting when you complete your debugging session.

This example uses the VideoDownloader app provided with Chapter 3, which downloads an RSS feed from NASA. When run with the proxy configured and recording data, the proxy captures a single HTTP transaction. Charles groups these transactions into a single line. If you select a transaction and the Structure tab above the transaction list, you can drill down into a request and examine the request and response in detail. Figure 9-13 shows the capture of a single request from the app caught by the proxy.



View various parts of the request/response

FIGURE 9-13

HTTP Breakpoints

Charles allows you to set a breakpoint on a URL so that you can interrupt and alter the request or response. You can have multiple breakpoints set on many different URLs. To set a breakpoint, perform the following steps:

1. Ctrl+click the transaction in the list that matches the URL of the transaction you want to catch. The resulting pop-up menu, shown in Figure 9-14, contains an option to set a breakpoint on that URL.
2. Select the Breakpoints option to enable a breakpoint on that URL. On a subsequent invocation of the VideoDownloader app, Charles can catch the request and display the breakpoint window, as shown in Figure 9-15.

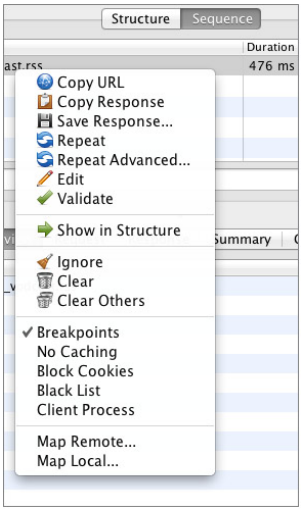


FIGURE 9-14

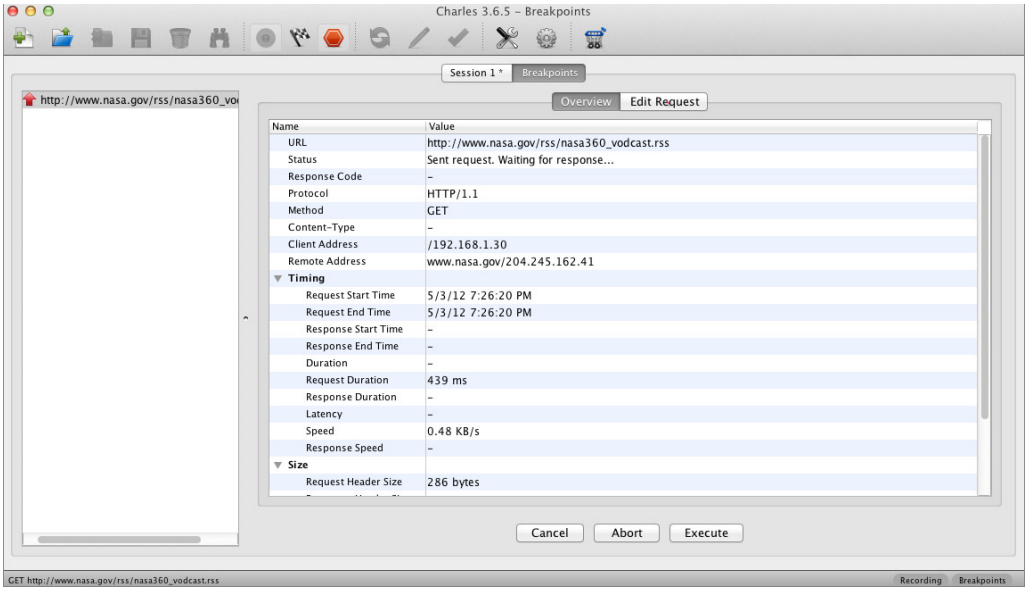


FIGURE 9-15

The request Overview panel provides summary information about the request that is extracted from the request. Some of the values are undefined because a response has not yet been received. If you select the Edit Request panel, you see the exact contents of the request that was sent by the device. On the Edit Request panel, as shown in Figure 9-16, you can manually manipulate the contents of the request.

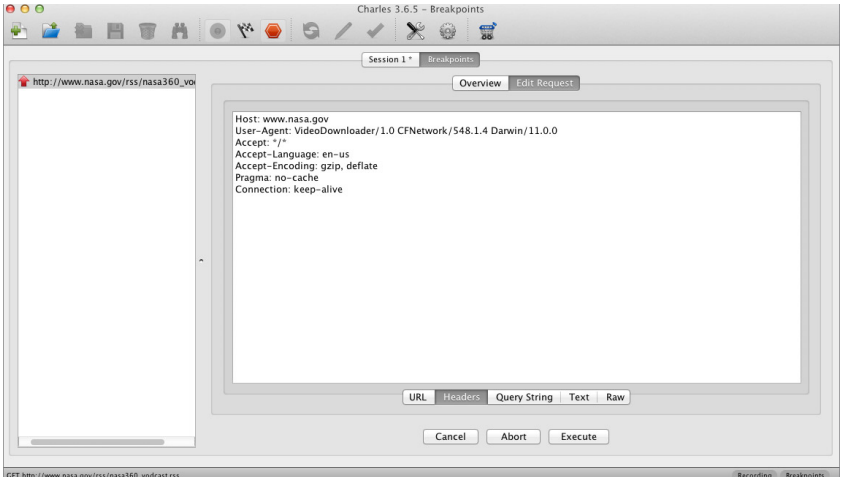


FIGURE 9-16

You can use this panel to modify the URL, headers, query string, text of the request body, or the raw bytes of the request. If you modify the URL, Charles sends the request to a different URL than that specified by the app. If you use this functionality, you can direct a single request to a test server to validate an older version of the apps behavior against a new server before deploying the server into production.

You can also add, remove, or modify headers or the body in the request. This is helpful to validate the server's response to different meta data or payload data in the request. If you press the Execute button, Charles forwards the request to the server.

If a breakpoint is set on a URL, then Charles also interrupts the transaction and displays the edit request panel, as shown in Figure 9-17, when the response to the request is received from the server. For every breakpoint you receive two interruptions.

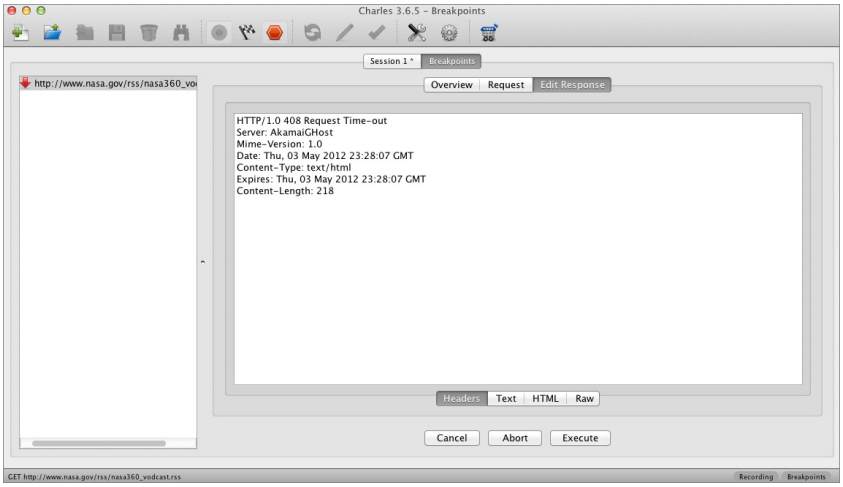


FIGURE 9-17

Using the response breakpoint panel, you can modify the HTTP response headers and body returned by the server. Like editing the request, all the content of the response may be altered. This is a powerful feature that allows you to validate an app's response to erroneous responses or simulate edge conditions that would normally be difficult to reproduce in the service tier.

If the app you test has short request timeout values, then intercepting requests with breakpoints may cause timeout errors in the app. Manual editing of requests or responses needs to be performed quickly, or by setting up a rewrite rule in Charles.

Rewrite Rules

Rewrite rules specify modifications to make to HTTP transactions automatically by Charles. A rewrite rule can be assigned to a set of URLs and contains a set of modifications to make to the request, response, or both. The modifications can be made to any part of the URL, header, or body. Each modification is described as a text pattern and replacement text, which can be regular expressions.

To add a rewrite rule select the Tools ➪ Rewrite menu option from Charles' application menu. The Rewrite Settings window is shown in Figure 9-18.

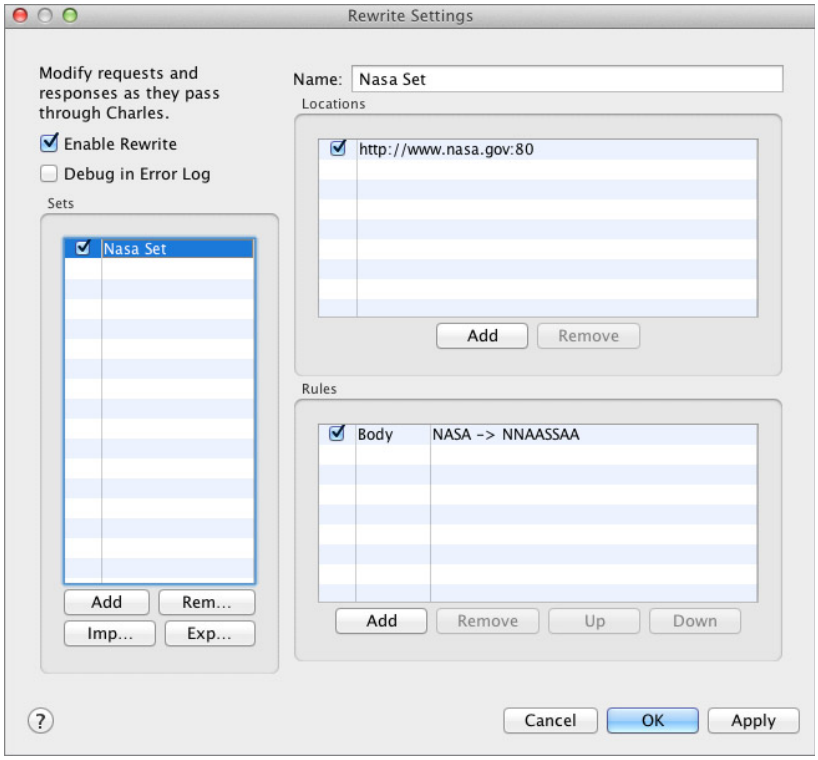


FIGURE 9-18

Using this window you can add a new rule set, add target URLs to the rule set, and create new rules. Each rule describes a text manipulation to perform on the requests and responses that match the URL. To add a rule, click the Add button on the rule section. The Rewrite Rule definition dialog, as shown in Figure 9-19, allows you to define the following:

- The part of the message to modify including headers, query parameters, URL, or body
- The phase of the transaction to which the rule applies: the request or the response
- The text or regular expression to match against
- The substitution text

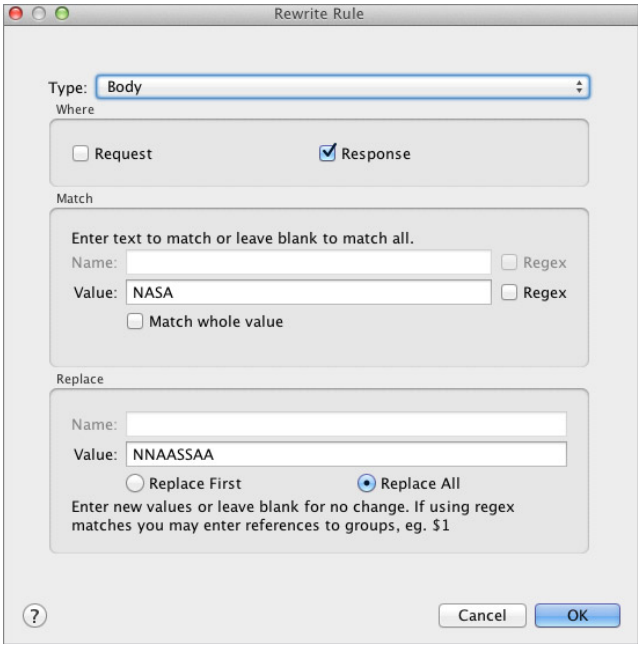


FIGURE 9-19

In Figure 9-19 the rule matches any occurrence of the text `NASA` in the response body and replaces it with `NNAASSAA`. If you apply this rule and run the VideoDownloader app you see the modified text in the descriptions of the videos provided by the RSS feed. This capability is helpful in quickly applying modifications to the HTTP data so that complex edge conditions or error scenarios can be simulated to validate the service tier or app.

Charles has many other features that you can use to manipulate the HTTP conversation between your app and a server. You can map requests to local files or other remote servers, which is helpful if the changes to a response are more complex than what can be accomplished with text replacement.

You can also intercept and modify HTTPS requests with Charles. To enable HTTPS decryption you must configure Charles, using the Proxy Settings SSL panel, to enable specific URLs for SSL decryption. You also must install Charles' CA certificate on the iOS device so that the device accepts the SSL certificates signed by Charles. The Charles CA certificate may be downloaded from `www.charlesproxy.com/charles.crt`. To install the certificate on your device, visit the download URL from the iOS device and iOS will confirm the installation of the untrusted certificate. When installed, the iOS device accepts other SSL certificates signed by Charles. Before SSL decryption is configured, the transaction contents displayed are cipher-text, which is unreadable. After successful configuration, the transaction results display as clear, readable text. This feature is invaluable for observing SSL connections from your app.

> **WARNING** *Do not leave the Charles certificate installed on a device used for nondevelopment activities. The presence of a rogue CA certificate can expose you to a man-in-the-middle attack on public networks.*

Using a proxy to intercept and modify HTTP requests is a powerful tool for simulating or generating errors to validate your application's behavior. Proxies are limited to intercepting only HTTP traffic.

SIMULATING REAL-WORLD NETWORK CONDITIONS

Network packet capture and HTTP proxy tools are excellent at providing visibility to network traffic. Another tool that all iOS developers should have in their toolbox and be familiar with is a *network traffic shaper* to simulate slow or unreliable networks.

Using a traffic shaper to degrade network performance can help you identify problems in your app. Problem areas that can be identified include:

- **Over-aggressive timeouts —** If you have set timeout values too low, testing the app on a degraded network highlights operations that would normally succeed but are aborted because of a timer expiration.
- **Missing error handling —** If you have timeouts occurring but the code to handle the timeout is missing or defective, running the app on a network that triggers the timeouts can identify those defects.
- **User interface freezing —** If your app is inadvertently making network calls on the main thread, running on a degraded network can help you detect the problem.
- **User confusion —** Running the app on a slow network during usability testing can illuminate areas where the app may not freeze but the user may be confused about what the app is doing when waiting for a response. These defects are usually resolved by user interface changes that inform the user of the activity performed by the app.

- **Cache implementation validation —** It can be difficult to quantify the improvement provided by data caching when running your app on a high-speed network. Using a traffic shaper you can quantify the improvement that your users can expect to see when caching is implemented.

OS X Lion provides a great traffic shaper called Network Link Conditioner (NLC). NLC interacts with the low-level network interface drivers of OS X and throttles the speed of every interface on the machine running it. It provides the ability to independently change the bandwidth, latency, and packet loss rate of both the received and transmitted data. NLC can also apply a delay to DNS responses.

NLC comes with several predefined network profiles that match common network types, such as Wi-Fi, 3G, and EDGE networks. Figure 9-20 shows the profile for a good 3G network. You can also define your own profiles if you need to test on networks with other types of behavior. The performance experienced through the preconfigured profiles is somewhat optimistic when compared to real-world networks; therefore, it may be necessary to apply settings slower than described by the default profiles.

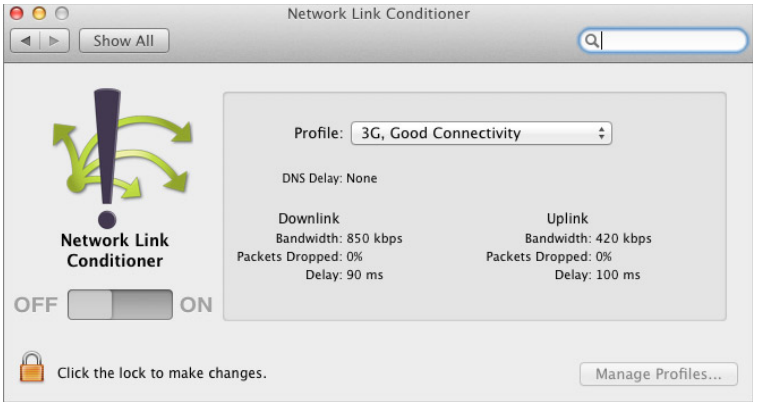


FIGURE 9-20

The NLC installation package is delivered as part of the XCode installation but not installed by default. Instead, a folder is written to `/Applications/Utilities/Network Link Conditioner` that contains the preference pane. Double-click the `Network Link Conditioner.prefPane` file to install the preference pane in your OS X System Preferences application. When installed, you can access NLC via the System Preferences.

When using NLC, it must first be unlocked and then turned on. The Profile drop-down list allows you to select a packaged or custom network profile to apply. Use the Manage Profiles button to create new profiles or edit the parameters of the packaged profiles.

For NLC to be reflected on iOS device communications, you need to run your device in a shared network topology (refer to Figure 9-2); otherwise the network packets from the device will bypass NLC. If you debug the app in the iOS Simulator, NLC actively shapes the network traffic. XCode

uses the local loopback network interface to communicate with the app running in the iOS Simulator, and NLC shapes the traffic through every network interface. Therefore, if you run a slow profile in NLC, it may take some time for your app to start in the simulator.

Although the Network Link Conditioner is useful for simulating real-world network conditions, it is not a replacement for testing in the real world. It cannot replicate the randomness of actual carrier networks or the behavior of nondeveloper users. Use NLC to perform development testing but don't bypass in-the-wild testing of your app.

SUMMARY

An enterprise-connected iOS application involves the use and cooperation of many network components outside of your control. Using tools to observe and manipulate network traffic between your app and enterprise infrastructure can help you identify and avoid defects in your application. Network sniffers help you see the truth about what data is moving between the device and any remote servers. HTTP proxies enable you to manipulate those communications to simulate new or defective conditions. Network traffic shapers help you observe how your app behaves on uncontrolled or defective networks.

10

Using Push Notifications

WHAT'S IN THIS CHAPTER?

- Interacting with local notifications
- Providing an excellent user experience with remote notifications
- Applying notification best practices to your application

WROX.COM DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code is in the Chapter 10 download and is divided into the following major examples:

- An Xcode project that includes code for local and remote notifications
- SQL script to build the database used for remote notifications
- Server-side scripts, written in PHP, to manage remote notifications

One key metric used to determine the success of an application is repeat usage. User acquisition costs can be high, and after you attract users, you must provide a predictable, nonintrusive method to inform them when you have something that requires their attention.

Push notifications are a mechanism that allows you to inform users that the application has new information for them. Notifications can take many forms: An enterprise app might inform users of a purchase order pending approval, or a game may alert users that it is their turn to play. When done properly, push notifications are a great tool to drive efficiency gains in the enterprise and engagement in commercial iOS applications.

Apple provides two notification methods: *local* and *remote*. Local notifications are *scheduled* for delivery by the system on behalf of the application. Local notifications are managed on the device and are free, meaning they do not require the overhead of a server, nor do they require

user permission. Local notifications are handled the same as remote notifications in terms of inclusion in the notification center and the overall user experience.

Alternatively, remote notifications are *registered* for delivery using the Apple Push Notification service (APNs). APNs launched in June 2009 with iOS 3.0 and makes use of a persistent IP connection for notification delivery. Remote notifications require the additional overhead of a server or third-party provider to facilitate communication with APNs. Remote notifications require explicit user approval as well as integration within business processes to be effective.

To reinforce the concepts behind local and remote notifications, this chapter's examples create a lightweight Relationship Manager akin to a Customer Relationship Manager (CRM) system. The first iteration of the application schedules reminders on the device using local notifications. From there, you create a custom remote notification service and integrate it into your Relationship Manager application.

SCHEDULING LOCAL NOTIFICATIONS

Local notifications are ideally suited for self-contained applications where information is restricted to a single device. Applications such as alarm clocks and task managers are perfect examples of applications where local notifications may be used.

Unlike remote notifications, local notifications do not require explicit user approval for delivery. As noted earlier, local notifications are delivered by the operating system, making them free to both developers and users, and they do not require an active network connection for delivery. However, local notifications are still bound by the user-configured notification preferences within the Settings application. Figure 10-1 depicts the different settings available to the user.

FIGURE 10-1

Creating Local Notifications

Local notifications are instances of `UILocalNotification` and require a `fireDate` for the system to know when to deliver the notification. Scheduling a local notification without a `fireDate` presents the notification immediately. You may also specify a `timeZone` so that notification delivery is adjusted as the user changes time zones. Reminder and task manager style applications can utilize the `repeatInterval`, which is specified using an `NSCalendarUnit` type. The following list includes the possible `NSCalendarUnit` values for a repeat interval. If specified, the system can use the value to schedule the next notification as the current one is delivered. If left blank, the default is a one-off notification that does not repeat.

- `NSEraCalendarUnit`
- `NSYearCalendarUnit`
- `NSMonthCalendarUnit`
- `NSDayCalendarUnit`
- `NSHourCalendarUnit`
- `NSMinuteCalendarUnit`

- `NSSecondCalendarUnit`
- `NSWeekdayCalendarUnit`
- `NSWeekdayOrdinalCalendarUnit`
- `NSQuarterCalendarUnit`
- `NSWeekOfMonthCalendarUnit`
- `NSWeekOfYearCalendarUnit`
- `NSYearForWeekOfYearCalendarUnit`

The user experience of a local notification will be driven by the configuration options set while creating it. The options allow you to alter the behavior of the notification, how the system delivers it, and how the application responds to being launched by a notification. Table 10-1 details the available properties for local notifications.

TABLE 10-1: UILocalNotification Properties

PROPERTY	TYPE	DESCRIPTION
`fireDate`	Date	Date and time the OS should deliver the message.
`timeZone`	Time Zone	Without specifying, `fireDate` is interpreted as GMT, which may result in a poor user experience depending on the application. Specifying this value adjusts the `fireDate` based on the current time zone.
`repeatInterval`	Calendar Unit	Specifies how often a notification should be repeated. Notifications will be rescheduled after each delivery. The default is to not repeat a notification.
`repeatCalendar`	Calendar	The calendar used during the creation of a notification. The default is to use the users' current calendar.
`alertBody`	String	Notification message.
`hasAction`	Boolean	Instructs the OS to display or not display the alert action.
`alertAction`	String	This is the title of the right-button if users have configured the alert style or the value placed next to Slide To in the lock screen. If a value is specified for `alertBody`, the default for this field is View.
`alertLaunchImage`	String	Filename of an image in the applications bundle displayed on launch of the application if triggered from the action button or by sliding the lock screen item.
`soundName`	String	The filename of the sound in the applications bundle to play upon delivery of the notification. Specifying `UILocalNotificationDefaultSoundName` uses the default system sound. Sounds are limited to 30 seconds, and anything longer results in the OS playing the system sound.

continues

TABLE 10-1 *(continued)*

PROPERTY	TYPE	DESCRIPTION
`applicationIconBadgeNumber`	Integer	The value to display as the application icons' badge number.
`userInfo`	Dictionary	Key-value store that allows you to pass custom data through your notification. Data in `userInfo` can be used to enhance the user experience by doing something like launching the app to a specific view or with a specific value.

One property that can be creatively leveraged to enhance the user experience is `alertLaunchImage`. It specifies the file that is shown as the launch image when the application is opened using the notification's action button or the launch slider on the lock screen. When paired with the `userInfo` property, `alertLaunchImage` can display a temporary view similar to the layout the user ultimately sees when the application completes the launch process. This gives your users immediate feedback while the application assembles the entire view.

After you create the notification, you are ready to schedule it with the operating system. Local notifications are scheduled using one of two `UIApplication` methods: `scheduleLocalNotification:` and `presentLocalNotificationNow:`. The first schedules a notification with the system for delivery on the `fireDate` specified during creation. The second ignores the specified `fireDate` and displays the notification to the user immediately.

Listing 10-1 covers a method to create, configure, and schedule a local notification. This method can serve as the foundation for notification scheduling within an application.

LISTING 10-1: Method to Schedule Local Notification (/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (void)scheduleNotificationWithFireDate:(NSDate*)fireDate
                                timeZone:(NSTimeZone*)timeZone
                          repeatInterval:(NSCalendarUnit)repeatInterval
                               alertBody:(NSString*)alertBody
                             alertAction:(NSString*)alertAction
                             launchImage:(NSString*)launchImage
                               soundName:(NSString*)soundName
                             badgeNumber:(NSInteger)badgeNumber
                             andUserInfo:(NSDictionary*)userInfo {

    // create notification using parameter values
    UILocalNotification *notification = [[UILocalNotification alloc] init];
    notification.fireDate = fireDate;
    notification.timeZone = timeZone;
    notification.repeatInterval = repeatInterval;
    notification.alertBody = alertBody;
    notification.alertLaunchImage = launchImage;
    notification.soundName = soundName;
    notification.applicationIconBadgeNumber = badgeNumber;
    notification.userInfo = userInfo;
```

```
    // special handling for action
    // default hasAction is YES, so if we don't have one
    // set to no. this removes button / slider
    if (alertAction == nil) {
        notification.hasAction = NO;
    } else {
        notification.alertAction = alertAction;
    }

    // schedule notification asynchronously
    dispatch_async(dispatch_get_main_queue(), ^{
        [[UIApplication sharedApplication]
         scheduleLocalNotification:notification];
    });
}
```

The method covered in Listing 10-1 schedules notifications asynchronously using *Grand Central Dispatch*, Apple's solution to managing concurrency on multicore hardware. As the number of scheduled notifications increases, the length of time to complete the scheduling processes also increases. Calling that process asynchronously allows the application to remain responsive while the notification is saved.

> **NOTE** *Although this will not be an issue for most applications, at the time of writing, there is a 64-notification limit imposed on local notifications. You can still schedule notifications, but delivery is limited to the 64 closest notifications in chronological order of* `fireDate`*, with the system disregarding the remaining notifications. This means that if you currently have 64 local notifications scheduled, scheduling another will discard the notification with the furthest* `fireDate` *from the current date. Recurring notifications count as a single notification as they are automatically rescheduled by the system. If you find yourself exceeding this limitation, you should review how you are engaging your users and whether local notifications are the correct method.*

Now that you have implemented the foundation of your notification scheduler, Listing 10-2 outlines a convenience method to schedule "client follow-up" notifications. This method calls the method created in Listing 10-1 using a combination of dynamic and static content. More specifically, you specify a consistent launch image and badge number, and ensure that no sound is used.

LISTING 10-2: Convenience Method for Scheduling Follow-up Notifications (/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (void)scheduleContactFollowUpForContact:(Contact*)contact
                                   onDate:(NSDate*)date
                                 withBody:(NSString*)body
```

continues

LISTING 10-2 *(continued)*

```
                                andAction:(NSString*)action {

    // add action to user info to help user experience on launch
    NSDictionary *userInfo = [NSDictionary dictionaryWithObjectsAndKeys:
                              contact.emailAddress, @"emailAddress",
                              contact.phoneNumber, @"phoneNumber",
                              @"contactProfile", @"type",
                              action, @"action", nil];

    [self scheduleNotificationWithFireDate:date
                                  timeZone:[NSTimeZone systemTimeZone]
                            repeatInterval:0        // don't repeat
                                 alertBody:body
                               alertAction:action
                               launchImage:@""      // contact default
                                 soundName:nil      // no sound
                               badgeNumber:1
                               andUserInfo:userInfo];

}
```

Canceling Local Notifications

Now that you've completed scheduling a notification, you need a way to cancel that notification should it become irrelevant prior to delivery. iOS provides two methods on `UIApplication` to cancel a local notification. The first, `cancelLocalNotification:`, enables you to cancel an individual notification; and the second, `cancelAllLocalNotifications:`, allows you to cancel all currently scheduled notifications, including those with repeat intervals set. Typically, you cancel notifications on an individual basis and reserve use of `cancelAllLocalNotifications:` for "reset" situations, because it will delete all notifications scheduled for the application. However, at times it may make sense to add a wrapper around `cancelLocalNotification:` so that you may delete all notifications of a certain type, or for a certain contact.

The lightweight CRM application allows you to schedule follow-up reminders for individual contacts. In the event that the contact is no longer a customer or changes positions, it should have the ability to cancel notifications for an individual contact. To accomplish this, you first need a method to retrieve all currently scheduled notifications for a given contact by looping through all local notifications and inspecting the `userInfo` property of each one. Listing 10-3 details how to retrieve all notifications for a specific contact.

LISTING 10-3: Method to Retrieve Notifications for a Contact (/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (NSArray*)notificationsForContact:(Contact*)contact {
    NSMutableArray *contactNotifications = [[NSMutableArray alloc] init];

    // get ALL scheduled notifications and loop through them
    NSArray *scheduledNotifications = [[UIApplication sharedApplication]
                                       scheduledLocalNotifications];
```

```
    for (UILocalNotification *notification in scheduledNotifications) {

        // if the email address in the notification user info matches
        // the contacts email, add it to your output
        if ([[notification.userInfo objectForKey:@"emailAddress"]
             isEqualToString:contact.emailAddress]) {

            [contactNotifications addObject:notification];
        }
    }

    return (NSArray*)contactNotifications;
}
```

Now that you have all notifications for a specific contact, you can easily cancel them. Because it is possible for users to have pending notifications for multiple contacts at the same time, you should not use `cancelAllLocalNotifications:` in this circumstance. Instead, use the `cancelLocalNotification:` method like so, where notification is an instance of `UILocalNotification`:

```
[[UIApplication sharedApplication]
         cancelLocalNotification:notification];
```

Listing 10-4 outlines the second approach, used for canceling all local notifications for a contact. This method uses the response from the method defined in Listing 10-3 as the basis for the notifications to cancel.

LISTING 10-4: Method to Cancel All Local Notifications for a Contact (/Application/RelationshipManager/RelationshipManager/Model.m)

```
- (void)cancelNotificationsForContact:(Contact*)contact {

    // retrieve all notifications for the specified
    // contact and loop through them
    NSArray *notifications = [self notificationsForContact:contact];
    for (UILocalNotification *notification in notifications) {

        // cancel the notification
        [[UIApplication sharedApplication]
         cancelLocalNotification:notification];
    }
}
```

Handling the Arrival of Local Notifications

Handling remote notifications is important to the overall user experience, therefore you must properly manage local notifications as they arrive and ensure necessary information is presented to the user. How the application responds to a notification depends on the configuration specified during creation and whether the application is currently active. If the application is not currently active and

a user selects the action button (or drags the lock screen slider) the application will launch. Just like a normal application launch, `application:didFinishLaunchingWithOptions:` is called in the application delegate, but the Options parameter contains the notification that triggered the launch.

If the application is active and implements `application:didReceiveLocalNotification:`, that method will be invoked with the triggering notification. You must understand that if the application is active, much of the configured notification behaviors are discarded, for example displaying an alert containing the `alertBody`. However, the notification still displays in the notification center provided the user has not disabled the notification center for the application. If you want to display an alert in both cases, you must manually create an alert view within `application:didReceiveLocalNotification:`.

To provide the optimal user experience, the example in this section implements both `application:didFinishLaunchingWithOptions:` and `application:didReceiveLocal Notification:`. The implementations are similar: They both inspect the notification and determine what, if any, action should be taken. Building on the client follow-up notification discussed earlier in Listing 10-2, the application responds by loading the contacts detail view. Using the contact method (call or email) specified when creating the notification, the application can also ask the user whether the application should initiate the call or e-mail. This approach enables you to easily add support for additional notification types as your business process grows.

Listing 10-5 and Listing 10-6 cover the specifics for each implementation.

LISTING 10-5: Local Notification Handling Within Launch Sequence (/Application/ RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
      didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // determine if app launched from a local notification
    UILocalNotification *localNotification =
        [launchOptions objectForKey:
         UIApplicationLaunchOptionsLocalNotificationKey];

    if (localNotification != nil) {
        NSDictionary *userInfo = localNotification.userInfo;

        NSString *action = [userInfo objectForKey:@"action"];
        Contact *contact = [[Model sharedModel]
                            contactWithEmailAddress:
                            [userInfo objectForKey:@"emailAddress"]];

        // initiate a phone call
        if ([action isEqualToString:@"Call"]) {
            NSString *phone = [NSString stringWithFormat:@"tel:%@",
                               contact.phoneNumber];

            [[UIApplication sharedApplication]
             openURL:[NSURL URLWithString:phone]];

        // start an email
```

```
        } else if ([action isEqualToString:@"Email"]) {
            NSString *email = [NSString stringWithFormat:@"mailto:%@",
                               contact.emailAddress];

            [[UIApplication sharedApplication]
             openURL:[NSURL URLWithString:email]];

        }

    }
    ...
}
```

LISTING 10-6: Local Notification Handling When Application Is Active (/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (void)application:(UIApplication*)application
     didReceiveLocalNotification:(UILocalNotification *)notification {

    // alert the user that a notification was received
    // because the user was in the application, we present
    // them with some additional information
    dispatch_async(dispatch_get_main_queue(), ^{
        NSDictionary *userInfo = notification.userInfo;

        NSString *action = [userInfo objectForKey:@"action"];
        Contact *contact = [[Model sharedModel]
                            contactWithEmailAddress:
                            [userInfo objectForKey:@"emailAddress"]];

        [UIAlertView alertViewWithTitle:@"Reminder"
                                message:notification.alertBody
                      cancelButtonTitle:@"Cancel"
                      otherButtonTitles:[NSArray
                                         arrayWithObjects:@"View Contact",
                                         action, nil]
                  onDismiss:^(int buttonIndex)
                     {
                         // display the contact details
                         if (buttonIndex == 0) {

                             ContactDetailTableViewController *contactVC =
                                [[ContactDetailTableViewController alloc]
                                 initWithStyle:UITableViewStyleGrouped];

                             contactVC.contact = contact;
                             contactVC.presentedModally = YES;

                             UINavigationController *nc =
                                [[UINavigationController alloc]
```

continues

LISTING 10-6 *(continued)*

```
                                    initWithRootViewController:contactVC];

                                [self.navigationController
                                 presentModalViewController:nc animated:YES];

                            // initiate the selected action
                            } else if (buttonIndex == 1) {
                                // initiate a phone call
                                if ([action isEqualToString:@"Call"]) {
                                    NSString *phone =
                                    [NSString stringWithFormat:@"tel:%@",
                                     contact.phoneNumber];

                                    [[UIApplication sharedApplication]
                                     openURL:[NSURL URLWithString:phone]];

                                // start an email
                                } else if ([action isEqualToString:@"Email"]) {
                                    NSString *email =
                                    [NSString stringWithFormat:@"mailto:%@",
                                     contact.emailAddress];

                                    [[UIApplication sharedApplication]
                                     openURL:[NSURL URLWithString:email]];

                                }
                            }
                        }
                    onCancel:^()
                        {
                            // don't do anything for cancel
                        });
}
```

Listing 10-6 uses a category on `UIAlertView` written by Mugunth Kumar that consolidates the `UIAlertView` display and delegate methods. This is one approach that enables you to handle `UIAlertView` input with data from the triggering notification. These categories are available at `https://github.com/MugunthKumar/UIKitCategoryAdditions`.

> **WARNING** *At the time of writing, there is a known issue within the iOS Simulator where the* `application:didReceiveLocalNotification:` *is fired twice, split seconds apart. Any logic placed within* `application:didReceiveLocalNotification:` *is executed twice unless you put validation in place. One solution to this is to test local notification functionality on a device.*

Now that you have implemented local notifications, you can learn how to enhance the Relationship Manager application using remote notifications.

REGISTERING AND RESPONDING TO REMOTE NOTIFICATIONS

Although local notifications can be a great solution in certain situations, they are limited because the application must be running to schedule the notification. Local notifications are also restricted to the device that scheduled them. Remote notifications solve this problem by enabling you to initiate a device notification from an external server, which is then delivered using the APNs.

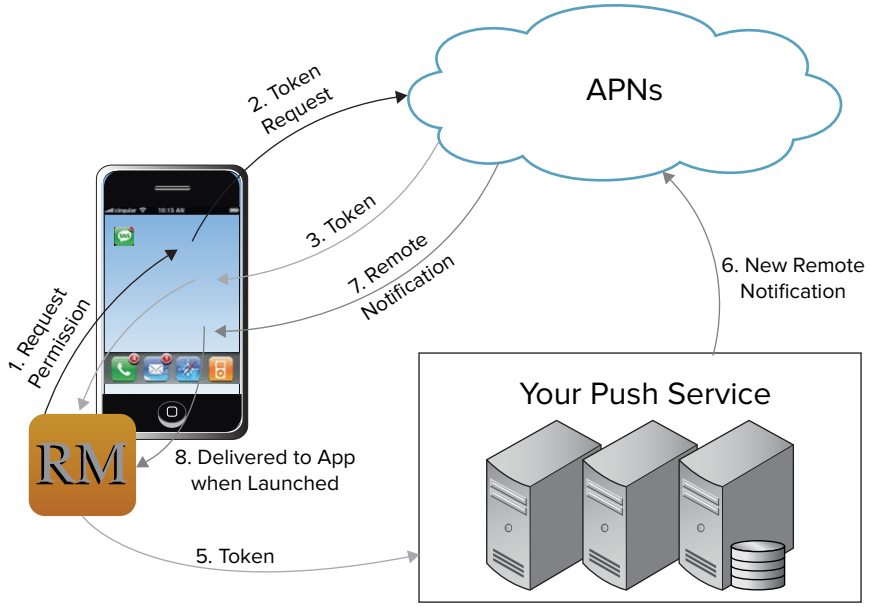
Remote notifications allow for greater flexibility because they can be initiated outside of the application. For example, assume your customers are in the receiving department at a given company and there is a new requirement to send purchase approvers a notification when something they approved is received at the warehouse. Remote notifications allow you to accomplish this without the application actively running. If you were using local notifications or custom alerts, you would need to change the application; distribute a new version to all users; and the biggest pitfall: They must actively be using the application to be notified.

This section covers implementing a custom remote notification service as well as how to register for and respond to remote notifications within an application. The custom notification service connects with APNs to register a notification for delivery. Although you can follow along with the examples in this section, to test your implementation you need two things:

- **A paid Apple Developer membership:** The iOS Simulator does not handle remote notifications, so you will need to install the test application on a device. In addition, you need to request an SSL certificate from Apple for your notification server, which is covered in the "Configuring Remote Notifications" section later in this chapter.
- **Access to a web-server connected to the Internet and running PHP and MySQL:** For testing purposes, a server running on your Mac should suffice.

APNs is the single communication gateway that controls all remote notification delivery. This gateway is a common interface for both consumer (distributed via the App Store) and enterprise applications. Figure 10-2 outlines the entire remote notification process as it is covered in this chapter. If you choose to outsource your remote notification delivery, your diagram will look slightly different.

The process begins when an application requests permission to deliver notifications. This request is made to the operating system, which then prompts the user for permission. If the user grants permission, the operating system fetches a device token from APNs, which is delivered to the application and subsequently stored at the application's remote push service. When an appropriate event occurs, the application's remote push service registers a remote notification with APNs using the previously retrieved device token. If the notification is successfully registered, APNs makes a "best effort" attempt at delivering the notification. With so many variables such as network connectivity and device status, it is not possible to determine whether a notification was delivered though. If things go accordingly, notifications are delivered to the device. If the application is open, the operating system



Adapted from Apple Developer Documentation

FIGURE 10-2

relays the notification to the active device for handling, as discussed in the previous section. If the application is not active, the operating system displays the notification based on the user's configured settings.

In practice, APNs delivers remote notifications to a *token* that is associated with a single device. It is the responsibility of the developer to obtain this token and provide it as the delivery destination when registering a notification with APNs. Permission to deliver remote notifications is at the discretion of each user; therefore, developers must design their applications to degrade gracefully. Remote notifications should enhance the experience, but the application should continue to function as expected without them should permission not be granted.

Configuring Remote Notifications

Before you can begin using APNs, you must configure remote notifications for each application within the iOS Provisioning Portal. This section walks you through the steps required to configure remote notifications for the application and obtaining the necessary certificate files to communicate with APNs and register notifications for delivery.

1. First you need to generate a Certificate Signing Request (CSR) using the Keychain Access application on your computer. A CSR is a request for an identity certificate from a certificate authority, which for APNs is Apple. Figure 10-3 outlines how to initiate the CSR.

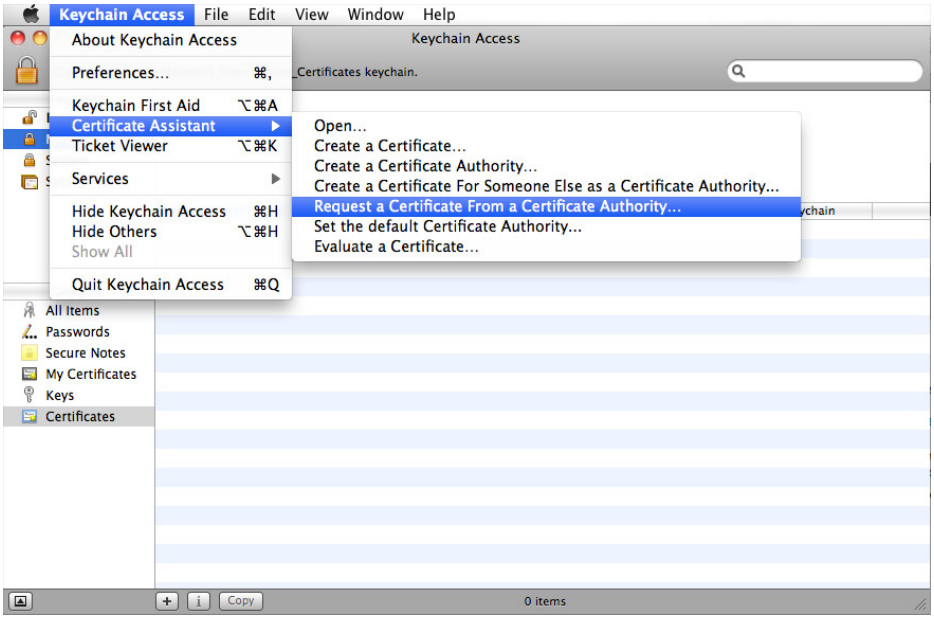


FIGURE 10-3

2. After you start the CSR wizard, you will be presented with a screen similar to Figure 10-4. Enter your e-mail address and a descriptive name for the request. This name will also be tied to the private key generated along with the CSR. Choosing a descriptive name makes locating the private key in the future much easier. You need your private key to connect with APNs.

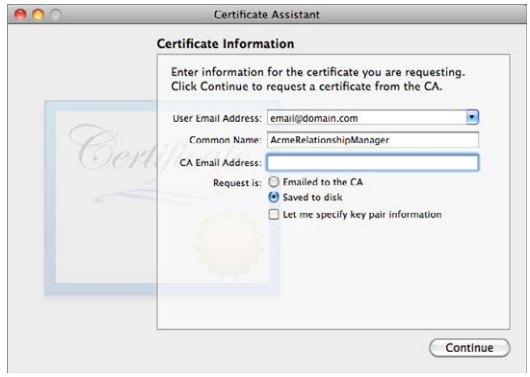


FIGURE 10-4

> **NOTE** *It may be worthwhile for you to save your CSR. APNs certificates, development, and production have lifespans of one year. Saving your CSR makes the renewal process easier and could help avoid service interruptions.*

3. After your CSR has been generated, select the new private key, and export it to a safe, memorable place. You can export the private key by Ctrl+clicking the private key entry in Keychain Access and choosing the export option, but be sure to remember the export password you set.

4. Now, open the iOS Provisioning Portal at `https://developer.apple.com/ios/` in your browser, and navigate to the application list. Because you are enhancing an existing application, the one created in the last section, you should already have an entry in the list for Relationship Manager. The entry should be similar to Figure 10-5. After you locate the application in the list, click the Configure action link in the Action column.

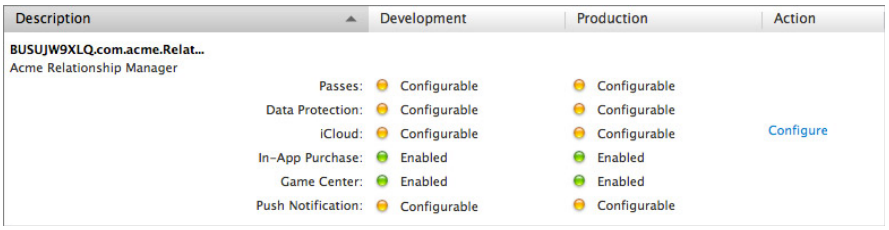


FIGURE 10-5

5. Clicking the Configure action displays a screen similar to Figure 10-6. This view is where you configure notifications for both development and production. Check the Enable for Apple Push Notification Service box and then choose the Configure option in the right-most column.

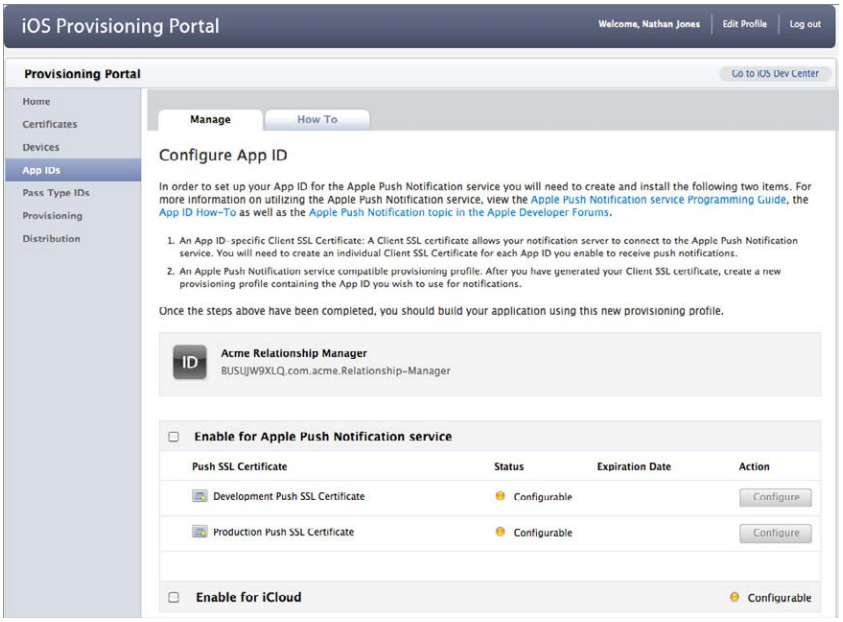


FIGURE 10-6

6. After you select Continue, a screen similar to Figure 10-7 appears. Because you have already generated your CSR (refer to Figure 10-4) you can click Continue through this view.

FIGURE 10-7

7. After clicking through the screen in Figure 10-7, you see another screen, similar to Figure 10-8, with a prompt to choose your CSR. Select the Choose File button, navigate to your CSR, select it, and choose the Generate button.

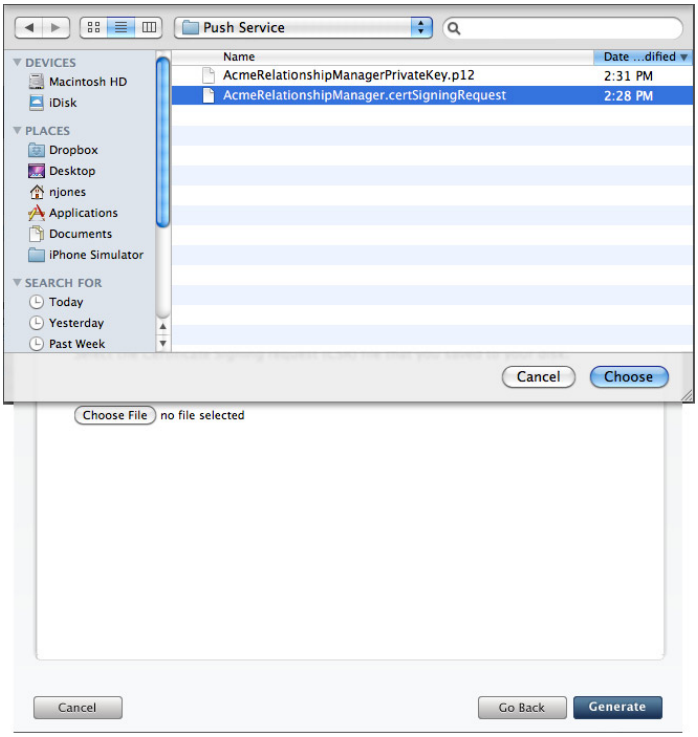


FIGURE 10-8

8. If everything works as planned, you see a slight overlay while Apple generates your APNs certificate and then a success message that resembles Figure 10-9. Click the Continue button to navigate to the next screen, and download your certificate, as shown in Figure 10-10.

FIGURE 10-9

FIGURE 10-10

9. Navigate back to the list of applications; you should now see that push notifications have been successfully configured for development, as shown in Figure 10-11.

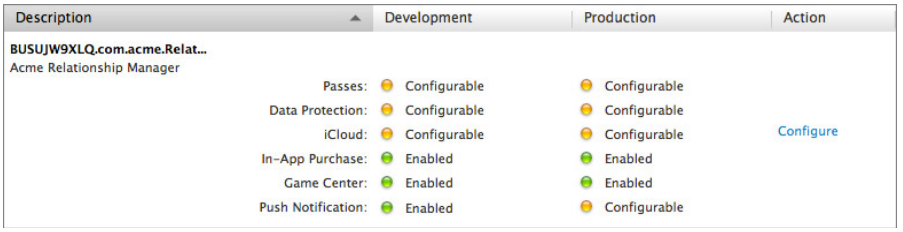


FIGURE 10-11

> **NOTE** *Application builds (development and distribution) that use remote notifications must be signed with a provisioning profile configured for remote notifications. If you have previously issued provisioning profiles for the application, you should re-issue them after configuring remote notifications. You cannot request permission to deliver notifications until the application is signed correctly.*

Because this chapter uses PHP for the service tier, you need to combine the private key created along with the CSR (refer to Figure 10-4) and the SSL certificate provided to you by Apple (refer to Figure 10-10) into a single PEM format file. You can use the PEM file format to specify a certificate, private key, or both concatenated together. In other server-side languages these steps may not be required,

but the function used to initiate an APNs connection in this chapter, `stream_context_create()`, requires that the certificate be in the PEM format. The following steps outline one way to convert and combine the two files:

1. Open the Terminal application on your computer, and navigate to the directory where you saved the two files. For example purposes, assume the files are in the Push Service directory on the desktop as shown here:

```
$ cd /Users/njones/Desktop/Push\ Service/
```

2. Convert the SSL certificate downloaded from Apple into the PEM format like so:

```
$ openssl x509 -inform der –in AcmeRelationshipManager.cer –out
AcmeRelationshipManagerCert.pem
```

3. Convert the private key from PKCS12 (.p12) format to PEM format like so:

```
$ openssl pkcs12 -in AcmeRelationshipManagerPrivateKey.p12 -out
AcmeRelationshipManagerKey.pem -nocerts
```

4. You will be prompted to enter the import password, which is the password you chose when exporting the private key from Keychain Access. After successfully entering the import password, you will be prompted to enter a PEM passphrase and then verify it. This should be secure, and you need to store it for use when connecting to APNs.

5. Finally, combine the certificate and private key PEM files into a single file like so:

```
$ cat AcmeRelationshipManagerCert.pem AcmeRelationshipManagerKey.
pem > AcmeCertKey.pem
```

> **NOTE** *You can test your connection to APNs by using the* `openssl s_client` *command from Terminal. The* `s_client` *command enables you to connect to SSL servers. APNs has two servers: development and production, available at* `gateway.sandbox.push.apple.com:2195` *and* `gateway.push.apple.com:2195`*, respectively. The production endpoint is used for applications signed for distribution. Visit* `www.openssl.org/docs/apps/s_client.html` *for more information on the* `s_client` *command.*

Remote notifications are now configured, and you have the necessary files for the server to connect to APNs and register notifications. Now, you need to request permission to deliver notifications to your users.

Registering for Remote Notifications

Before you can send your first notification, the remote server must be configured to register each user's device. Users can have more than one device, and your storage method must allow for it. Listing 10-7 includes a set of SQL statements that you can use to create the foundation for storing this data on the remote server.

LISTING 10-7: Database Structure for Remote Notification Handling (/Push Server/pushdatastructure.sql)

```
 CREATE TABLE IF NOT EXISTS 'users' (
  'userid' varchar(120)
       NOT NULL,
  'datecreated' timestamp
       NOT NULL DEFAULT '0000-00-00 00:00:00',
  PRIMARY KEY ('userid')
) ENGINE=MyISAM DEFAULT CHARSET=utf8;

CREATE TABLE IF NOT EXISTS 'user_tokens' (
  'userid' varchar(120) NOT NULL,
  'token' varchar(64) NOT NULL,
  'datecreated' timestamp
       NOT NULL DEFAULT '0000-00-00 00:00:00',
  'dateremoved' timestamp
       NOT NULL DEFAULT '0000-00-00 00:00:00'
       COMMENT 'date the feedback service was polled for token',
  PRIMARY KEY ('userid','token')
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

After building the database, you should create a simple web-service script to register new users and devices. This script creates the user-device relationship used to determine which devices a notification should be delivered to. Listing 10-8 outlines a simple PHP script to register users and their device. This is a single script that first determines if the user on the inbound request already exists. If the user does not exist, the script creates the users and then register the device.

LISTING 10-8: Server-side Script to Register Users and Devices (/Push Server/register.php)

```
<?php
    ...

    // get the post body
    $userid = $_REQUEST['user'];
    $token = $_REQUEST['token'];

    if (empty($userid) || empty($token)) {
        sendAPIResponse(400);
        return;
    }

    // determine if user exists
    $sql = "SELECT userid
            FROM users
            WHERE userid='".$userid."' LIMIT 1;";
    $query = mysql_query($sql, $dbConnection);
    $userExists = mysql_fetch_row($query);

    // add a 'user' record
    if (!$userExists) {
```

```
        $sql = "INSERT INTO users (userid, datecreated)
                VALUES ('".$userid."', '".$now."');";
        if (!mysql_query($sql, $dbConnection)) {
            // return error
            sendAPIResponse(400);
            return;
        }
    }

    // determine if token already exists
    $sql = "SELECT token
            FROM user_tokens
            WHERE userid='".$userid."'
            AND token='".$token."' LIMIT 1;";
    $query = mysql_query($sql, $dbConnection);
    $tokenExists = mysql_fetch_row($query);

    // add a token for current user
    if (!$tokenExists) {

        $sql = "INSERT INTO user_tokens (userid, token, datecreated)
                VALUES ('".$userid."','".$token."','".$now."');";
        if (!mysql_query($sql, $dbConnection)) {
            // return error
            sendAPIResponse(400);
            return;
        }
    }

    // close the database connection
    mysql_close($dbConnection);

    // return success
    sendAPIResponse(200);
?>
```

Once you have a registration script in place, it's time to request permission from the application's user to deliver remote notifications. This must be repeated for each application on each of the user's devices. Listing 10-9 shows how to initiate the approval process for remote notifications as part of the application launch process. Refer to the "Understanding Notification Best Practices" section for best practices on requesting remote notification permissions.

LISTING 10-9: Request Permission to Deliver Remote Notifications (/Application/Relationship Manager/RelationshipManager/AppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
      didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    // request permission to deliver remote notifications, if needed
    BOOL requested = [[NSUserDefaults standardUserDefaults]
                      boolForKey:kPushTokenTransmitted];
    if (requested != YES) {
        [[UIApplication sharedApplication]
```

continues

LISTING 10-9 *(continued)*

```
            registerForRemoteNotificationTypes:
            (UIRemoteNotificationTypeAlert |
             UIRemoteNotificationTypeBadge |
             UIRemoteNotificationTypeSound)];
        }
        ...
    }
```

After the application launches, you see an alert similar to Figure 10-12 prompting the user to approve the delivery of remote notifications.

FIGURE 10-12

Table 10-2 outlines the four remote notification types that you can request permission to deliver. As you can see from Listing 10-9, you can request any combination of the four notification types. The user can change these permissions anytime using the Settings application and restrict what can and cannot be triggered as part of your remote notification. For example, if a user has disabled sound for the application, no sound plays during delivery even if the notification payload specifies a sound file. The application should be designed such that any notification type can be enabled or disabled independently.

TABLE 10-2: Remote Notification Types

TYPE	DESCRIPTION
`UIRemoteNotificationTypeAlert`	This permission allows access to display an alert view or banner depending on the users' configuration. This alert is also what may display in the users' Notification Center and Lock Screen if they have configured it.
`UIRemoteNotificationTypeBadge`	This permission allows access to set the application's icon badge.
`UIRemoteNotificationTypeSound`	This permission allows access to play a short sound when either an alert or badge notification is delivered.
`UIRemoteNotificationTypeNewsstandContentAvailability`	This permission allows access to notify the application when there is new content available for download via the Newsstand framework.

After a user grants permission, the application invokes the `application:didRegisterForRemoteNotificationsWithDeviceToken:` method on the application delegate. Conversely, if a user rejects the request, the `application:didFailToRegisterForRemoteNotificationsWithError:` method is called. In `application:didRegisterForRemoteNotificationsWithDeviceToken:` the application should create a request to the remote server script covered in Listing 10-8 to register the user and the device token.

Listing 10-10 demonstrates how to retrieve the device token and one approach for transmitting it to the script created in Listing 10-8. If you chose to implement server-side localization, discussed in the Remote Notification Payloads section, this is where you would initially retrieve the users' locale and transmit it along with the token. A best practice is to store an indicator that the token has been successfully transmitted to the notification provider. This helps avoid unnecessary calls to the server because subsequent calls to `registerForRemoteNotificationTypes:` after permission has been granted invokes `application: didRegisterForRemoteNotificationsWithDeviceToken:`. You can see that Listing 10-10 sets a `BOOL` if the call to Listing 10-8 is successful.

LISTING 10-10: Handling Remote Notification Approval (/Application/RelationshipManager/RelationshipManager/AppDelegate.m)

```
- (void)application:(UIApplication *)application
      didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken {
    // hardcode the current user, this would typically
    // be a token or value retrieved after they logged
    // in to use the app
    NSString *userId = @"nate@emaildomain.com";
    NSString *token = [NSString stringWithFormat:@"%@", deviceToken];

    // clean the token
    token = [token stringByTrimmingCharactersInSet:
             [NSCharacterSet characterSetWithCharactersInString:@"<>"]];
    token = [token stringByReplacingOccurrencesOfString:@" "
                                             withString:@""];

    // handle the request off the main thread
    dispatch_async(dispatch_get_main_queue(), ^{

        // build the post body
        NSString *postBody = [NSString
                              stringWithFormat:@"user=%@&token=%@",
                              userId,
                              token];
        // build the request
        NSString *endpoint = @"http://yourdomain.com/push/register.php";
        NSMutableURLRequest *request =
            [[NSMutableURLRequest alloc]
             initWithURL:[NSURL URLWithString:endpoint]
             cachePolicy:NSURLRequestReloadIgnoringCacheData
             timeoutInterval:30.0];

        // configure the remaining request properties
        request.HTTPMethod = @"POST";
        request.HTTPBody = [postBody
                            dataUsingEncoding:NSUTF8StringEncoding];
        [request setValue:@"application/x-www-form-urlencoded"
       forHTTPHeaderField:@"Content-Type"];

        NSError *error = nil;
```

continues

LISTING 10-10 *(continued)*

```
            NSHTTPURLResponse *response;

            // this method returns NSData, but in this case
            // we don't care about it
            [NSURLConnection sendSynchronousRequest:request
                                  returningResponse:&response
                                              error:&error];

            // verify we got a success
            if (response.statusCode == 200) {

                // save our local flag so that we don't
                // hit this logic each time the app is opened
                [[NSUserDefaults standardUserDefaults]
                 setBool:YES forKey:kPushTokenTransmitted];
                [[NSUserDefaults standardUserDefaults] synchronize];

            // alert the user if we didn't get a success
            } else {
                [[[UIAlertView alloc] initWithTitle:@"Error"
                                            message:@"Unable to "
                                           delegate:nil
                                  cancelButtonTitle:@"OK"
                                  otherButtonTitles:nil] show];
            }

        });
    }
```

Remote Notification Payloads

APNs payloads are JSON objects with a strict limit of 256 bytes. It is your responsibility to ensure that the payload does not exceed this limit, or the notification will be rejected by APNs. Payloads are key-value pairs with a top-level namespace `aps`. The following snippet outlines a standard APNs payload structure in human-readable form.

```
{
    "aps" : {
        "alert" : "New delivery received.",
        "badge" : 1,
        "sound" : "delivery.caf"
    }
}
```

Given that each byte counts against the 256 byte limit, one best practice is to remove space and newline characters. The following snippet depicts the condensed payload outlined in the previous snippet.

```
{"aps":{"alert":"New delivery received.","badge":1,"sound":"delivery.caf"}}
```

The `aps` namespace contains any of the three available properties: `alert`, `badge`, and `sound`. Table 10-3 details how each property is used. Note that the value passed for the `alert` property can either be a string or a dictionary. Passing a dictionary allows you to further configure how the notification appears and how the application functions as a result. Table 10-4 discusses the various properties available for an `alert` dictionary.

TABLE 10-3: Possible aps Dictionary Properties

PROPERTY	TYPE	DESCRIPTION
`alert`	String or Dictionary	When a string is passed, it is used as the message body and two buttons are used: Close and View. Refer to Table 10-4 for possible values available to you when passing a dictionary.
`badge`	Number	This is the value to appear on the application icon. No logic (for example, increment or decrement) is performed on this number; it appears as sent. Therefore, displaying an `unread` count requires that you maintain the read status on your server and badge appropriately. If nothing is sent in this property, the system removes any existing badge value.
`sound`	String	This is the name of the sound file to play as the notification displays. This file must exist in the application bundle. Supported formats include .aiff, .wav, and .caf.

TABLE 10-4: Available Child Properties of alert

CHILD-PROPERTY	TYPE	DESCRIPTION
`body`	String	Notification message.
`action-loc-key`	String	If specified, this value modifies the display of the alert by adding a second button with this value being the text of the right button. If not specified, a single button is used.
`loc-key`	String	Key to a localized value that will be used as the notification body. Values for this key can be formatted with `%@`, `%n`, and `%$` to accommodate argument values from `loc-args`.
`loc-args`	Array	Array of strings to replace format placeholders in the value set/retrieved for `loc-key`.
`launch-image`	String	Filename of an image in the applications bundle that displays on launch of the application if triggered from the action button or by sliding the lock screen item.

There are two methods for localizing remote notifications: server-side and within the application. Server-side localization involves translating the message body and action button text prior to delivery. This strategy also requires that you maintain a reference to the users' current locale so that messages are properly translated.

Localizing remote notifications within the application does require up-front planning but only requires that you specify a value for the `loc-key` parameter, which is the key to the localized value used as the message body text. You can also provide an optional array of arguments to display in the localized message body using the `loc-args` property. Localized messages are still subject to the 256 byte payload restriction. This approach has the added benefit of requiring only the key to a message instead of the entire message body. Depending on if the language is localized, this could be a significant advantage considering the payload constraints.

After configuring the `aps` namespace, you can also create a custom namespace to deliver information specific to the application. Custom namespaces are similar to the `userInfo` property on `UILocalNotification` discussed in the previous section. Custom namespaces must be included at the same level as the `aps` namespace in the payload structure. Note that the 256 byte payload restriction is enforced on the entire payload, not just the data in the `aps` namespace. You must consider this restriction when designing your remote notifications to avoid size rejections. The following code provides an example of how to include custom namespaces within the notification payload:

```
{
    "aps" : {
        "alert" : "New delivery received.",
        "badge" : 1,
        "sound" : "delivery.caf"
    },
    "emailAddress" : "nate@prospect.com",
    "action" : "Email"
}
```

Sending Remote Notifications

With your users' device successfully registered, it is time to send a notification. To register notifications for delivery, you must connect to APNs using the SSL certificate created during the configuration process discussed earlier in this chapter. The approach covered in this section includes two key server-side scripts: one to handle the APNs connection and another that provides a simple interface for initiating notifications. Listing 10-11 demonstrates how to connect to APNs, construct the binary payload using the *simple format*, and register a notification.

LISTING 10-11: Method to Connect with APNs (/Push Server/apns.php)

```
function sendPushNotification ($token, $payload) {
        $certificate = "../Path/To/Certificate/AcmeCertKey.pem";
    $passphrase = "YourPassphrase";
    $endpoint = "ssl://gateway.sandbox.push.apple.com:2195"

    $context = stream_context_create();
    stream_context_set_option ($context,
                               'ssl',
                               'local_cert',
                               $certificate);

    stream_context_set_option ($context,
```

```
                                'ssl',
                                'passphrase',
                                $passphrase);

    // connect to APNs server
    $conn = stream_socket_client(
                $endpoint,
                $err,
                $errstr,
                60,
                STREAM_CLIENT_CONNECT | STREAM_CLIENT_PERSISTENT,
                $context
    );

    if (!$conn) {
        echo "Connection to APNs Failed...";
        return;
    }

    // build the binary
    $message = chr(0) .
               pack('n', 32) .
               pack('H*', $token) .
               pack('n', strlen($payload)) .
               $payload;

    // push the notification
    $result = fwrite($conn, $message, strlen($message));

    // close the connection to APNs
    fclose($conn);

    if ($result) {
        echo "Notification sent...";
    } else {
        echo "Error sending notification...";
    }
}
```

To address some of the issues with the simple format, Apple introduced the *enhanced format*. The enhanced format enables developers to specify notification expiration dates and retrieve additional details if an error occurs.

APNs stores the last notification registered for each application on a device. If the device was offline, this allows APNs to forward that notification when connected. The issue is that the material in the notification can become dated and irrelevant to the point that you no longer want it to be delivered. The expiration date is the fixed time at which APNs can discard the message because it is no longer valid. You can instruct APNs not to store the notification by specifying a value of zero or less.

With the simple format, if an error occurs, APNs severs the connection without any indication as to what happened. Using the enhanced format allows developers to assign an identifier to each notification being transmitted. If an error occurs, APNs returns an error response along with the assigned identifier for further investigation. Table 10-5 details the different APNs response codes and their meaning.

TABLE 10-5: APNs Enhanced Format Response Codes

RESPONSE CODE	DESCRIPTION
0	No errors encountered
1	Processing error
2	Missing device token
3	Missing topic
4	Missing payload
5	Invalid token size
6	Invalid topic size
7	Invalid payload size
8	Invalid token
255	None (Unknown)

Listing 10-12 covers a simple script that can be called from a browser to register a notification. The script retrieves all the devices associated with the user being notified and calls the script created in Listing 10-11 to handle APNs communication. In practice, a script like this would not be accessible to the public; it would be placed behind some type of authentication mechanism.

LISTING 10-12: Remote Notification Test Delivery Script (/Push Server/ sendNotification.php)

```
<?php

    ...

    // get request parameter values
    $userId = $_REQUEST['userid'];
    $contactEmail = $_REQUEST['contact'];
    $message = $_REQUEST['message'];
    $badge = $_REQUEST['badge'];
    $sound = $_REQUEST['sound'];

    // clean up the action value
    $action = $_REQUEST['action'];
    $action = (!empty($action)) ? $action : "View";

    // get token(s) for user
    $sql = "SELECT token
            FROM user_tokens
            WHERE userid='".$userId."';";
```

```
    $query = mysql_query($sql, $dbConnection);

    // send push to ALL devices that belong to user
    while ($row = mysql_fetch_array($query)) {
        // create the payload
        $alert['body'] = $message;
        $alert['action-loc-key'] = $action;

        $aps['alert'] = $alert;

        // add sound
        if (!empty($sound)) {
            $aps['sound'] = $sound;
        }

        // add badge
        if (!empty($badge)) {
            $aps['badge'] = intval($badge);
        }

        $payload['aps'] = $aps;

        // add custom namespace fields
        $payload['emailAddress'] = $contactEmail;

        // connect to apns and send push
        sendPushNotification ($row['token'], json_encode($payload));
    }

    // close the database connection
    mysql_close($dbConnection);
?>
```

With the server-side components in place and the device registered, you can now test everything. Call the script created in Listing 10-12 with the following parameter values. If all goes as expected, you should receive a notification shortly similar to Figure 10-13.

- `userid`: ***nate@emaildomain.com***
- `contact`: ***John@prospect.com***
- `message`: ***Email John about proposal***
- `badge`: `1`
- `action`: **`Email`**

FIGURE 10-13

The `userid` value in the preceding parameter list matches the hardcoded value in the registration process covered in Listing 10-10. The value specified for `contact` should match an entry that has been added to the Relationship Manager application. This ensures that you trigger the handling logic covered in the next section.

Depending on your requirements and especially if you expect significant throughput in your push service, you should consider using a queued

message approach to improve flexibility and efficiency. This involves an additional step in the server-side script-flow to queue notifications to a new database table for delivery rather than connect with APNs immediately. A background process would then process the queued notifications, establish a single connection to APNs, and register and then archive the notifications.

Apple advertises that it actively monitors APNs connections and may consider constantly establishing connections as a denial of service attack. Please review the APNs Provider Communication documentation on the developer portal at `https://developer.apple.com/ios` for additional information. The preceding enhancement helps optimize your APNs connections and provides you with the flexibility to implement many of the best practices discussed later in this chapter, such as "Do Not Disturb" and server-side localization.

Responding to Remote Notifications

The process to handle remote notifications is the same as handling local notifications with the exception of the delegate method invoked and how the data for the `UIAlertView` is retrieved. How the application responds is still driven primarily by the payload delivered from the notification provider and whether the application is active. If the application is not currently active and the user selects the action button (or drags the lock screen slider) the application launches. Just like a standard launch, the `application:didFinishLaunchingWithOptions:` method is invoked in the application delegate. You can retrieve the remote notification payload from the `launchOptions` dictionary using the key `UIApplicationLaunchOptionsRemoteNotificationKey`. Listing 10-13 demonstrates how to add custom notification handling during the application launch process.

LISTING 10-13: Remote Notification Handling Within Launch Sequence (/Application/Relationship Manager/RelationshipManager/AppDelegate.m)

```
- (BOOL)application:(UIApplication *)application
      didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    ...

    // determine if app launched from a push notification
    NSDictionary *pushNotification =
        [launchOptions
         objectForKey:UIApplicationLaunchOptionsRemoteNotificationKey];
    if (pushNotification != nil) {
        NSString *action = [[[pushNotification objectForKey:@"aps"]
                                objectForKey:@"alert"]
                                objectForKey:@"action-loc-key"];

        Contact *contact = [[Model sharedModel]
                            contactWithEmailAddress:
                            [pushNotification
                             objectForKey:@"emailAddress"]];

        // initiate a phone call
        if ([action isEqualToString:@"Call"]) {
            NSString *phone = [NSString stringWithFormat:@"tel:%@",
```

```
                                contact.phoneNumber];

            [[UIApplication sharedApplication]
             openURL:[NSURL URLWithString:phone]];

            // start an email
        } else if ([action isEqualToString:@"Email"]) {
            NSString *email = [NSString stringWithFormat:@"mailto:%@",
                               contact.emailAddress];

            [[UIApplication sharedApplication]
             openURL:[NSURL URLWithString:email]];

        }
    }
    ...

}
```

If the application is active and implements `application:didReceiveRemoteNotication:`, that method is invoked with the payload from the notification. As with local notifications, much of the standard behavior is discarded if the application is active. Thus, if you want to inform your user, you must implement the functionality. Listing 10-14 demonstrates how to intercept a remote notification when the application is active and presents the user with a list of actions to perform. The bulk of this logic could be consolidated with the local notification implementation with the exception of how the action text is derived. The location within the dictionary differs depending if it is a local or remote notification.

LISTING 10-14: Remote Notification Handling When Application Is Active (/Application/Relationship Manager/RelationshipManager/AppDelegate.m)

```
- (void)application:(UIApplication*)application
      didReceiveRemoteNotification:(NSDictionary *)userInfo {

    // alert the user that a notification was received
    // because the user was in the application, we present
    // them with some additional information / options
    dispatch_async(dispatch_get_main_queue(), ^{

        NSString *action = [[[userInfo objectForKey:@"aps"]
                                objectForKey:@"alert"]
                                objectForKey:@"action-loc-key"];
        Contact *contact = [[Model sharedModel]
                            contactWithEmailAddress:
                            [userInfo objectForKey:@"emailAddress"]];

        // get the reminder message
        NSString *message;
        message = [[[userInfo objectForKey:@"aps"]
                    objectForKey:@"alert"] objectForKey:@"body"];
        if (message == nil) {
            // no message found at that path
```

continues

LISTING 10-14 *(continued)*

```
    // that implies a simple notification structure
    message = [[userInfo objectForKey:@"aps"]
                objectForKey:@"alert"];
}

[UIAlertView alertViewWithTitle:@"Reminder"
                        message:message
              cancelButtonTitle:@"Cancel"
              otherButtonTitles:[NSArray
                                 arrayWithObjects:@"View Contact",
                                 action, nil]
                      onDismiss:^(int buttonIndex)
 {
     // display the contact details
     if (buttonIndex == 0) {

         ContactDetailTableViewController *contactVC =
         [[ContactDetailTableViewController alloc]
          initWithStyle:UITableViewStyleGrouped];

         contactVC.contact = contact;
         contactVC.presentedModally = YES;

         UINavigationController *nc =
         [[UINavigationController alloc]
          initWithRootViewController:contactVC];

         [self.navigationController
          presentModalViewController:nc animated:YES];

     // initiate the selected action
     } else if (buttonIndex == 1) {
         // initiate a phone call
         if ([action isEqualToString:@"Call"]) {

             NSString *phone = [NSString
                                stringWithFormat:@"tel:%@",
                                contact.phoneNumber];

             [[UIApplication sharedApplication]
              openURL:[NSURL URLWithString:phone]];

         // start an email
         } else if ([action isEqualToString:@"Email"]) {

             NSString *email = [NSString
                                stringWithFormat:@"mailto:%@",
                                contact.emailAddress];

             [[UIApplication sharedApplication]
              openURL:[NSURL URLWithString:email]];

         }
     }
```

```
        }
                                onCancel:^()
        {
            // don't do anything for cancel
        }];    });

    // reset the application badge to 0
    [UIApplication sharedApplication].applicationIconBadgeNumber = 0;
}
```

The application now contains custom support for remote notifications and provides users with an enhanced experience if a notification is received while they use the application. Install the application on a test device and initiate a few test notifications similar to what you did earlier when confirming notifications were registered with APNs successfully. When the application is not active, you should continue to see the standard alert, similar to what you saw in Figure 10-13. However, when the application is active the user should see an alert that resembles Figure 10-14.

FIGURE 10-14

UNDERSTANDING NOTIFICATION BEST PRACTICES

Following are a few best practices to keep in mind as you implement notifications. These can enhance the overall user experience while still allowing you to achieve the engagement and efficiency gains you need. In some cases, they may help you exceed your goals.

- **Request permission only when needed:** Applications tend to inundate users during the initial launch with prompts for permission to access services such as location (GPS), remote notifications, and contacts. This can be frustrating and can result in lower conversion rates. You should request permission only to deliver remote notifications when you need to. For example, if you offer remote notifications as a paid upgrade, do not request permission to deliver remote notifications until the user initiates the upgrade process.
- **Ensure remote notification value is clear:** Users must understand the value they will receive by allowing you to deliver remote notifications. Ideally, this value proposition is understood prior to requesting permission.
- **Limit remote notification frequency:** Notifications are intended to inform users that a specific event has taken place and can now be acted on within the application, such as a friend request. Follow a delivery cadence that makes sense for your application and content. Users would expect that a messaging application deliver a notification every time they receive a message; however, those same users would likely expect only a single, daily notification from a daily-deals application.
- **Allow users to configure the notification experience:** Users want to define their own experience. Allow them to configure the types of notifications they receive (for example, friend requests and new messages), the content included in those notifications (for example, simply

displaying New Message versus the entire message content) and setting a Do Not Disturb window in which notifications will be not be delivered. In iOS 6, Apple provides a native Do Not Disturb feature. When enabled, this feature enforces a system wide Do Not Disturb window where all alerts are silenced. User configurable settings should not extend to everything, though. For example, specific bank account details and other personally identifiable information should never be delivered via a notification.

- **Support multiple devices:** Actions resulting from a notification on one device should be reflected on all other devices linked to the same user. For example, if a user has two registered devices, you push an Unread Message count of 3 to each device. As the user reads one of the messages on the first device, the unread count on each device should be updated to reflect the new unread count of 2. This requires additional resources on the server-side and initial planning within the application but makes for a much better experience overall.
- **Develop an In-App notification center:** An In-App notification center provides a centralized location for users to view important updates. Remote notifications are not reliable. When a device is not connected to a network, Wi-Fi or cellular, APNs queue only one notification per application for delivery when the devices reconnects. A notification center that retrieves sent notifications on launch provides a simple mechanism to ensure users receive your updates. This becomes more valuable for enterprise applications where users are notified of workflow items via remote notification.
- **Poll the feedback service:** When an application no longer exists on a device and a remote notification is attempted, the operating system reports the undeliverable failure to APNs. To limit unnecessary failures, Apple provides the feedback service, which is a binary interface similar to the interface to register a remote notification, available at `feedback.push.apple.com:2196` (sandbox access is available at `feedback.sandbox.apple.com:2196`). The feedback service provides a running list of per-application device tokens that had a failed delivery. As a provider, you should periodically review the feedback service and update your database accordingly.
- **Measure notification success:** Monitor how frequently notifications are sent and the number of times a notification drives an application to be launched. Consider assigning notification types or using a custom notification payload value, to enhance your understanding of which messages are most effective.

SUMMARY

Local and remote notifications provide a unique channel for engaging your users and driving efficiency in your business. Local notifications can be effective in driving users to respond to local reminders and require little overhead and management when deployed. Remote notifications offer the most flexibility for external integration but require additional configuration and maintenance compared to local notifications. Depending on your requirements, you may choose to deploy a hybrid approach to optimize cost and functionality.

In the next chapter, you learn how to communicate within applications installed on the device using techniques such as URL schemas and keychain.

PART IV
Networking App to App

11

Inter-App Communication

WHAT'S IN THIS CHAPTER?

- Registering custom URL schemes
- Changing behavior based on the presence of other installed apps
- Leveraging enterprise single sign-on
- Detecting and reusing data from previous installations

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code is in the Chapter 11 download and individually named according to the names throughout the chapter.

After mastering the traditional forms of network communication, it's natural to look at other apps on a device and wonder how you can interact with them as well. The sandboxed operating model of iOS applications limits the capabilities of the inter-app communication techniques described in this chapter, but with some creative thinking, you can accomplish more than you might think. The most direct approach is to implement one or more URL schemes in your app. URL schemes enable an app to sense the presence of other apps and perform specific actions in them. Alternatively, a more indirect approach uses a shared keychain as a common key-value store for a group of related apps by the same developer. This chapter provides concrete examples of both of these approaches. Many of these examples are meant for Facebook and Twitter, but also include a generic implementation ready to use for any other app's custom scheme.

URL SCHEMES

URL schemes have three main uses: adjusting logic based on the presence of other apps on a device, switching to another app, or responding to another app opening your app. You can also use schemes to open an app from a website or at the conclusion of a web-based authentication flow. This section uses a suite of applications to demonstrate the basic uses of custom URL schemes. Each application implements its own scheme, and the entire suite configures itself to include functionality of all other installed apps in the suite. The code examples cover best practices for implementing and responding to custom schemes, sensing other apps installed on a device, and sending serialized data between apps. Given that in most cases only one application is active at a time, data predominantly flows only one way, and the receiving app becomes active to process the data.

Implementing a Custom URL Scheme

The first step to implementing a custom URL scheme is to determine which features of your application would be useful for another application to invoke. For example, a shopping app might want to display a product's information when given a UPC code or a messaging app could prepopulate a recipient and message. It is also useful to allow other apps to link to the most popular views in your app, which in a tab bar application is commonly the root view for each tab. To do this you should assign each of the views or features a short identifier to use when responding to incoming URL requests, and create a short name to use as the URL identifier (for example, http or telnet). Ensure your short name is unique because it is undefined how the operating system will handle multiple applications that implement the same scheme.

Each application includes its URL scheme(s) in `Info.plist`, which registers them with the operating system during installation. For example, an application named Acme Employee Directory might use the scheme acme-directory in its `Info.plist`. Figure 11-1 shows the required keys and values in Xcode's plist editor.

Key	Type	Value
Localization native development region	String	en
Bundle display name	String	Directory
Executable file	String	${EXECUTABLE_NAME}
▶ Icon files	Array	(0 items)
Bundle identifier	String	com.acme.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle name	String	${PRODUCT_NAME}
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
▼ URL types	Array	(1 item)
▼ Item 0	Diction…	(2 items)
URL identifier	String	com.acme.directory
▼ URL Schemes	Array	(1 item)
Item 0	String	acme-directory
Bundle version	String	1.0
Application Category	String	
Application requires iPhone environmer	Boolean	YES
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(1 item)
Icon already includes gloss effects	Boolean	YES

FIGURE 11-1

When the operating system encounters an `acme-directory://` URL, it will call one of two methods in the application delegate, depending on the current state of the application. If the application is not running, it is launched and `application:didFinishLaunchingWithOptions:` will have an options dictionary containing the keys `UIApplicationLaunchOptionsSourceApplicationKey` and `UIApplicationLaunchOptionsURLKey`, which identify the calling application and full URL, respectively. If your application determines it can handle the given URL, it should return `YES` from `application:didFinishLaunchingWithOptions:`. The operating system then calls the application delegate's `application:openURL:sourceApplication:annotation:` to actually handle the URL. When the app is brought to the foreground, the operating system displays its default image (usually called `Default.png`). Alternatively, if the application was running in the background or suspended, it will resume and only `application:openURL:sourceApplication:annotation:` will be called to handle the URL. The two possible execution paths are shown in Figure 11-2.

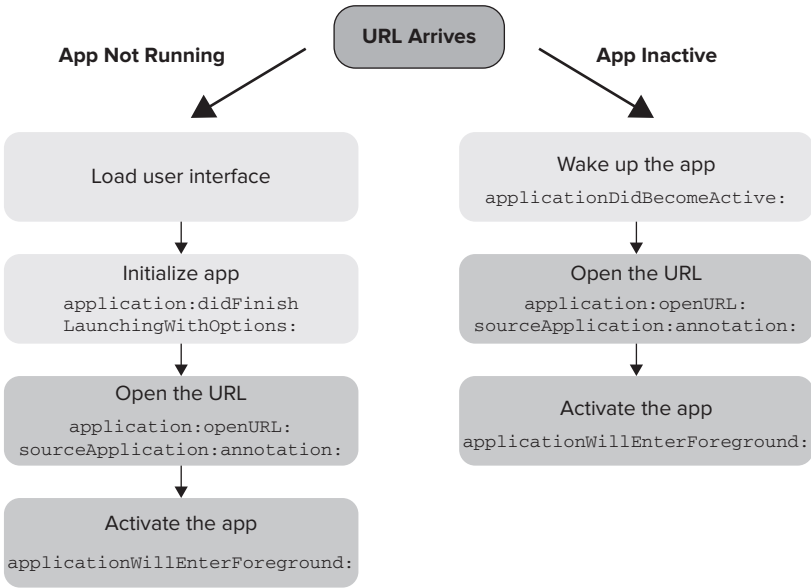


FIGURE 11-2

Because your URL scheme often displays a different view than would normally show on launch, you can define a variety of default images for each URL scheme you implement. Each variation follows a specific format with a mix of required and optional values.

- The `UILaunchImageFile` value is required and set to `Default` unless otherwise specified in `Info.plist`.
- The URL scheme string is the `acme-directory` value previously specified in `Info.plist`, and it is also required.
- An orientation value is optional, but enables you to use `Portrait` or `Landscape` to show a launch image positioned in the device's current orientation.

- Scale is an optional value that is familiar to most developers, and it enables you to use `@2x` to specify images for Retina devices.
- The final optional value is device, which displays images customized to phone-style devices with `iphone` or tablet-style devices with `ipad`.

Take the Employee Directory application as an example of the device value. It includes custom images specific to its URL scheme on phone-style devices for both non-Retina and Retina displays with the files `Default-acme-directory-portrait@2x.png` and `Default-acme-directory-portrait.png`. This snippet shows the full pattern for naming scheme-specific default images. Values in curly braces are required, and values in square brackets are optional.

```
{UILaunchImageFile value}-{URL scheme string}[-orientation][scale][~device].png
```

Determining if you can handle a URL is highly dependent on the specific application; however, for a quick sanity check you can validate that the URL is requesting one of the feature identifiers you chose previously. More specific bounds checking, pattern matching, or some other validation may also be appropriate. In the Acme Employee Directory shown in Listing 11-1, the URL is checked to ensure it uses the custom scheme `acme-directory`, requests the identifier `employee`, and supplies an employee identifier that exists.

LISTING 11-1: Validating an Acme Directory URL (EDAppDelegate.m)

```
- (BOOL)canHandleURL:(NSURL *)url
fromSourceApplication:(NSString *)sourceApplication
       withAnnotation:(id)annotation {

    NSLog(@"Determining if we can handle URL '%@' "
          "requested by '%@' with data '%@'",
          url, sourceApplication, annotation);

    // we're only requiring a URL, but we could also check
    // sourceApplication or annotation if necessary
    if (url == nil) {
        return NO;
    }

    // we'll only respond to URLs of the pattern:
    //    "acme-directory://employee/{integer}"
    if ([[url scheme] isEqualToString:@"acme-directory"]) {
        NSString *viewIdentifier = [url host];

        // there is only one view identifier we'll accept
        if ([viewIdentifier isEqualToString:@"employee"]) {
            NSInteger employeeNumber = [self employeeNumberFromURL:url];
            EDEmployee *employee = [[EDEmployeeManager sharedManager]
                                    employeeForId:employeeNumber];

            // if we got an employee, then accept the URL
            if (employee != nil) {
                return YES;
```

```
                }
            }
        }

        return NO;
    }
```

If an app responds to more than one scheme, identifier, or needs a longer path, it is easy to extend the example `canHandleURL:fromSourceApplication:withAnnotation:` method. You need to be aware of one unintuitive aspect of URL handling: Even if your application returns `NO` from `application:didFinishLaunchingWithOptions:`, it is still launched or brought to the foreground because it registered the URL scheme.

Sensing the Presence of Other Apps

Social networking features are now commonly included in a wide variety of applications to increase user engagement or spread awareness of the service or product. In many cases the developer includes Twitter and Facebook by default, and the social features work the same way for all users. However, even if users have a mobile application for either network, they are forced to use the app's unfamiliar user interface and are often required to authorize the application before proceeding with the desired feature. This added friction can hurt conversion rates for your social features and can have a material impact on the popularity of your app. The proper approach to combat this issue is to detect when a native application is installed and optionally invoke that app's custom URL scheme. Most social or messaging apps provide URL schemes for precisely this situation. Your users will appreciate using a familiar user interface in a client that has already been authorized on the service.

To start sensing the presence of another app, you first need to know the custom URL scheme(s) provided by the destination app. Many developers publish the details of any custom schemes in their API documentation or on a developer-specific page on their websites. After you have the scheme name, implementing the feature is relatively easy using `UIApplication`'s `canOpenURL:` method. It returns `YES` if an application is installed that registered the scheme in the given URL and `NO` otherwise. For example, to test for the presence of Facebook or Twitter, test if `canOpenURL:` returns `YES` for the respective URL schemes like so:

```
if ([[UIApplication sharedApplication] canOpenURL:
    [NSURL URLWithString:@"twitter://"]]) {

    // this device has the Twitter for iPhone application
}

if ([[UIApplication sharedApplication] canOpenURL:
    [NSURL URLWithString:@"fb://"]]) {

    // this device has the Facebook application
}
```

After you detect the presence of another app, you can alter your user interface or functionality to match those of that app. For example, you may want to disable the sharing feature if the app is not found, or provide a fallback that accomplishes the same task in an alternative way. Some apps register two similar URL schemes, but the second one appends the current version to the end

(for example, `acme-directory-1-0://` for version 1.0). An app that wants to use URL scheme features that are only available in certain versions of the target app can then test not only for its presence, but also for a specific version's presence.

When the same team or coordinating teams are developing many related apps, detecting installed apps becomes more interesting. In an enterprise environment, many companies deploy multiple apps internally, each with a specific focus or target audience. These apps can enable extra features or external hooks if they know applications are present that can deliver extra value.

The Employee Directory app described in the previous section can be extended to include hooks for users that have the installed Employee Records, another example app. In this scenario, Employee Records is given only to HR employees because it contains sensitive personal information. Each of the applications may want to reveal buttons to open an employee's specific page in the other apps only if they are installed.

In the directory app, users who have Employee Records can see a corresponding entry under the Companion Apps table section, as shown in Figure 11-3. This section is populated from `dictionaryOfInstalledCompanionApps` shown in Listing 11-2, which simply checks for each URL scheme. Similar code is also added to Employee Directory and Employee Records to enable the same functionality.

LISTING 11-2: Checking for Companion Applications (EDUtils.m)

```
+ (NSDictionary*)dictionaryOfInstalledCompanionApps {
    NSMutableDictionary *companionApps = [NSMutableDictionary dictionary];

    // employee records
    if ([[UIApplication sharedApplication] canOpenURL:
        [NSURL URLWithString:@"acme-records://"]]) {

         [companionApps setValue:@"acme-records://employee/"
                          forKey:@"Employee Records"];
    }

    return companionApps;
}
```

FIGURE 11-3

Advanced Communication

The previous examples have used only plaintext to convey information to the destination app, but you can send almost any serializable object in a URL. The Employee Records application example can be further extended to receive an image from another application and add it to an employee's file. Any object that can be serialized to `NSData` can be sent via a custom URL scheme. For most common iOS types and all the primitive types, you can use methods provided by `NSKeyedArchiver` to create encapsulated data objects. Objects are serialized with `archivedDataWithRootObject:` and primitive types with one of many

methods that follow the pattern `encode{type}:forKey:`. `NSDictionary` objects and `UIImage` objects have custom serializers specific to them. To serialize your own custom object, it must implement `NSCoding` and two methods: `initWithCoder:` and `encodeWithCoder:`. `NSKeyedArchiver` can use the two methods automatically when your object is passed to `archivedDataWithRootObject:`. Table 11-1 maps common iOS object types to their respective serializers.

TABLE 11-1: Serializers for Common iOS Types

TYPE	SERIALIZER
`NSDictionary`	`NSPropertyListSerialization`
`UIImage`	`UIImageJPEGRepresentation()` or `UIImagePNGRepresentation()`
Objects conforming to `NSCoding`	`NSKeyedArchiver`
Primitive types	`NSKeyedArchiver`

After the object has been represented as an `NSData` object, it must be converted to a string to pass through a URL. An obvious answer is to use `NSString`'s `initWithData:encoding:`; however, the resulting string would not be safe to include in a URL. RFC 3986 defines a list of valid characters that may be included in a URL, summarized in Table 11-2.

TABLE 11-2: Valid URL Characters

Character	Name
A–Z	Uppercase letters
a-z	Lowercase letters
0–9	Numbers
-	Hyphen
.	Period
_	Underscore
~	Tilde
:	Colon
/	Forward slash
?	Question mark

continues

TABLE 11-2 *(continued)*

Character	Name
#	Number sign or hash
] or [	Left or right square brackets
@	At sign
!	Exclamation mark
$	Dollar sign
&	Ampersand
‘	Single quote
) or (	Left or right parentheses
*	Asterisk
+	Plus sign
,	Comma
;	Semicolon
=	Equals sign

RFC 3986, `http://tools.ietf.org/html/rfc3986`

Knowing this set of characters, the Internet Engineering Task Force standardized an encoding known as `base64` in RFC 4648. It is used specifically to represent binary data (the image in this example application) as an ASCII text string. Every six bits of the binary data is encoded into one character using a standardized conversion table, as shown in Table 11-3.

TABLE 11-3: base64 Conversion Table

6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1

6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

RFC 4648, `http://tools.ietf.org/html/rfc4648`

Although the `base64`-encoded string is close, it is still not quite ready for inclusion in the URL. Characters 62 ("+" or plus) and 63 ("/" or forward slash) each have a special meaning in a URL and could potentially disrupt the receiving app from decoding your binary data correctly. The plus sign represents a space character when used in the query string, and if your data is interpreted as a space, the parser could stop prematurely or interpret erroneous data. The forward slash character is used to represent components of the URL's path and could confuse the parser into starting or stopping too early. For example, consider a case in which the start of the binary data encodes to `employee/5/`. When passed to Employee Records the URL could be parsed as `acme-records://employee/5/...`, which would open the fifth employee's detail view instead of passing the full binary data as intended.

There are two solutions to the unintended consequences of the `+` and `/` characters in a `base64`-encoded string. The most straightforward fix is to URL-encode the `base64` string. Many applications that use network communication already have a utility method to URL-encode a string, and it can be reused here. If your application does not have one, be wary of solutions that use only `NSString`'s `stringByAddingPercentEscapesUsingEncoding:`, which Apple describes as returning:

> *... a representation of the receiver using a given encoding to determine the percent escapes necessary to convert the receiver into a legal URL string.*

Recall that the plus sign and forward slash *are* legal URL characters; thus, they will not be encoded by it. When given a `base64`-encoded string, `stringByAddingPercentEscapesUsingEncoding:` actually returns the exact same string unchanged.

The second solution is to use a `base64` variant called `base64url` that was created to address this situation. It uses a modified conversion table that replaces the plus sign with a hyphen (-) and the forward slash with an underscore (_). The modified conversion table is shown in Table 11-4.

TABLE 11-4: base64url Conversion Table

6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER	6-BIT VALUE	ENCODED CHARACTER
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	–
15	P	31	f	47	v	63	_

RFC 4648, `http://tools.ietf.org/html/rfc4648`

A fourth sample application Employee Records Image Adder adds an image to an employee using a second identifier in the Employee Records URL scheme. For example, to add an image to the first employee, the app should open `acme-records://addimage/1/{image data string}`. To create the image data string, it employs the three-step process discussed previously: Serialize the image, `base64`-encode it, and then URL-encode it. The code in Listing 11-3 uses a `base64`-encoding category on `NSData` that provides `base64EncodingWithLineLength:`; however any `base64` implementation will do. The receiving application performs the same three steps in reverse order to recover the `UIImage` object.

LISTING 11-3: Passing an Image in a Custom URL Scheme (IAViewController.m)

```
// encode the image as data
NSData *imageData = UIImagePNGRepresentation(imageView.image);

// turn the data into a string by base64-encoding it
NSString *imageString = [imageData base64EncodingWithLineLength:0];

// url-encode the base64 string
NSString *encodedString = [IAUtils encodeURL:imageString];

// create and open the URL
NSURL *url = [NSURL URLWithString:[NSString stringWithFormat:
                @"acme-records://addimage/%@/?%@",
                employeeNumberField.text,encodedString]];
[[UIApplication sharedApplication] openURL:url];
```

Although URL schemes are great when you intend to open the receiving app and any size data can be passed this way, the user experience will suffer as the time required to validate and open a long URL becomes noticeable. Alternative techniques exist that offer the ability to share data while maintaining focus on your application.

SHARED KEYCHAINS

A shared keychain is especially useful in the enterprise because it creates a common area accessible to all apps that share a bundle seed ID. This shared space makes it straightforward to implement a single sign-on (SSO) authentication system for a group of related apps. Additionally, an app that stores data in the keychain can detect previous installs of itself, which can improve user experience by reusing previously provided authentication credentials or adjusting the user interface for an experienced user.

The iOS keychain provides a single area for secure storage for protected operating system data like Wi-Fi passwords or account credentials. That storage is also available for third-party apps to store similar protected data. To remain secure, keychain entries are always encrypted on disk and in device backups. Protected data remains in the keychain even after an app is deleted, which enables later installations to reuse the same credentials. Multiple apps from the same developer can be configured to use the same encryption key, which allows each of those apps to access shared keychain items. The code examples in this section offer a template for advanced features, such as implementing SSO for a suite of apps or detecting previous installations of an app.

Enterprise SSO

Single sign-on functionality is a requirement for most enterprise applications delivered on the desktop or over the web because it enhances security and convenience for the user. As the number of internally deployed apps grows, SSO will become just as important in the mobile environment as well. SSO is commonly implemented as an end-to-end authentication framework that enables users to authenticate to multiple applications with a single set of shared credentials. When a successful

login attempt has been made, the SSO provider usually issues an authentication token that is stored and used to sign all subsequent requests. If an application's security requirements allow for this token to be shared, it can be securely stored in the shared keychain and made available to other apps without requiring the user to log in to those apps as well. If the token cannot be shared, the user's account identifier or e-mail address can still be saved, providing a smaller but still significant productivity gain.

Three preliminary steps must be taken before a project can use a shared keychain for SSO:

1. Each app must share a common Bundle Seed ID, which is a value set in the iOS Provisioning Portal when applications are created. Figure 11-4 shows an example application's entry in the Provisioning Portal.

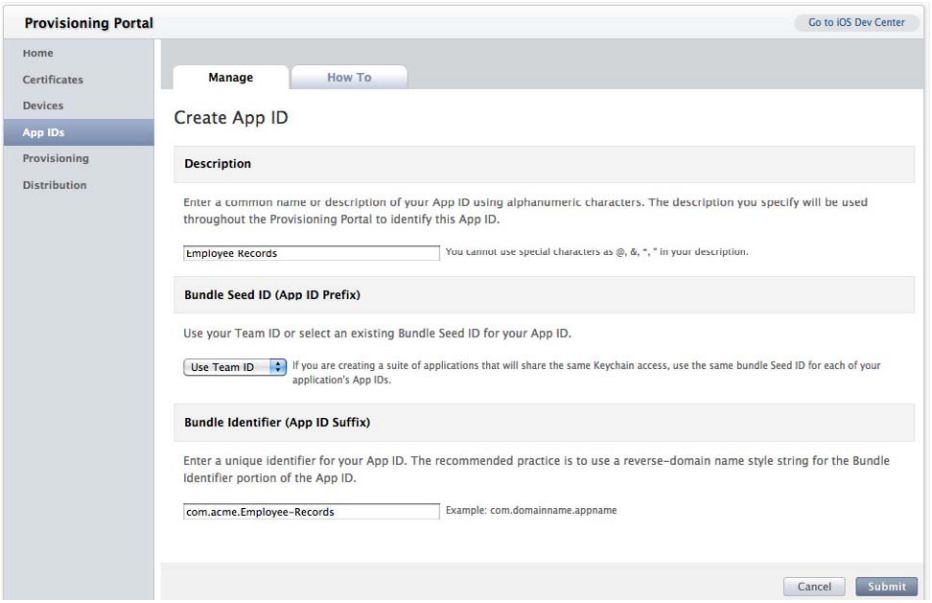


FIGURE 11-4

2. Each project needs to include an `Entitlements.plist` file that specifies the Bundle Seed ID in one or more `keychain-access-groups`. Figure 11-5 demonstrates the required keys and values in Xcode's plist editor.

Key	Type	Value
▼ keychain-access-groups	Array	(2 items)
Item 0	String	$(AppIdentifierPrefix)com.acme.Employee-Records
Item 1	String	$(AppIdentifierPrefix)com.acme.sso

FIGURE 11-5

3. Each project needs to include the `Security.framework` to properly compile. To include this, perform the following steps:
   - **A.** In the Project Navigator, highlight your project.
   - **B.** Highlight your target.

C. Select the Build Phases tab.

D. Expand Link Binaries with Libraries.

E. Click the + button.

F. Search for and add `Security.framework`.

Now that the project is configured, it is prudent to begin by developing some low-level keychain utility methods that can help drive high-level functionality of the application. Because this utility class will be included in each individual SSO-enabled app, its implementation should be decoupled from implementation details of any one application. Ideally, it should be compiled into a static library that can be included in each application with little or no configuration needed.

Most keychain Create, Read, Update, and Delete (CRUD) operations use a common set of configuration parameters that are given in an `NSDictionary` object that contains a specific set of keys. Listing 11-4 contains a setup method, `keychainSearch:`, that creates the dictionary object with the shared configuration. Each operation calls `keychainSearch:` at some point to initialize the keychain search context. Your utility method needs to include many keys, each of which is described in more detail in Chapter 7, "Optimizing Request Performance." The most important key is `kSecAttrAccessGroup`, which defines the shared keychain space used by the SSO framework. Note that the iOS Simulator's keychain implementation does not support `kSecAttrAccessGroup`, and it must be included only in compilations for an iOS device. Because of this quirk, all apps have access to all keychain items on the simulator.

LISTING 11-4: Initializing a Keychain Search Dictionary (SSOUtils.m)

```
+ (NSMutableDictionary*)keychainSearch:(NSString*)identifier {
    NSData *encodedIdentifier = [identifier dataUsingEncoding:
                                 NSUTF8StringEncoding];
    NSMutableDictionary *keychainSearch = [[[NSMutableDictionary alloc]
                                          init] autorelease];

    // set the type to generic password
    [keychainSearch setObject:(id)kSecClassGenericPassword forKey:(id)kSecClass];

    // set the item's identifier
    [keychainSearch setObject:encodedIdentifier forKey:(id)kSecAttrGeneric];
    [keychainSearch setObject:encodedIdentifier forKey:(id)kSecAttrAccount];

    // use the shared keychain
    // note: not supported in the simulator and will cause
    // all keychain calls to fail
    #if !(TARGET_IPHONE_SIMULATOR)
        [keychainSearch setObject:kKeychainSSOGroup
                           forKey:(id)kSecAttrAccessGroup];
    #endif

    return keychainSearch;
}
```

To read an existing keychain value, Listing 11-5 implements `getValueForIdentifier:`, which returns a keychain value. It is given an identifier that acts as a key for the desired value. It calls `keychainSearch:` to initialize the configuration dictionary and then sets two additional parameters.

LISTING 11-5: Retrieving a Keychain Item (SSOUtils.m)

```
+ (NSString*)getValueForIdentifier:(NSString*)identifier {
    NSMutableDictionary *search = [self keychainSearch:identifier];

    // limit to the first result
     [search setObject:(id)kSecMatchLimitOne forKey:(id)kSecMatchLimit];

    // return data vs a dictionary of attributes
     [search setObject:(id)kCFBooleanTrue forKey:(id)kSecReturnData];

    // perform the search
    NSData *value = nil;
    OSStatus status = SecItemCopyMatching((CFDictionaryRef)search,
                                          (CFTypeRef *)&value);

    if (status == noErr) {
        return [NSString stringWithUTF8String:[value bytes]];
    }

    return nil;
}
```

Listing 11-5 specifies a value of `kSecMatchLimitOne` for `kSecMatchLimit` to ensure the keychain returns the first result it finds. To return more than one result, you can pass a `CFNumberRef` specifying the desired maximum number of results or pass `kSecMatchLimitAll` to return all possible results. The method also sets `kSecReturnData` to `kCFBooleanTrue` to instruct the keychain to return the raw data of the wanted item's value. Other types of return data are possible by setting other keys to `kCFBooleanTrue`:

- `kSecReturnAttributes` returns an attribute of the item.
- `kSecReturnRef` returns a reference to the item.
- `kSecReturnPersistentRef` returns a persistent reference to the item.

If more than one of these types is true, then the keychain request returns a dictionary of the requested information. The query itself is handled by the call to `SecItemCopyMatching()`, which takes your configuration parameters and executes the search. The results of the search are copied into the reference provided by `&value`, and the returned `OSStatus` is the status of the search. Possible status codes are summarized in Table 11-5. The utility method ends by returning the string value of the returned value or `nil` in the case of any error.

TABLE 11-5: Keychain Search Status Codes

RETURN CODE CONSTANT	DESCRIPTION
errSecSuccess	No error.
errSecUnimplemented	Function or operation not implemented.
errSecParam	One or more parameters passed to the function were not valid.
errSecAllocate	Failed to allocate memory.
errSecNotAvailable	No trust results are available.
errSecAuthFailed	Authorization/authentication failed.
errSecDuplicateItem	The item already exists.
errSecItemNotFound	The item cannot be found.
errSecInteractionNotAllowed	Interaction with the security server is not allowed.
errSecDecode	Unable to decode the provided data.

The next utility method `setValue:forIdentifier:` is shown in Listing 11-6 and implements both `create` and `update` operations on keychain items. Separate keychain methods exist for the two operations, and your implementation must determine which is appropriate. Thus, it calls `getValueForIdentifier:` to determine if an item with this identifier already exists. If it does exist, the method branches into the `update` logic. If the new value differs from the existing value, it will be converted to an `NSData` object and then inserted in the `update` parameters `NSDictionary` with the key `kSecValueData`. There are many other attributes that can be included in this dictionary, and the full list is available in Apple's Keychain Constants documentation. If an item for the given identifier does not already exist, the method executes the `else` branch, which creates the item. Similarly to `update`, it sets the item's `NSData` representation to the key `kSecValueData`; however, the key/value pair is inserted into the search dictionary instead of an entirely new dictionary. The method passes `NULL` to the result parameter of `SecItemUpdate()` because it does not need to retain a reference to the newly inserted item. Both branches of code return `YES` if the operation is successful and `NO` otherwise.

LISTING 11-6: CREATING OR UPDATING A KEYCHAIN ITEM (SSOUTILS.M)

```
+ (BOOL)setValue:(NSString*)value forIdentifier:(NSString*)identifier {
    NSString *existingValue = [self getValueForIdentifier:identifier];

    // check if value exists
    if (existingValue) {

        // update if the new value is different
        if (![existingValue isEqualToString:value]) {
```

continues

LISTING 11-6 *(continued)*

```
            NSMutableDictionary *search = [self keychainSearch:identifier];

            NSData *valueData = [value dataUsingEncoding:NSUTF8StringEncoding];
            NSMutableDictionary *update = [NSMutableDictionary
                dictionaryWithObjectsAndKeys:valueData,
                (id)kSecValueData, nil];

            OSStatus status = SecItemUpdate((CFDictionaryRef)search,
                                            (CFDictionaryRef)update);

            if (status == errSecSuccess) {
                return YES;
            }

            return NO;

        } else {
            return YES;
        }

    // if no value exists, create a new entry
    } else {
        NSMutableDictionary *add = [self keychainSearch:identifier];

        NSData *valueData = [value dataUsingEncoding:NSUTF8StringEncoding];
        [add setObject:valueData forKey:(id)kSecValueData];

        OSStatus status = SecItemAdd((CFDictionaryRef)add,NULL);

        if (status == errSecSuccess) {
            return YES;
        }

        return NO;
    }
}
```

The last CRUD operation, Delete, is shown in Listing 11-7 and is simple to implement using the existing `keychainSearch:` method. The keychain method `SecItemDelete()` can search for all items matching the search dictionary and remove them from the keychain. If the Delete operation is successful, the utility method returns `YES`; otherwise it returns `NO`.

LISTING 11-7: Deleting a Keychain Item (SSOUtils.m)

```
+ (BOOL)deleteValueForIdentifier:(NSString*)identifier {
    NSMutableDictionary *search = [self keychainSearch:identifier];

    OSStatus status = SecItemDelete((CFDictionaryRef)search);

    if (status == errSecSuccess) {
```

```
        return YES;
    }

    return NO;
}
```

Now that the app has some building blocks to use for keychain interaction, high-level functionality for the SSO implementation can be implemented with them. The most important is `authenticateWithUsername:andPassword:` that can perform an authentication check and save the resulting token into the keychain. The example provided in Listing 11-8 can accept any username/password combination; however a real enterprise implementation always requires a network call to an SSO provider. That provider should return a cryptographically secure authentication token that expires after a specified time interval. The app should store the token for use when making subsequent network requests, and it can optionally store the expiration time if available.

LISTING 11-8: Authenticating with a Username and Password (SSOUtils.m)

```
+ (BOOL)authenticateWithUsername:(NSString *)username
                     andPassword:(NSString *)password {

    // you should do a check of the given credentials here
    // this dummy application will always return a successful login
    BOOL loginResult = YES;

    if (loginResult == YES) {
        // set SSO username
         [self setValue:username forIdentifier:kCredentialUsernameKey];

        // set SSO token
        [self setValue:@"SSOValidToken" forIdentifier:kCredentialTokenKey];
    }

    return loginResult;
}
```

Later views within the secured application should call `credentialsAreValid` before requesting or displaying sensitive data, typically in a view controller's `viewWillAppear:` method or in the app delegate's `applicationWillEnterForeground:`. The code in Listing 11-9 validates the saved authentication token and informs the application whether it can continue to display the view, or if it should request that the user re-authenticate. This example implementation does a dummy check, but a rigorous check may include a separate network call to an SSO provider or a client-side cryptographic check.

LISTING 11-9: Validating Stored Credentials (SSOUtils.m)

```
+ (BOOL)credentialsAreValid {
    NSString *token = [self credentialToken];

    if (token == nil) {
        return NO;
```

continues

LISTING 11-9 *(continued)*

```
    }

    // you should do a secure check of the token here
    // we'll do a dummy check to make sure it matches our
    // secret value 'SSOValidToken'
    return [token isEqualToString:@"SSOValidToken"];
}
```

The last high-level operation logs out the current user by deleting the stored authentication token. Listing 11-10 shows an example implementation of `logout`. Your implementation could also delete `kCredentialUsernameKey`; however, in most cases the user experience benefit of prepopulating a username field with the previous value outweighs any potential security weakness.

LISTING 11-10: Removing Stored Credentials for Logout (SSOUtils.m)

```
+ (void)logout {
    // destroy the saved token
    [self deleteValueForIdentifier:kCredentialTokenKey];
}
```

With this foundation, an enterprise application has everything it needs to implement an SSO system that improves user experience and efficiency across a group of related apps. As mobile apps become more prevalent in the enterprise, experience with shared keychain programming will become a requirement for every iOS developer.

Detecting Previous Installations

Using a combination of persistent and temporary storage, an app can detect previous installations of itself and initialize its user interface to help the returning user. The most common use for this capability is to remember previous authentication values and prepopulate parts of the app with the existing username or password. This functionality is nearly built-in for apps that employ the SSO pattern described previously. The installed app can also skip introductory tutorials or feature tours to let an experienced user get up-and-running as soon as possible.

This section describes an example app that records the user's birthday to the keychain and loads it on subsequent launches. If the app is deleted and reinstalled, it asks the user if he would like to use the previously saved birthday or discard it. Listing 11-11 demonstrates a view controller's `viewDidLoad` method that loads the saved birthday (if any) and prepopulates the text field if it isn't the first launch. Although this method also checks if this is the first launch of the newly installed app, it is not required and is done only to avoid filling in the birthday field before the user responds to the `UIAlertView`.

LISTING 11-11: Changing the UI Based on a Previous Installation (IDViewController.m)

```
- (void)viewDidLoad {
    // check for a previous installation
    NSString *savedBirthday = [IDUtils savedBirthday];
    BOOL firstLaunch = [IDUtils isFirstLaunch];

    if (savedBirthday != nil && firstLaunch) {
         [[[UIAlertView alloc] initWithTitle:NSLocalizedString(
                @"Previously Saved Birthday",@"Previously Saved Birthday")
            message:NSLocalizedString(@"It appears you saved a birthday
                in a previous installation of this application.  Do you
                 want to keep it?",@"It appears you saved a birthday
                in a previous installation of this application.  Do you
                 want to keep it?")
            delegate:self
            cancelButtonTitle:NSLocalizedString(@"Discard", @"Discard")
            otherButtonTitles:NSLocalizedString(@"Keep", @"Keep"),nil]
         show];
    }

    // pre-populate the birthday field if this isn't the first launch
    if (firstLaunch == NO) {
    self.birthdayField.text = savedBirthday;
    }

    // focus the birthday field
    [self.birthdayField becomeFirstResponder];
}
```

The `savedBirthday` method in Listing 11-12 simply fetches the value in the keychain for a predetermined key `kKeychainBirthdayKey`. If no value is found, you can safely assume the app was never previously installed. Reliably testing this situation is difficult because you get only one chance per test device. However, using `deleteValueForIdentifier:` to delete the keychain entry for `kKeychainBirthdayKey` puts the device in the pre-installation state. For easy back-to-back testing, call the delete utility method in the app delegate's `applicationDidEnterBackground:` and make sure to background the application before stopping the task in Xcode.

LISTING 11-12: Retrieving a Saved Birthday (IDUtils.m)

```
+ (NSString *)savedBirthday {
    return [self getValueForIdentifier:kKeychainBirthdayKey];
}
```

You should recognize `getValueForIdentifier:` from the discussion of the SSO pattern, and the exact same code is reused here. The `NSUserDefaults` storage is used to implement `isFirstLaunch:` in Listing 11-13.

LISTING 11-13: Detecting the First Launch of an Application (IDUtils.m)

```
+ (BOOL)isFirstLaunch {
    BOOL hasBeenLaunched = [[NSUserDefaults standardUserDefaults]
                            boolForKey:kDefaultsHasBeenLaunchedKey];

    if (hasBeenLaunched == NO) {
        // this is the first launch, so set a defaults value
        // saying that we were launched at least once
        [[NSUserDefaults standardUserDefaults] setBool:YES
              forKey:kDefaultsHasBeenLaunchedKey];
        [[NSUserDefaults standardUserDefaults] synchronize];

        return YES;
    }

    return NO;
}
```

This method returns `YES` when the app is launched for the first time under its current installation. Unlike the keychain, any `NSUserDefaults` entries are cleared when the app is uninstalled; therefore if `kDefaultsHasBeenLaunchedKey` is not found (which `boolForKey:` returns as `NO`) you know it must have been cleared. After it is called for the first time, this method writes `YES` to `kDefaultsHasBeenLaunchedKey`.

Although trivial, the birthday example application can get you started on the mechanics to modify the UI for previous users of an app. The tricky issue is determining how to streamline features for those users while retaining a clean user experience for brand new users. You certainly want to avoid altering the app too drastically or the code base can become completely unmanageable.

SUMMARY

This chapter covered four different ways to use inter-app communication to add functionality or enhance the user's experience of an iOS application. Detecting other apps' custom URL schemes gives a developer the ability to enable additional functionality when other apps are installed or link to specific views in installed third-party apps. Reading and writing to a shared keychain is an indirect way to communicate between apps from the same developer or company. The shared keychain provides a common key-value store for those apps to store SSO or other persistent data. Inter-app communication is not always the most elegant or direct way to send data, but in the compartmentalized world of iOS apps, it can offer functionality that is otherwise impossible.

12

Device-to-Device Communication with Game Kit

WHAT'S IN THIS CHAPTER?

- Using Game Kit classes and configuring transfer options
- Understanding traditional client-server communication
- creating peer-to-peer connections

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code for this chapter is found in the Chapter 12 download and is all from one example project: `Game Kit Auctioneer.zip`.

All communication topics covered so far have assumed that the device is connected to a network that is connected to the Internet at large; however, iOS devices can transfer data even in the absence of a traditional network. Apple's Game Kit framework facilitates device-to-device communication in environments lacking cellular service, access to power to run a Wi-Fi infrastructure, or some other limitation that precludes offering access to a local area network (LAN) or the Internet. Examples might be deep in a national forest, on a remote stretch of highway, or far underground in a building's subbasement.

Although the name implies its most common use, Game Kit is not only about enabling multiplayer games. The framework is data-agnostic, and apps can send any type of data using a variety of communication options. Its unique capability to operate over both short-range

personal area networks (PANs) without any networking infrastructure and on more conventional Wi-Fi LANs makes Game Kit an essential tool for a well-rounded iOS developer. This chapter discusses initializing a networking session, discovering other devices, and sending data packets in the context of a Game Kit-powered auction client.

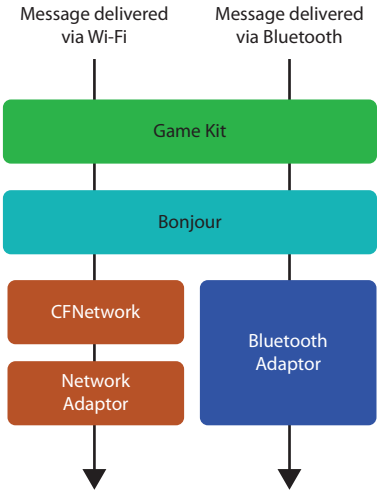


FIGURE 12-1

GAME KIT BASICS

Game Kit encompasses game-specific technologies such as achievements, leaderboards, and matchmaking in addition to the underlying network communication features. Figure 12-1 gives an overview of Game Kit's position on top of the other high-level communication frameworks.

> **NOTE** *This chapter does not discuss the gaming features because they are used only with Apple's Game Center service and are not applicable for general purpose communication.*

Game Kit networking gives developers three communication modes that control the flow of data messages among the devices in a network. These networks are represented on each device as a `GKSession` instance, and each connected device, or *peer*, is identified by its peer ID.

- **Peer-to-peer (P2P) —** This mode treats each peer equally and sends all messages to everyone connected to the network.
- **Client-server —** In this communication mode one peer is designated the host of the session, and all other peers are only clients of that host. The data transmission properties of the network remain unchanged; however, in a client-server network peers can't see other peers in the same role.
- **Turn-based match —** This final mode is less favorable for enterprise software because each participant must have a Game Center account. Because of this significant drawback, this chapter does not discuss turn-based communication.

Within these high-level session configurations, there are also developer-configurable settings to control the availability and behavior of the underlying networking stack. Session availability is controlled differently based on the wanted user interface. `GKPeerPickerController`, Apple's provided implementation, enables you to specify if you want to search for peers over Bluetooth and

Wi-Fi using the `connectionTypesMask`. The following code example demonstrates the three connection configurations. In addition to enabling you to search for peers of a certain type, Apple's implementation can prompt the user to turn on Bluetooth if it is not enabled. Because the Bluetooth setting is not something exposed as a public API, the same functionality is impossible to implement with a custom user interface. If either connection type is required for the application to function, you should indicate that requirement in the `UIRequiredDeviceCapabilities` dictionary of the application's `Info.plist`. To require the ability to make Wi-Fi connections, set the key `wifi` with the value `YES`. To require Bluetooth connections on iOS 3.1 or later, set the key `peer-peer` to `YES`. If your application uses a custom UI instead of `GKPeerPickerController`, `GKSession` always responds to both Bluetooth and Wi-Fi peers.

```
GKPeerPickerController *picker = [[GKPeerPickerController alloc] init];

// search for only Bluetooth peers
picker.connectionTypesMask = GKPeerPickerConnectionTypeNearby;

// search for only Wi-Fi peers
picker.connectionTypesMask = GKPeerPickerConnectionTypeOnline;

// search for Bluetooth or Wi-Fi peers
picker.connectionTypesMask = (GKPeerPickerConnectionTypeNearby |
                              GKPeerPickerConnectionTypeOnline);

[picker show];
```

Although Bluetooth has the unique advantage of operating independently of network infrastructure, it also carries some significant drawbacks. Its maximum range of 32 feet is considerably shorter than that of a Wi-Fi network, which can span multiple access points to cover very large areas. Wi-Fi networks also have nearly ten times more bandwidth available to each peer. Although a Bluetooth radio uses less power than a Wi-Fi radio, if all peers are available over both connection types, Game Kit still prefers Wi-Fi to Bluetooth. While this may seem like a mistake when considering just one connection, it actually reduces the device's overall power consumption. An iOS device connected to a Wi-Fi network will use that connection for all background data requests such as periodic e-mail checks or push notifications. Because that radio will already be in use, it actually increases power consumption to power the Bluetooth radio at the same time.

Even though Apple identifies that its "nearby" implementation is based on Bluetooth, apps cannot directly control the Bluetooth interface without joining the Made for iPhone (MFi) program. MFi is designed for interfacing with external accessories such as speaker docks, medical sensors, and other specialized hardware. Similarly, Apple's Wi-Fi communication is implemented with Bonjour, but applications cannot interact with the Bonjour service directly. For more information about Bonjour, see Chapter 13, "Ad-Hoc Networking with Bonjour."

Message reliability is the last important configuration decision to make before hooking up the Game Kit classes and delegates. The framework provides two reliability settings, `GKSendDataReliable` and `GKSendDataUnreliable`, when sending individual datagrams. A *datagram* is a message sent over the network that can be composed of one or more packets. If the message size is larger than

1000 bytes, the amount of data that fits in one packet, it is split into chunks, transmitted individually to the receiving party, and then stitched back together into the original message. Because the message chunks are sent separately, the receiver must wait for all chunks to arrive before processing the final message, which significantly degrades performance. Game Kit enforces a maximum size of 87 kilobytes for a single datagram. Sending a datagram with the `GKSendDataReliable` mode ensures that it will be retransmitted if a network error occurs, and it guarantees that messages will be delivered in the order that they were sent. With `GKSendDataUnreliable` however, the datagram is sent only once, and if it never reaches the intended destination, then it is lost forever. Readers who are familiar with transport layer protocols will notice that these two reliability modes almost exactly match the operating characteristics of Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) datagrams, and Game Kit does use TCP and UDP under the covers to enable the guarantees of `GKSendDataReliable` and `GKSendDataUnreliable`. Examples of both reliability modes are shown in the following code.

```
NSError *error;
GKSession *session;
NSMutableData *stateUpdatePacket;
NSMutableData *heartbeatPacket;

// initialization code omitted for brevity

// send this state update packet reliably
if (![session sendDataToAllPeers:stateUpdatePacket
                    withDataMode:GKSendDataReliable
                           error:&error]) {

    NSLog(@"Error sending packet: %@",[error localizedDescription]);
}

// send this heartbeat packet unreliably
if (![session sendDataToAllPeers:heartbeatPacket
                    withDataMode:GKSendDataUnreliable
                           error:&error]) {

    NSLog(@"Error sending packet: %@",[error localizedDescription]);
}
```

The reliability settings are not per session but per datagram, which lets apps choose the most appropriate reliability setting for the type of data being sent. For example, initialization or state update datagrams are essential to the operation of an app and should always be sent reliably. You want to guarantee their order because peers who receive those messages out of order will potentially be in different states of operation. Simple heartbeat datagrams or frequent UI updates, which may be sent hundreds of times per session, will typically be sent unreliably because by the time the sender discovers that the datagram was lost in transit, it is most likely time to send the next one. Thus, using `GKSendDataReliable` might actually decrease performance of the app by using resources to send stale updates and queue the more recent datagrams, causing a staleness feedback loop.

All communication over a Game Kit session is sent unencrypted. If datagram confidentiality is essential to the application, for example, if a mobile checkout device transmits credit card information back to a register, the developer is responsible for encrypting the traffic before handing it off to the `GKSession`. See Chapter 6, "Securing Network Traffic," for more information on encryption for iOS devices. If a business case does not require the security of encryption, all values sent via Game Kit should be checked for sanity to protect against tampering by a malicious user. For example, a mobile checkout device shouldn't allow for purchase transactions with a negative price or for items that don't exist in a retailer's database.

When debugging Game Kit applications, it's important to use real devices whenever possible. The iOS Simulator cannot connect to Bluetooth sessions and behaves differently than a physical device when communicating over Wi-Fi. Even though this requires that you have two or more devices at a time, Xcode supports running the same app on multiple devices simultaneously and can easily switch between log consoles or debuggers on each one. This greatly helps debugging because you can log state changes or packets to the console and compare what each device sees at all times.

> **NOTE** *When two devices connect via USB, both appear in the Scheme/Device drop-down along with any installed Simulators. To debug an app on both devices, simply select one device and run the app; then select the second device and run the app again. To switch between consoles or debuggers on each device, choose between the devices in the Debug Area's top toolbar drop-down.*

PEER-TO-PEER NETWORKING

Peer-to-peer Game Kit connections allow any peer in the network to behave as both a server and client simultaneously. In many situations the same device switches between client and server roles as it works through a business process, but a single P2P connection can be used instead of having the device reconnect under each new role. P2P is also well suited for a process where the peer's role is not known before the process begins.

The ability to be both a client and a server is powerful, but it also has its drawbacks. P2P can make your code more convoluted as you keep track of your status in the network in coordination with all other peers. Additionally, although Apple doesn't explicitly set a maximum number of P2P connections that Game Kit supports, the reliability of the network drops slightly as each peer joins the session, and it drops precipitously after the fifth peer. Symptoms of decreased reliability include peers disappearing from the session without cause or datagrams not successfully arriving at their intended destination. Thus it is recommended that you connect only between two and four peers for any one session. Game Kit implicitly steers you in this direction because `GKPeerPickerController` supports only connecting one peer to another without offering support for three or more peers. These red flags are something to keep in mind when exploring the functionality of P2P.

To demonstrate a P2P networking, this section provides an example application called Game Kit Auctioneer. It allows peers to create an auction for an item, accept multiple bids from other peers, and end bidding at any time. Peers who want to buy instead of sell can simply wait for an auction to appear, join it, and then make a bid if the price is favorable. In the age of eBay, an auction doesn't seem like a good fit for Game Kit; however, consider the case of a farmer's market in a remote location or a surplus warehouse with thick steel walls. In these places an Internet connection might not exist or reception may be too poor for it to be useful. With a more traditional web services-based auction app, it becomes completely useless when the Internet is no longer available. Game Kit's capability to create a Bluetooth PAN means that it can work both in remote locations and wherever Wi-Fi is available.

Connecting to a Session

To create or connect to an existing Game Kit session, each peer creates a `GKSession` object, which interacts with the network on the peer's behalf. It takes three parameters: the session ID, the peer's display name, and the desired networking mode for the peer. For this P2P application, the networking mode is always `GKSessionModePeer`. Listing 12-1 shows the complete session creation code.

LISTING 12-1: Creating a GKSession (GANetworkingManager.m)

```
#define kGameKitSessionID @"auctioneer1.0"

// create a new GameKit session
_session = [[GKSession alloc] initWithSessionID:kGameKitSessionID
                                    displayName:nil
                                    sessionMode:GKSessionModePeer];
_session.delegate = self;
```

The session ID is the first parameter and is constant across all instances of the application and ensures that you connect only to peers who understand your data messages. Only peers advertising the same session ID will be visible to others as available. It is essential that your session ID contains the current version of the application to allow newer versions to modify or add new message types. You want to ensure that mismatched versions never communicate; consider what would happen if the next version of the Auctioneer app changed the order of how winning bids are identified. Version 2.0 users might see that the winner was named $12.00 instead of Johnny Appleseed. More likely is that the data formats won't be compatible in any way, and both clients will ignore each other's malformed messages or even crash.

The second parameter, display name, is a human-readable name for the peer that is returned by `displayNameForPeer:`. If a display name of `nil` is given when creating the session, the device's name, as shown in the Settings app, is used. Each peer also has a machine-readable `peerId`, which is used to uniquely identify each peer in the session. Display name is not guaranteed to be unique and should be used only for display purposes in the UI.

Now that the app has a session to use, the user is presented with the `GALobbyViewController`, which shows all available peers in a `UITableView` (refer to Figure 12-2). If a user wants to host an auction, he taps on the other peers to invite and waits for their response. To join someone else's auction, he simply has to wait for an invitation, as shown in Figure 12-3, to appear.

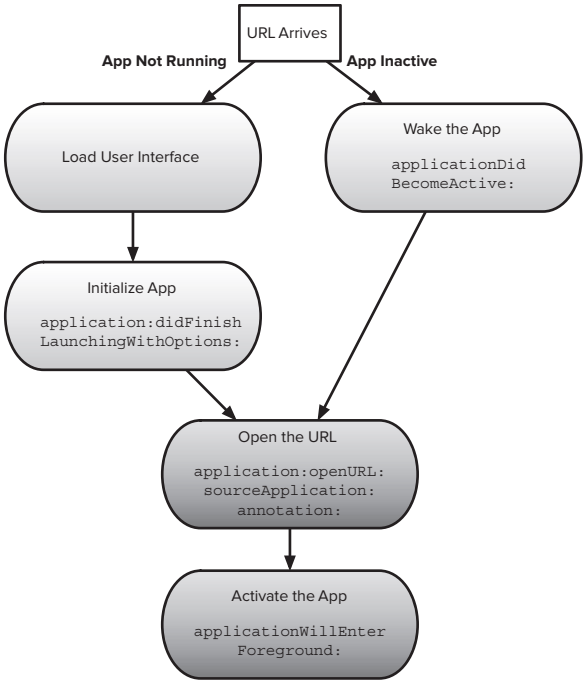


FIGURE 12-2

FIGURE 12-3

When tapping a row of the `UITableView`, Auctioneer calls the `GKSession`'s `connectToPeer:withTimeout:`, which prompts that peer's `GKSessionDelegate` to receive `session:didReceiveConnectionRequestFromPeer:`, which shows the invitation `UIAlertView`. If the remote peer accepts, it calls `acceptConnectionFromPeer:error:` on its `GKSession`, and the local peer receives `session:peer:didChangeState:` with a new state of `GKPeerStateConnected`. Other peer states are shown in Table 12-1. The remote peer can then enter the auction UI and wait for the auction to start. If the remote peer declines the invitation, it calls `denyConnectionFromPeer:` and the local peer receives `session:connectionWithPeerFailed:withError:`. Peers that reject invitations simply stay in the auction lobby. Figure 12-4 illustrates the methods called on the local and remote peer during the connection process.

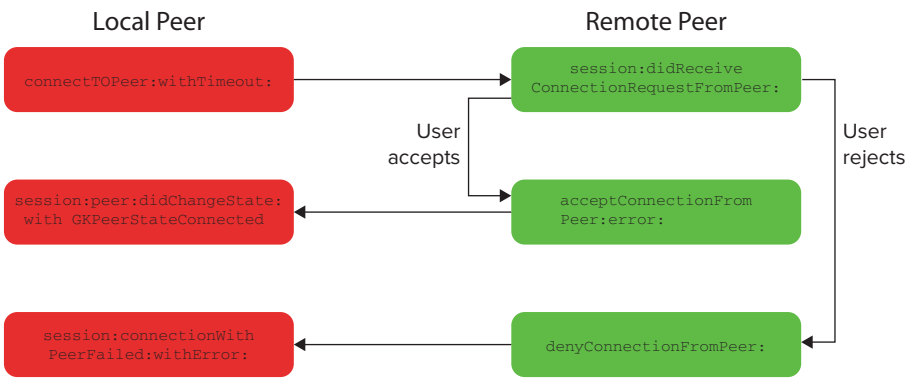


FIGURE 12-4

TABLE 12-1: Peer Connection States

STATE	DESCRIPTION
`GKPeerStateAvailable`	The peer is eligible to connect, but no connection exists yet.
`GKPeerStateUnavailable`	The peer is not eligible to connect.
`GKPeerStateConnected`	The peer is already connected to this session.
`GKPeerStateConnecting`	The peer has started the connection process but has not reached `GKPeerStateConnected`.
`GKPeerStateDisconnected`	The peer is not connected to this session.

The auction host repeats the invitation process until she has invited up to six people and then taps the start button to begin the auction. The auction start message is the first custom datagram sent through the `GKSession`.

Sending Data to Peers

Before exchanging data with other peers, you must first identify all possible types of messages you need to send to the group. Each application defines its own datagram types, which could range from just one to many dozens for a complex app. Remember that the largest possible message size is 87 kilobytes, so design your messages to fit within that constraint. In Game Kit Auctioneer, there are four possible datagrams:

- `GAPacketTypeAuctionStart`: Informs all peers that the auction has started
- `GAPacketTypeBid`: Submits a bid to the auctioneer
- `GAPacketTypeAuctionStatus`: Informs all peers of the current highest bid
- `GAPacketTypeAuctionEnd`: Informs all peers that the auction is over, the name of the winner, and the winning bid

Listing 12-2 shows the four message types defined as an enumeration `GAPacketType`, which maps them to integer values.

LISTING 12-2: Packet Type Enumeration (GANetworkingPackets.h)

```
typedef enum {
    GAPacketTypeAuctionStart = 0,
    GAPacketTypeAuctionEnd,
    GAPacketTypeAuctionStatus,
    GAPacketTypeBid
} GAPacketType;
```

The integer mapping is important because you can prefix the data of the packet with the integer value of the `GAPacketType`, allowing each peer to easily determine the packet type and how to decode it. When sending any binary data over the network, it must be sent in network byte or big-endian order. This step should always be done; however, it is especially important because the ARM processors used in iOS devices natively use little-endian order.

ENDIANNESS

Big- and little-endian byte orders are ways to represent a set of binary data. Big-endian ordering stores the bytes from most significant to least significant, whereas little-endian is the inverse. An example of big-endian ordering is a telephone number, where the groups of digits are listed from most significant (the country code) to least significant (the subscriber number). Endianness has its roots in hardware implementations for storing values in memory, and the network byte order was standardized to prevent a little-endian machine from inadvertently interpreting binary data that is stored in big-endian order without knowing that translation was required. The names *big-endian* and *little-endian* originated in Jonathon Swift's novel *Gulliver's Travels* and describe two different ways to break a boiled egg.

When the host is ready to start the auction, it sends the start packet that includes the host's peer ID, the item up for auction, and the number of other peers that will be bidding. Listing 12-3 covers populating the packet's data dictionary, serializing it, and sending it to `GANetworkingManager` for transmission. The keys of the `NSMutableDictionary` must match between the sender's message creation code and the remote peer's receiving code.

LISTING 12-3: Populating and Serializing a Data Packet (GALobbyViewController.m)

```
/*
 * tell all participants this auction is starting
 */
NSMutableDictionary *dataDict = [NSMutableDictionary dictionary];
GAPeer *devicePeer = [[GANetworkingManager sharedManager] devicePeer];

// auction owner
[dataDict setObject:devicePeer.peerID forKey:@"ownerPeerID"];

// item name
[dataDict setObject:itemName forKey:@"itemName"];

// number of participants
[dataDict setObject:[NSNumber numberWithInt:[confirmedPeers count]]
             forKey:@"numberOfParticipants"];

// participants
```

continues

LISTING 12-3 *(continued)*

```
if ([confirmedPeers count] > 0) {
    GAPeer *peer = [confirmedPeers objectAtIndex:0];
    [dataDict setObject:peer.peerID forKey:@"participant1PeerID"];
}

// other 5 participants removed for brevity

NSMutableData *data = [[NSMutableData alloc] init];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
                             initForWritingWithMutableData:data];
[archiver encodeObject:dataDict forKey:@"AuctionStarted"];
[archiver finishEncoding];

// send the message
[[GANetworkingManager sharedManager] sendPacket:data
                                         ofType:GAPacketTypeAuctionStart];
```

When the packet has been populated, it must be sent across the airwaves to all other interested parties. The following code snippet, found in code file `GANetworkingManager.m`, shows how to send a packet type and its data to all peers connected to a Game Kit session. `CFSwapInt32HostToBig()` is used to convert the 32-bit integer `GAPacketType` to big-endian ordering. Notice that the packet's `data` is not converted to big-endian order because it was encoded by `NSKeyedArchiver`, which creates `NSData` objects that are endian-independent. For more information on converting to other data types, see `https://developer.apple.com/library/mac/#documentation/CoreFoundation/Reference/CFByteOrderUtils/Reference/reference.html`.

```
- (void)sendPacket:(NSData*)data ofType:(GAPacketType)type {
    NSMutableData *newPacket = [NSMutableData dataWithCapacity:(
                                    [data length]+sizeof(uint32_t))];

    // data is prefixed with GAPacketType so the peer knows how to handle it
    uint32_t swappedType = CFSwapInt32HostToBig((uint32_t)type);
    [newPacket appendBytes:&swappedType length:sizeof(uint32_t)];
    [newPacket appendData:data];

    // reliably send the packet
    NSError *error;
    if (![_session sendDataToAllPeers:newPacket
                         withDataMode:GKSendDataReliable
                                error:&error]) {

        NSLog(@"Error sending packet: %@",[error localizedDescription]);
    }
}
```

Each peer receives the data packet and passes it to `GANetworkingManagerAuctionDelegate`'s `manager:didReceivePacket:ofType:` method for further action, as shown in the following snippet from the code file `GANetworkingManager.m`. The packet type is decoded from the beginning of the data packet, and the remaining data is restored into an exact copy of the `NSData` that was originally sent.

```
- (void)receiveData:(NSData*)data
           fromPeer:(NSString*)peerID
          inSession:(GKSession*)session
            context:(void*)context {

    GAPacketType header;
    uint32_t swappedHeader;

    if ([data length] >= sizeof(uint32_t)) {
        // separate the bytes of the header into swappedHeader
        [data getBytes:&swappedHeader length:sizeof(uint32_t)];

        // convert to the host's endianness
        header = (GAPacketType)CFSwapInt32BigToHost(swappedHeader);

        // separate the remaining bytes of the message into payload
        NSRange payloadRange = {sizeof(uint32_t), [data length]-sizeof(uint32_t)};
        NSData* payload = [data subdataWithRange:payloadRange];

        // tell the auction that we received a packet
        [auctionDelegate manager:self
                 didReceivePacket:payload
                           ofType:header];
    }
}
```

After the network manager processes the packet, it is given to `GAAuctionViewController` to be interpreted, and the auction state is updated to reflect its contents. The code in Listing 12-4 decodes the `GAPacketTypeAuctionStart` packet, but similar code is used for the other three packet types. The start packet populates the list of participants and updates the `biddingHasStarted` flag, which prevents peers from displaying the bid UI before the auction has begun. Figures 12-5 and 12-6 show the auction view controller from the perspective of the host and a participant, respectively.

LISTING 12-4: Decoding a Packet (GAAuctionViewController.m)

```
- (void)manager:(GANetworkingManager*)manager didReceivePacket:(NSData*)data
        ofType:(GAPacketType)packetType {

    switch (packetType) {
        case GAPacketTypeAuctionStart: {
            NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                 initForReadingWithData:data];
            NSDictionary *dataDict = [unarchiver decodeObjectForKey:
                 @"AuctionStarted"];
            [unarchiver finishDecoding];

            // item name
            self.itemName = [dataDict objectForKey:@"itemName"];

            // participants
```

continues

LISTING 12-4 *(continued)*

```
                self.peerList = [[NSMutableArray alloc] init];

                int numberOfParticipants = [[dataDict
                      objectForKey:@"numberOfParticipants"] intValue];

                //NSString *ownerPeerID = [dataDict objectForKey:@"ownerPeerID"];

                NSString *p1PeerID = [dataDict objectForKey:@"participant1PeerID"];
                if (numberOfParticipants > 0) {
                    [self.peerList addObject:[[GAPeer alloc] initWithPeerID:p1PeerID]];
                }

                // update UI with the info from this packet
                [self.tableView reloadData];

                // allow participants to make bids
                biddingHasStarted = YES;

                break;
            }
        }
    }
```

FIGURE 12-5

FIGURE 12-6

Now that Game Kit Auctioneer can send and receive custom datagrams, the rest of the sample code simply implements the business rules of an auction. In this specific application, peers are not disconnected after the auction finishes; however, other apps might have a requirement to disconnect either the host or other peers. For example, if there were a rule that a person could participate in only one auction per day, she should disconnect from the rest of the session. To do this, a peer can call `GKSession`'s `disconnectFromAllPeers`. If a peer determines that another peer should be forcibly disconnected from the session, it can call `disconnectPeerFromAllPeers:` with the offending peer's peer ID.

CLIENT-SERVER COMMUNICATION

Implementing a `GKSession` with peers in both `GKSessionModeServer` and `GKSessionModeClient` modes is similar to the P2P session, with only a few significant differences. Peers that connect to a session as servers will only be able to see other peers connected as clients, and client peers can only see other peers connected as servers. The following code example demonstrates connecting a server peer and client peer to a Game Kit session.

```
#define kGameKitSessionID @"auctioneer1.0"

// create a new GameKit session as a server
_session = [[GKSession alloc] initWithSessionID:kGameKitSessionID
                                    displayName:nil
                                    sessionMode:GKSessionModeServer];

// create a new GameKit session as a client
_session = [[GKSession alloc] initWithSessionID:kGameKitSessionID
                                    displayName:nil
                                    sessionMode:GKSessionModeClient];


_session.delegate = self;
```

The first difference from P2P networking is that client-server sessions have a maximum limit of 16 connected peers, which is much higher than the P2P case because of the simplified network topology. The simpler networking logic in the framework also results in a slight performance boost over P2P code. In many cases, the same business rules that might lend themselves to a P2P network can also be implemented with a client-server network as well, with minimal extra development time.

Another difference between P2P and client-server is that in the latter, peer visibility changes based on its session mode. Server peers can see only client peers, and client peers can see only server peers. This behavior ensures that each session has only one server peer and one or more client peers. A common misconception is that all networking traffic is routed through the server; however, just as in a P2P configuration, all datagrams are visible to all peers. To prevent client peers from interpreting datagrams that were not intended for them, you should include the recipient's peer ID in the datagram, and then all peers that don't match that peer ID should ignore it. In Game Kit Auctioneer, a similar check is done to choose the alert style for a winning bid. If the device's peer ID were the winning peer ID, the alert displays "You Won the Auction …," but if the peer ID is a remote peer, the alert displays "{*peer name*} won the auction …" instead. The following code snippet from the code file `GAAuctionViewController.m` demonstrates this check as part of the processing for the auction end datagram.

```
- (void)manager:(GANetworkingManager*)manager didReceivePacket:(NSData*)data
ofType:(GAPacketType)packetType {

    switch (packetType) {
        case GAPacketTypeAuctionEnd: {
            NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
                initForReadingWithData:data];
```

```
            NSDictionary *dataDict = [unarchiver
                decodeObjectForKey:@"AuctionFinish"];
            [unarchiver finishDecoding];

            // update data model
            NSInteger winningBid = [[dataDict objectForKey:@"winningBid"]
                intValue];
            NSString *winnerPeerID = [dataDict objectForKey:@"winnerPeerID"];

            // tell the user who won
            NSString *message;

            if ([winnerPeerID isEqualToString:
                [GANetworkingManager sharedManager].devicePeer.peerID]) {

                message = [NSString stringWithFormat:@"You won the auction with a
                    bid of $%i!", winningBid];
            } else {
                message = [NSString stringWithFormat:@"%@ won the auction with a
                    bid of $%i!",
                    [[GANetworkingManager sharedManager] displayNameForPeer:
                        [[GAPeer alloc] initWithPeerID:winnerPeerID]],
                winningBid];
            }

            UIAlertView *finishedAlert = [[UIAlertView alloc]
                initWithTitle:@"Auction Finished"
                message:message
                delegate:self
                cancelButtonTitle:nil
                otherButtonTitles:@"OK", nil];

            finishedAlert.tag = 700;
            [finishedAlert show];
        }
    }
}
```

SUMMARY

Game Kit fits a unique role in the iOS ecosystem because of its Bluetooth integration to create a network of nearby devices. Its Wi-Fi functionality presents an easy-to-use wrapper around Bonjour services; however, the real benefit comes from supporting both networking technologies with the same code base. Its innovative P2P model enables applications to provide unparalleled flexibility when joining networks, and the client-server model provides a more familiar environment with improved stability.

13

Ad-Hoc Networking with Bonjour

WHAT'S IN THIS CHAPTER?

- Using zero configuration networking
- Resolving and connecting to Bonjour services
- Implementing Bonjour to provide an excellent user experience

WROX.COM DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `www.wrox.com/WileyCDA/WroxTitle/Professional-iOS-Network-Programming-Connecting-the-Enterprise-to-the-iPhone-and-iPad.productCd-1118362403.html` on the Download Code tab. The code for this chapter is in the Chapter 13 download and is divided into two example apps, which both contain a shared Bonjour library:

- The Associate Help application publishes a Bonjour service and acts as a host in communication with a single client.
- The Consumer Help application browses for available Bonjour services, acts as the client, and requests a connection with the host.
- A shared library containing `Bonjour`, a Bonjour Service, and `BonjourBrowser`, a Bonjour Browser class, can be customized and dropped into your project to abstract the publication, discovery, resolution, and communication aspects from your front end.

Consumers using iOS devices have a lot of choices for applications, and those apps must provide an excellent user experience to gain recognition in a crowded marketplace. With the extensive penetration of networked devices and Wi-Fi networks, this presents an opportunity

for companies to, quite literally, connect with their customers, engage them, and deliver an experience that exceeds their expectations.

Bonjour is such a technology; it enables devices to easily discover and connect to other devices on the same network with limited or no user involvement. The framework has a number of applicable mobile use cases ranging from network-based game play, file sharing between devices, and even home automation. Although Bonjour is not limited to mobile devices, it is a practical use case with popular examples including Apple technology such as the Remote application, AirPrint, and Game Kit, and it is also how TiVo enables customers to record on one TiVo box and view on another. There are a number of third-party applications that use Bonjour to share files, contact details, and calendar information between iOS-based and OS X-based applications.

This chapter provides a brief history of Bonjour and an overview of the technology before stepping through two example applications to demonstrate how to integrate Bonjour into an application. These examples reinforce the topics covered in this chapter by solving a common problem in retail: quickly getting associates in front of inquisitive customers to provide that exceptional shopping experience. This pair of applications consists of an internal, employee-only application and a consumer-facing application like you would find in the App Store.

ZEROCONF OVERVIEW

Bonjour is a technology that enables devices to easily discover and connect with other devices on the same network. It was released in 2002 under the moniker Rendezvous and in 2005 was eventually renamed to Bonjour. Although Bonjour can greatly enhance the user experience, the real power and impact lies with application developers. Bonjour gives developers the flexibility and freedom to perform networking tasks such as detecting and connecting to a supported device without requiring user input.

More specifically, Bonjour is Apple's implementation of *zero configuration networking*, or zeroconf. *Zeroconf* is a set of techniques aimed at simplifying networking by alleviating the need for components such as *Dynamic Host Configuration Protocol* (*DHCP*) and *Domain Name System* (*DNS*) servers. Zeroconf was designed to accommodate the creation of new, evolving classes of networked products. Zeroconf currently operates to fulfill three main requirements: addressing, resolution, and discovery.

Addresses

Devices need the capability to secure a network address without the overhead of a DHCP server. Zeroconf uses *link-local* addressing, which enables hosts on the same link, or network, to communicate with each other. iOS natively supports automatic configuration for IPv4 and IPv6 addresses as defined in RFC 3927 (`http://www.tools.ietf.org/html/rfc3927`) and RFC 2462 (`http://www.tools.ietf.org/html/rfc2462`), respectively. IPv4 link-local addresses fall within the 169.254/16 prefix, and IPv6 addresses are assigned with the fe80::/64 prefix. Each of these prefixes has been reserved for link-local addressing.

Resolution

Part of zeroconf's intent is that devices be able to refer to services on the network by a name instead of only an address. Each device provides a unique name to identify itself, and these names end with `.local`. The client can provide this name as long as there is not another device with that name on the network. Host names are analogous to the link-local addresses discussed in the previous section in that they are meaningful only to the network on which they originated.

Zeroconf uses *Multicast Domain Name Service* (*mDNS*) to provide name resolution without the need to configure a conventional DNS server. mDNS is similar to unicast DNS but is implemented over a multicast protocol where each device on the network actively listens for DNS queries. These queries, ending in `.local`, are sent to the mDNS multicast address, and the device with the corresponding name replies with its address. The link-local mDNS multicast address is 224.0.0.251 for IPv4 and ff02::fb for IPv6.

There are two types of mDNS queries: *One-Shot Multicast* and *Continuous Multicast*. The One-Shot approach is simpler and takes the first response that it receives without listening for additional responses. Although simple, it can fulfill the requirement where an end user enters a fully qualified local address, such as `http://njones.local`, or `http://OfficeJet6300.local`.

Continuous querying does not assume that a single response paints the entire picture. It is asynchronous, and as its name implies, continues to listen for responses and relays them as they are received. Typically, it displays a list of all the available services, for example, all available printers on a network. If you stopped listening for results after the quickest printer responded, you would continually present users with a single-item list. This might not necessarily be the optimal printer available and can lead to a poor user experience.

To support continuous querying, the host performing the query should implement *known answer suppression*, which informs responders that they have already replied to a particular query so that they do not do so again. A number of other efficiencies are built into mDNS that you can find at `http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns.txt`.

Discovery

Devices must browse for available services on a network without maintaining a service directory. The Apple implementation of zeroconf uses *DNS-based Service Discovery* (*DNS-SD*) to advertise available services over the network. Each available service publishes its instance name, service type, and domain. DNS-SD paired with mDNS forms the basis for Bonjour. DNS-SD strives to provide a method to query services by type and to abstract the network address layer from the user by persisting instance names. If a user chooses a service today, a printer for example, that service should continue to function tomorrow even if the underlying network address of that service changes.

To provide this functionality, Bonjour uses a tuple of the instance name, service type, and domain. This tuple uniquely identifies a service and is the preferred method to store a reference to a service, instead of just the address. Service types are given on a first-come, first-served basis. Custom service types should be registered with the Internet Assigned Numbers Authority (IANA),

which is simple and free; see RFC 6335 (`http://tools.ietf.org/html/rfc6335`). One benefit of registering with IANA is that it manages service type conflicts for you. For a list of the currently registered service types, visit `http://www.dns-sd.org/ServiceTypes.html`.

Services are represented by the format *`<instance name>`* . *`<service type>`* . *`<domain>`*. For example, a laptop broadcasting the iTunes Home Sharing service on the local network might be `njones-mbp._home-sharing._tcp.local`. Here are some additional details for each component.

- **Instance name:** Configurable and is the user-friendly name of the service displayed to users. According to zeroconf documentation, implementations should not require that a value be specified.
- **Service type:** Pair of DNS labels. The first label is an underscore followed by the service type, or Application Protocol Name such as `_home-sharing`. Technically, this name can be any value as long as other devices you intend to communicate with also know it; however, it is best to register with IANA as discussed previously. The second label is either `_tcp` for services running over TCP or `_udp` for all others. See `http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt` for more information.
- **Domain:** Represents the DNS subdomain within which the service name is registered. This value could be `local`. Subdomains are supported and provide a simple means of organizing services. For example, you might broadcast printers on `printers.apple.com` to unclutter `apple.com`.

BONJOUR OVERVIEW

Given that Bonjour is a zeroconf implementation, it's natural to assume that deploying and connecting to a Bonjour service follows a series of steps similar to those needed for zeroconf. However, iOS abstracts most of the low-level networking and leaves developers with a set of APIs and a simple, four-step process. The process includes publishing, browsing for, resolving, and ultimately connecting with a service. After the service is broadcast on the network, devices browsing for the same type can discover it. When a device finds a wanted service, it attempts to resolve the service's address. After determining the service's address, the host and client establish a two-way communication channel on which to share data. This communication channel is independent of Bonjour and directly handled by each individual app. Figure 13-1 outlines the Bonjour discovery process, and each of these phases is covered further in the following sections.

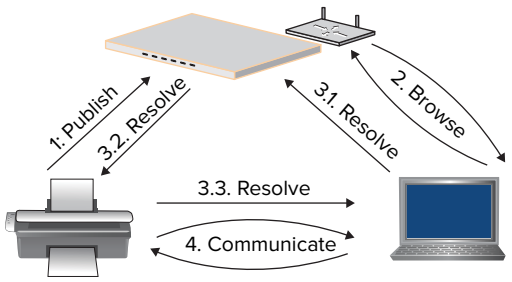


FIGURE 13-1

Publishing a Service

The `NSNetService` class represents Bonjour services. When you want to register a Bonjour service, you must create an `NSNetService` object that may be published. The method to create a publishable

Bonjour service is `initWithDomain:type:name:port:` as demonstrated here, which assumes an `NSNetService` instance variable named *`service`* exists.

```
service = [[NSNetService alloc] initWithDomain:@""
                                          type:@"_serviceType._tcp."
                                          name:name
                                          port:port];
service.delegate = self;
```

The first three parameters — `domain`, `type`, and `name` — should look familiar. As discussed earlier in the "Discovery" section, these three parameters align with the three components of a service name. Technically, only the `type` and `port` parameters are required for a service to be published. Typically, `domain` is left empty so that the service is made available on all possible domains on the network. The `name` parameter is optional, and if a value is not specified, the device name will be used by default.

> **NOTE** *There is a similar method,* `initWithDomain:type:name:`*, intended for use when the* `domain`*,* `type`*, and* `name` *are already known and the application wants to circumvent the browsing process. It is typically used when the client has previously connected to a service. This is discussed further in the "Resolving a Service" section.*

The `type` argument is required and identifies how and what the service does. It consists of the service type, or Application Protocol Name, and the transport protocol. The two possible values for transport protocol are `._tcp` for services running over TCP and `._udp` for any others. Most iOS applications will use `._tcp`. As previously mentioned in the "Discovery" section, the Application Protocol Name can be any value you want, but it is recommended to register the name with `http://www.iana.org`. The Application Protocol Name is limited to a 15-character maximum. Apple provides an informative resource on domain naming conventions, currently available at `https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/NetServices/Articles/domainnames.html`.

The last parameter is `port`, which specifies the application's port so that data is properly routed while communicating with a connected peer. The port you specify during service creation is part of the address that is ultimately resolved during the connection process. A port can be used by only one application at a time, so it is imperative to choose a port that is not already in use. A best practice is to allow the kernel to assign the port from those currently available by passing a `0` for a port to the first `CFSocketSetAddress()` function call.

Although not required for Bonjour to facilitate service discovery, each application that intends to accept connection requests must configure a listening socket to handle those requests. Unfortunately, in iOS this requires developers to jump down a level from Cocoa to CFNetwork, which is covered in detail in Chapter 8, "Low-Level Networking" Listing 13-1 shows how to let the system assign an available port for the service and register a socket callback that listens for connection attempts.

LISTING 13-1: Configuring Socket and Fetching Port (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
CFSocketContext socketCtxt = {0, (__bridge void*)self, NULL, NULL, NULL};
ipv4socket = CFSocketCreate(kCFAllocatorDefault,
                            PF_INET,
                            SOCK_STREAM,
                            IPPROTO_TCP,
                            kCFSocketAcceptCallBack,
                            (CFSocketCallBack)&BonjourServerAcceptCallBack,
                            &socketCtxt);

ipv6socket = CFSocketCreate(kCFAllocatorDefault,
                            PF_INET6,
                            SOCK_STREAM,
                            IPPROTO_TCP,
                            kCFSocketAcceptCallBack,
                            (CFSocketCallBack)&BonjourServerAcceptCallBack,
                            &socketCtxt);

if (ipv4socket == NULL || ipv6socket == NULL) {
    if (ipv4socket) CFRelease(ipv4socket);
    if (ipv6socket) CFRelease(ipv6socket);
    ipv4socket = NULL;
    ipv6socket = NULL;
    return;
}

int yes = 1;
setsockopt(CFSocketGetNative(ipv4socket),
           SOL_SOCKET,
           SO_REUSEADDR,
           (void *)&yes,
           sizeof(yes));

setsockopt(CFSocketGetNative(ipv6socket),
           SOL_SOCKET,
           SO_REUSEADDR,
           (void *)&yes,
           sizeof(yes));

// set up the IPv4 endpoint
// if port is 0, causes the kernel to choose a port
struct sockaddr_in addr4;
memset(&addr4, 0, sizeof(addr4));
addr4.sin_len = sizeof(addr4);
addr4.sin_family = AF_INET;
addr4.sin_port = htons(port);
addr4.sin_addr.s_addr = htonl(INADDR_ANY);
NSData *address4 = [NSData dataWithBytes:&addr4 length:sizeof(addr4)];

if (kCFSocketSuccess != CFSocketSetAddress(ipv4socket,
```

```
                                                (__bridge CFDataRef)address4)) {

    NSLog(@"Error setting ipv4 socket address");
    if (ipv4socket) CFRelease(ipv4socket);
    if (ipv6socket) CFRelease(ipv6socket);
    ipv4socket = NULL;
    ipv6socket = NULL;
    return;
}

if (port == 0) {
    // get the port number, port will be used for IPv6 address and service
    NSData *addr = (__bridge NSData *)CFSocketCopyAddress(ipv4socket);
    memcpy(&addr4, [addr bytes], [addr length]);
    port = ntohs(addr4.sin_port);
}

// set up the IPv6 address
struct sockaddr_in6 addr6;
memset(&addr6, 0, sizeof(addr6));
addr6.sin6_len = sizeof(addr6);
addr6.sin6_family = AF_INET6;
addr6.sin6_port = htons(port);
memcpy(&(addr6.sin6_addr), &in6addr_any, sizeof(addr6.sin6_addr));
NSData *address6 = [NSData dataWithBytes:&addr6 length:sizeof(addr6)];

if (kCFSocketSuccess != CFSocketSetAddress(ipv6socket,
                                                (__bridge CFDataRef)address6)) {

    NSLog(@"Error setting ipv6 socket address");
    if (ipv4socket) CFRelease(ipv4socket);
    if (ipv6socket) CFRelease(ipv6socket);
    ipv4socket = NULL;
    ipv6socket = NULL;
    return;
}

// set up sources and add sockets to run loop
CFRunLoopRef cfrl = CFRunLoopGetCurrent();
CFRunLoopSourceRef src4 = CFSocketCreateRunLoopSource(kCFAllocatorDefault,
                                                      ipv4socket,
                                                      0);

CFRunLoopAddSource(cfrl, src4, kCFRunLoopCommonModes);
CFRelease(src4);

CFRunLoopSourceRef src6 = CFSocketCreateRunLoopSource(kCFAllocatorDefault,
                                                      ipv6socket,
                                                      0);

CFRunLoopAddSource(cfrl, src6, kCFRunLoopCommonModes);
CFRelease(src6);
```

In this listing the application first creates a socket context and two sockets, `PF_INET` for IPv4 and `PF_INET6` for IPv6. `kCFSocketAcceptCallback` instructs the socket to accept connection requests and call the callback function passed as a pointer to argument six, `(CFCallBack)&BonjourServerAcceptCallback`, which is discussed in Listing 13-6. The last argument is the socket context, created to pass a reference to `self`, which is ultimately passed to the callback function. This is done because C-functions do not have access to Objective-C constructs. In *Automatic Reference Counting* (*ARC*)-enabled applications, the pointer cast must be bridged using `__bridge`. After the sockets are created, the application instructs the kernel to reuse ports in the `TIME_WAIT` state by calling the `setsockopt` function and passing `SO_REUSEADDR` for the third parameter.

The application then creates an IPv4 address struct, `sockaddr_in`, and passes an instance variable, `port`, which has been initialized to `0` (zero), to `sin_port`. Passing a port of zero instructs the kernel to choose an available port. The application then assigns the address to the IPv4 socket using the `CFSocketSetAddress()` function. When the application sets the IPv4 socket address, it fetches the actual port assigned by the kernel using `CFSocketCopyAddress()` and uses that port to create an IPv6 address struct, `sockaddr_in6`, and sets the IPv6 socket address similar to what was done for the IPv4 socket. After both socket addresses have been assigned, each socket is registered with the *run loop*.

PORT ASSIGNMENT

When manually assigning port numbers, you must understand what ports are available. In general, you should never use a port between 0 and 1023. These are *well-known ports* assigned to protocols such as HTTP. Ports 1024 through 49151 are publicly available for use but are *registered ports* with IANA. If you want to use a port in this range, it should be registered with IANA. Ports 49152 to 65535 are available for use without registration. Ports assigned by the kernel fall within the latter range.

`NSNetService` is integrated into the application's *run loop* and communicates various state changes via a series of delegate calls from the `NSNetServiceDelegate` protocol. Each application has a main run loop in which certain objects can be registered. `NSNetService` objects are scheduled in the current threads run loop when they are created.

Registering with the run loop allows an object to perform a task and issue any necessary delegate calls each time through the run loop. Most high-level networking APIs in iOS register with the run loop so that they may listen for network activity. `NSNetService` objects can be scheduled in a run loop using the `scheduleInRunLoop:forMode:` method. However, unless your service is operating on a secondary thread or you need to specify a different mode, this is typically not needed. If the application does need to schedule the service on a different run loop, the application should remove the service from the current run loop using the method `removeFromRunLoop:forMode:`. After the service is removed from the current run loop, it is safe to schedule it again.

When the application is ready to advertise a service to the network, it calls the service's `publish` method. By default, if the name specified for the new service already exists on the network, it will

be renamed when calling `publish`. If the application requires additional control over the publication process, it should use the `publishWithOptions:` method. Currently only one option is available, `NSNetServiceNoAutoRename`, which suppresses the renaming behavior.

Success or failure of the publishing process is communicated via the `netServiceDidPublish:` and `netService:didNotPublish:` delegate methods, respectively. The second parameter of `netService:didNotPublish:` is an `NSDictionary` object containing two key-value pairs. One pair is the error domain, which can be accessed with the key `NSNetServicesErrorDomain`, and the other is the error code, which can be accessed with the key `NSNetServicesErrorCode`. Error codes are represented by the `NSNetServicesError` enumeration. In the case of a service name collision, for example, the `netService:didNotPublish:` delegate method would be called with an error code of `NSNetServicesCollisionError`. Errors can also occur after a service is successfully published. The `netServiceWillPublish:` delegate method is called before a service is actually advertised on the network but is not called if the service will not be published due to an error.

Bonjour services can also contain something known as a *TXT record*. TXT records are a mechanism to store custom, key-value pairs of additional information about the service. The intent is that TXT records be used to convey small pieces of information that are not necessary prior to establishing a connection with a service. The intended size of 100 bytes or less makes it ideal to communicate a service version number to clients, for example. TXT records can be set by the publisher and read, but not altered, by the client. Reading TXT records are covered in the "Resolving a Service" section later in this chapter. `NSNetService` provides a class method `dataFromTXTRecordDictionary:` to create the appropriate record data from an `NSDictionary` as demonstrated in the following code. Dictionary keys must be `NSString` objects, and values must be `NSData` objects. If there is an error generating the proper format, the method returns `nil`.

```
- (void)netServiceDidPublish:(NSNetService *)sender {
    ...
    // Advertise the service version
    NSData *versionData = [@"1.0" dataUsingEncoding:NSUTF8StringEncoding];
    NSDictionary *txtRecord = [NSDictionary
                               dictionaryWithObject:versionData
                               forKey:@"version"];

    NSData *txtRecordData = [NSNetService
                             dataFromTXTRecordDictionary:txtRecord];
    sender.TXTRecordData = txtRecordData;
}
```

When the application is ready to stop advertising the service on the network, it calls the `stop` method. The `stop` method instructs the service to halt broadcasting, which will be executed the next time through the run loop, and the application will be notified via the `netServiceDidStop:` delegate method. After a service is stopped, it will not be discoverable, and no new connections are accepted. The service still exists, however, so the application can resume broadcasting by calling the `publish` method again. The following code is an example of how to stop a service from broadcasting on the network assuming `service` was an instance of `NSNetService`.

```
[service stop];
```

BONJOUR SERVICES IN A MULTITASKING ENVIRONMENT

Apple's introduction of multitasking in iOS 4, although a welcome addition, greatly complicated how networking is handled within applications as they enter the background and are resumed to the foreground. Applications sent to the background are candidates to be suspended. Bonjour communicates over sockets, which cannot be processed by an application when it is suspended. The operating system, however, still sees the socket as active and accepts the connection, but the suspended application cannot communicate over it.

Applications should stop listening and halt broadcasting of any Bonjour services as they prepare to enter the background. They can re-open these connections and republish services when they are brought to the foreground. For additional information, review Technical Note TN2277 available at `http://developer.apple.com/library/ios/#technotes/tn2277/_index.html`.

Browsing for Services

To provide users with the ability to choose the published service with which they would like to connect, applications need a mechanism to browse available services on the network. In iOS, browsing for services on the network is easy using the `NSNetServiceBrowser` class. Implementing `NSNetServiceBrowser` is similar to `NSNetService`, discussed in the previous section. Applications create an instance of `NSNetServiceBrowser`, optionally schedule it in a run loop, initiate the search (instead of publish), and then wait for delegate methods to be called. As with `NSNetService`, `NSNetServiceBrowser` objects are scheduled in the current thread's run loop during creation so it is not generally necessary for an application to schedule it manually. If the application does need to schedule the browser in a different run loop, it should first remove it from the current run loop and then schedule it again. The following example demonstrates how to create a service browser and initiate a search. Don't forget to set the delegate so that the application is notified of the various changes.

```
if (browser == nil) {
    browser = [[NSNetServiceBrowser alloc] init];
}

browser.delegate = self;
[browser searchForServicesOfType:@"_serviceType._tcp."
                        inDomain:@""];
```

You can see that the `searchForServicesOfType:inDomain:` method on `NSNetServiceBrowser` initiates a search on the network. The `type` specified for a search should match the `type` specified during the service publication step performed by the host. When the service browser configuration is complete and ready to commence a search, the `netServiceBrowserWillSearch:` method notifies the delegate. This method does not need to be implemented, but is a good place to update the user interface to indicate that a search is in progress.

As services are discovered the `netServiceBrowser:didFindService:moreComing:` method notifies the delegate. This method is called once for each discovered service, so the application must maintain a collection of all services if you want to display them to the user. The `moreComing` parameter indicates whether this method will be called again with additional services. To provide the optimal user experience, you should refrain from updating the user interface until the `moreComing` parameter is `NO`. If you expect a large number of services and want to provide incremental feedback, the application could "group" results and update the user interface every *n*th service discovered. Receiving a value of `NO` does not mean that more services will not be discovered in the future, such as when a new service is published. Following is one possible implementation of the `netServiceBrowser:didFindService:moreComing:` method that adds each `NSNetService` object to the `services` collection, an `NSMutableArray`, at which point you would trigger an update to the user interface.

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
           didFindService:(NSNetService *)aNetService
               moreComing:(BOOL)moreComing {

    if (![services containsObject:aNetService]) {
        [services addObject:aNetService];
    }

    if (moreComing == NO) {
        // Update UI
    }
}
```

Likewise, the `netServiceBrowser:didRemoveService:moreComing:` method notifies the delegate when a previously discovered service is no longer available. This method is also called once for each service that has been removed and includes the `moreComing` parameter, which has the same behavior as it does for the `netServiceBrowser:didFindService:moreComing:` method. The application could use this delegate method to update a collection of available services and user interface like so:

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
         didRemoveService:(NSNetService *)aNetService
               moreComing:(BOOL)moreComing {
    [services removeObject:aNetService];

    if (moreComing == NO) {
        // Update UI
    }
}
```

If the search fails for any reason, the `netServiceBrowser:didNotSearch:` method notifies the delegate. Similar to the `netService:didNotPublish:` delegate method discussed in the "Publishing a Service" section, the second parameter is an `NSDictionary` containing an error code and domain. The error code and domain can be extracted with the `NSNetServicesErrorCode` and `NSNetServicesErrorDomain` keys, respectively. The error code will be a value from the

NSNetServicesError enumeration. If a search fails, the application should inspect the error and stop the search as shown in the following code:

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
             didNotSearch:(NSDictionary *)errorDict {

    NSString *errorCode = [errorDict
                           objectForKey:NSNetServicesErrorCode];
    NSString *errorDomain = [errorDict
                             objectForKey:NSNetServicesErrorDomain];

    // alert user of the error

    [browser stop];
}
```

After the search stops, the netServiceBrowserDidStopSearch: delegate method is called. This provides the application with an opportunity to perform any necessary clean up and update the user interface to indicate that a search is no longer being performed. Depending on the application's structure, this could also be a good time to reset the service browser instance and reset the delegate as seen here.

```
- (void)netServiceBrowserDidStopSearch:
        (NSNetServiceBrowser *)aNetServiceBrowser {

    // clears browser and delegate
    // a new browser will be created
    // if search is initiated again
    browser = nil;

    // Update UI
}
```

If the application needs the capability to browse services outside of the local network, it should implement a domain search by calling the searchForBrowsableDomains method on NSNetServiceBrowser. As new domains are discovered or removed from the network, the netServiceBrowser:didFindDomain:moreComing: and netServiceBrowser:didRemoveDomain:moreComing: methods will be notified as shown here. As done with discovered services, the application should maintain a collection of domains that can be displayed to users for selection.

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
            didFindDomain:(NSString *)domainString
               moreComing:(BOOL)moreComing {

    if (![domains containsObject:domainString]) {
        [domains addObject:domainString];
    }
}

- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
          didRemoveDomain:(NSString *)domainString
```

```
                moreComing:(BOOL)moreComing {

    [domains removeObject:domainString];
}
```

Resolving a Service

Before communicating with a service, the application first needs to determine the service's network address. To do this, you need to call the `resolveWithTimeout:` method on the service with which the application wants to connect. There are two methods to retrieve an `NSNetService` object. First, the user could select one from a list fetched by an `NSNetServiceBrowser` as discussed previously. The second approach is to create an instance of `NSNetService` directly using the `initWithDomain:type:name:` method. The latter is used in situations in which the device has previously connected to the service and the application saved the connection details as demonstrated in the snippet here. Printers are a prime example of this behavior.

```
NSNetService *savedService = [[NSNetService alloc]
                                    initWithDomain:@""
                                              type:@"_serviceType._tcp."
                                              name:@"Kids Shoes Department"];
savedService.delegate = self;
[savedService resolveWithTimeout:5.0];
```

SAVING SERVICE CONNECTION DETAILS

Resolution is performed each time a service is used because the underlying address may have changed. When saving a service for future use, it is imperative that the application saves the tuple of `domain`, `type`, and `name` instead of the service's current network address. The intent of Bonjour is to abstract the network address information; therefore, as services come and go on the network they continue to function.

The application can use the saved `domain`, `type`, and `name` to resolve a service by directly creating an `NSNetService` object using the `initWithDomain:type:name:` method and sending the service the `resolveWithTimeout:` message.

Prior to calling `resolveWithTimeout:` on an instance of `NSNetService`, the application needs to assign the delegate. This delegate is notified of success and failure via the `netServiceDidResolveAddress:` and `netService:didNotResolve:` delegate methods, respectively. In addition, the `netServiceWillResolve:` method is called prior to resolving a service and presents a great opportunity for updating the user interface. The client initiates service resolution; therefore, the client needs to indicate that it conforms to the `NSNetServiceDelegate` protocol,

often in addition to `NSNetServiceBrowserDelegate`. Assuming that a user selected a service with which to connect, the code may look something like the following:

```
- (void)connectToService:(NSNetService*)service {
    // set the services delegate so the
    // app gets the resolve callbacks
    service.delegate = self;
    [service resolveWithTimeout:5.0];

    // halt browsing since the app
    // is connecting to a service
    [browser stop];
}
```

The `netServiceDidResolveAddress:` delegate method is invoked as each address is resolved for the service. The `netServiceDidResolveAddress:` method may be called more than once, especially on devices that support both IPv4 and IPv6. Address information can be partially resolved, so developers should ensure that all necessary address information is set prior to initiating a connection. Address information can be retrieved from the `addresses` property on the `NSNetService` object, which is passed to the delegate. The `addresses` property is an `NSArray` of `NSData` objects each containing a `sockaddr` struct. If address information is missing, you can expect another call to the `netServiceDidResolveAddress:` method. You can extract address components like so.

```
- (void)netServiceDidResolveAddress:(NSNetService *)sender {

    for (NSData *addressData in [sender addresses]) {
        struct sockaddr *address;
        address = (struct sockaddr*)[addressData bytes];

        switch (address->sa_family) {
            // IPv6
            case AF_INET6: {
                struct sockaddr_in6 *addr6 =
                    (struct sockaddr_in6*)address;
                //addr6->sin6_port;    // Port
                //addr6->sin6_addr;    // IP Address
                break;
            }

            // IPv4
            case AF_INET:
            default: {
                struct sockaddr_in *addr4 =
                    (struct sockaddr_in*)address;
                //addr4->sin_port;        // Port
                //addr4->sin_addr;        // IP Address
                break;
            }
        }
    }
}
```

Likewise, the `netService:didNotResolve:` delegate method is notified if there are any issues resolving the address. Similar to the `netService:didNotPublish:` and

`netServiceBrowser:didNotSearch:` delegate methods discussed previously, the second parameter of `netService:didNotResolve:` is an `NSDictionary` containing an error code and domain. The error code can be retrieved with the key `NSNetServicesErrorCode`, and the domain is fetched with the key `NSNetServicesErrorDomain`. Optionally, the application could read any available address information from the service passed to the delegate method to determine specifically what address information was not resolved.

Depending on the service being resolved, it may be necessary for the connecting application to retrieve additional custom attributes, such as a version number, from the host services *TXT record data*. TXT records are stored on the `NSNetService` object in the `TXTRecordData` property. Similar to how TXT record data is initially set, as discussed earlier in the "Publishing a Service" section, the `dictionaryFromTXTRecordData:` class method on `NSNetService` converts TXT record data to an `NSDictionary` as shown here.

```
- (void)netServiceDidResolveAddress:(NSNetService *)sender {

    NSDictionary *txtDictionary = [NSNetService
                                   dictionaryFromTXTRecordData:
                                   [sender TXTRecordData]];

    NSData *versionData = [txtDictionary objectForKey:@"version"];
    NSString *version = [[NSString alloc]
                         initWithData:versionData
                         encoding:NSUTF8StringEncoding];
    ...
}
```

As a reminder, the object stored for each key is an instance of `NSData`, which typically requires an additional statement to convert it to something meaningful. However, because the objects are instances of `NSData`, this provides a lot of flexibility around what can be transmitted to connecting services as long as the application is respectful of the intended use and size restrictions.

In addition, the connecting application can be notified of changes to TXT record data via the `netService: didUpdateTXTRecordData:` delegate method. The application must call the `startMonitoring` method on the `NSNetService` object with which it is connecting to be notified of changes. Likewise, the application should call the `stopMonitoring` method when it no longer needs to be notified of changes.

After the client application has resolved the selected service, it can connect and begin communicating with the host.

Communicating with a Service

When Bonjour has facilitated address resolution, its job is done, and the two devices are ready to connect and begin communicating. Communication is done via *streams*, which are a straightforward, device-independent method to exchange data. A stream is a sequence of bytes transmitted between the endpoints of a connection. In iOS, the concrete classes `NSInputStream` and `NSOutputStream` represent streams.

There are two methods to connect to the host: using the selected `NSNetService` objects' `hostName` and `port` properties to connect manually or using a pair of preconfigured `NSStream`

objects provided by the service. The examples in this chapter use the preconfigured streams for communication. Chapter 8 provides a more in-depth discussion on low-level networking and sockets.

You can fetch the preconfigured streams by calling the `getInputStream:outputStream:` method on the `NSNetService` object and passing `NSInputStream` and `NSOutputStream` pointers to the appropriate parameters. If the application needs only one of the streams, it should pass `NULL` instead of an `NSStream` object pointer. Assuming an application needs to only read data, the following is an example of how to retrieve a preconfigured stream.

```
- (void)netServiceDidResolveAddress:(NSNetService *)sender {
    ...
    NSInputStream *is;
    if (![sender getInputStream:&is outputStream:NULL]) {
        NSLog(@"Error getting stream");
    }

    // Input Stream
    if (is != NULL ) {
        inputStream = is;
        inputStream.delegate = self;
        [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                               forMode:NSDefaultRunLoopMode];
        if (inputStream.streamStatus == NSStreamStatusNotOpen) {
            [inputStream open];
        }

    }
    ...
}
```

This example fetches the input stream and then tests to ensure that a valid stream was returned. If a valid stream were returned, it is assigned to an instance variable used in the stream delegate method to test which stream is processed. To be notified of events via the `NSStreamDelegate`, you must set a delegate. The returned stream will not have been scheduled in any run loops, so that should also be done at this point. This allows the stream to serve its purpose, such as determining whether the host has any data available to be read, each time through the run loop. The returned stream object should not be open, but you should code defensively to avoid any possible problems. The `streamStatus` property contains a status value from the `NSStreamStatus` enumeration.

The `stream:handleEvent:` delegate method is notified of important stream events defined in the `NSStreamEvent` enumeration and listed here.

- `NSStreamEventNone`
- `NSStreamEventOpenCompleted`
- `NSStreamEventHasBytesAvailable`
- `NSStreamEventHasSpaceAvailable`
- `NSStreamEventErrorOccurred`
- `NSStreamEventEndEncountered`

Depending on the read/write requirements of the application, the events most applications are concerned with are `NSStreamEventOpenCompleted`, `NSStreamEventHasBytesAvailable`, and `NSStreamEventHasSpaceAvailable`. However, in the interest of completeness and providing the optimal experience, applications should implement `NSStreamEventErrorOccurred` so that users can be informed of any issues accordingly, and `NSStreamEventEndEncountered` so that the application can close the stream and remove it from the run loop. If an error is encountered, the application should inspect the stream's `streamError` property, an `NSError` object, for additional information.

Continuing with the example of an application that receives data via an input stream, the following is one possible implementation of the `stream:handleEvent:` delegate method assuming the data received is an `NSString` object.

```
- (void)stream:(NSStream *)aStream handleEvent:(NSStreamEvent)eventCode {
    switch (eventCode) {
        case NSStreamEventHasBytesAvailable:
            if (aStream == inputStream) {
                if (receiveData == nil) {
                    receiveData = [[NSMutableData alloc] init];
                }
                uint8_t buffer[1024];
                unsigned int len = 0;
                len = [(NSInputStream *)aStream read:buffer
                                            maxLength:1024];

                if(len) {
                    [receiveData appendBytes:(const void *)buffer
                                      length:len];

                    bytesRead = [NSNumber
                                 numberWithInt:([bytesRead intValue]+len)];

                    if (![inputStream hasBytesAvailable]) {

                        NSString *result = [[NSString alloc]
                                            initWithData:receiveData
                                            encoding:NSUTF8StringEncoding];
                        NSLog(@"** Result: %@", result);

                        // clean up
                        receiveData = nil;
                        bytesRead = nil;
                    }
                } else {
                    NSLog(@"No data found in buffer.");
                }
            }
            break;

        case NSStreamEventOpenCompleted:
            if (aStream == inputStream) {
                NSLog(@"Input Stream Opened");
            }
            break;
```

```
            case NSStreamEventEndEncountered: {
                [aStream close];
                [aStream removeFromRunLoop:[NSRunLoop currentRunLoop]
                                   forMode:NSDefaultRunLoopMode];
                break;
            }

            case NSStreamEventErrorOccurred:
                if (aStream == inputStream) {
                    NSLog(@"Input stream error: %@", [aStream streamError]);
                }
                break;

            default:
                break;
        }
    }
```

This example may look intimidating, but it's straightforward. First, the delegate method is broken down into logical parts to respond to the various `NSStreamEvent`s that are possible. When the `NSStreamEventOpenCompleted` event is received, the application simply logs it to the console. However, if the application wants to maintain a particular streams status outside of the `streamStatus` property, this is the proper place to do so. `NSStreamEventEndEncountered` is equally as straightforward in that it closes the stream that has ended and removes it from the run loop so that it does not continue to be monitored each time through the run loop. If an `NSStreamEventErrorOccurred` event is received, the application logs a statement to the console that includes the additional `streamError` details.

The code within the `NSStreamEventHasBytesAvailable` event may look foreign compared to some of the code discussed in this book, but it should be familiar from Chapter 8. The `NSStreamEventHasBytesAvailable` event is triggered when the sending system has received the `NSStreamEventHasSpaceAvailable` event and has data pending transfer. Apple recommends that applications send and receive stream data in moderate sizes, typically 512 or 1,024 bytes. Given that recommendation, it is possible that a single read event would not retrieve an entire dataset being transferred. In situations in which a dataset exceeds the buffer limit of the receiver, the delegate method is continuously called with the `NSStreamEventHasBytesAvailable` event until the entire dataset has been read from the stream. In the previous example, the data read into the buffer is appended to an `NSMutableData` instance variable until the input stream has transferred all its data. After all the data is received, the code converts this simple dataset into an `NSString` and logs it to the console. The chapter example demonstrated in the next section covers how to transfer more complex data structures.

The `getInputStream:outputStream:` method made available to `NSNetService` objects greatly simplifies the connection and communication process between devices. Although this section sits outside the bounds of Bonjour, it is important to understand how clients and services (or peers) communicate after the discovery facilitated by Bonjour is complete. The next section covers an in-depth example of Bonjour and device-to-device communication.

IMPLEMENTING BONJOUR-BASED APPLICATIONS

As mentioned during the chapter opening, the example for this chapter consists of two sample applications; one deployed to employee devices at a clothing retailer and one for its customers that is deployed as part of an existing application. Employees broadcast their availability to assist customers on the store's customer Wi-Fi network. Customers with the retailers' application can browse available employees and request help to their current specified location.

Technically, instances of the employee application will always be the host, or server side, of the interaction model. Customers fall into the client-side of the equation and always initiate any connection requests. Both sample applications follow a similar pattern in which the model abstracts all Bonjour activities and communicates to the front end via `NSNotificationCenter`.

Before the applications can communicate, they need to be speaking the same language. How structured information is transmitted depends on the requirements for both applications; however, the ones discussed in this chapter use two classes, `HelpRequest` and `HelpResponse`, which are shared between the employee and customer applications. The interaction model of the two applications is such that the customer requests help from an employee and that employee responds with her answer. Customer help requests are represented by the `HelpRequest` class, as defined in Listing 13-2.

LISTING 13-2: HelpRequest Class Definition (/Apps/Associate/associate-help/associate-help/HelpRequest.h)

```
@interface HelpRequest : NSObject <NSCoding>

@property(nonatomic,strong) NSString *question;
@property(nonatomic,strong) NSString *location;

@end
```

As you can see in Listing 13-2, the `HelpRequest` class conforms to the `NSCoding` protocol, which is a simple mechanism for serializing and unserializing structured objects. When combined with `NSKeyedArchiver` and `NSKeyedUnarchiver`, it enables applications to serialize structured objects into bytes that can be transferred over the network. Because the `HelpRequest` class is shared between applications, the transferred data can be easily converted back into a structured object on the receiving end of the transmission. Listing 13-3 outlines how to implement `NSCoding` for the `HelpRequest` class.

LISTING 13-3: HelpRequest Class Implementation (/Apps/Associate/associate-help/associate-help/HelpRequest.m)

```
#import "HelpRequest.h"

@implementation HelpRequest

@synthesize question, location;
```

continues

LISTING 13-3 *(continued)*

```
#pragma mark - NSCoding
- (void)encodeWithCoder:(NSCoder*)aCoder {
    [aCoder encodeObject:self.question forKey:@"question"];
    [aCoder encodeObject:self.location forKey:@"location"];
}

- (id)initWithCoder:(NSCoder*)aDecoder {
    self = [super init];
    self.question = [aDecoder decodeObjectForKey:@"question"];
    self.location = [aDecoder decodeObjectForKey:@"location"];
    return self;
}

@end
```

Associate responses to help requests are represented by the `HelpResponse` class. `HelpResponse` is a simple class that also conforms to `NSCoding`. `HelpResponse` is also an example of how to encode and decode different data types, compared to `HelpRequest`, as shown in Listing 13-4.

LISTING 13-4: HelpResponse Interface and Implementation (/Apps/Associate/associate-help/associate-help/HelpResponse.h/m)

```
@interface HelpResponse : NSObject <NSCoding>

@property(nonatomic,assign) BOOL    response;

@end

@implementation HelpResponse

@synthesize response;

#pragma mark - NSCoding
- (void)encodeWithCoder:(NSCoder*)aCoder {
    [aCoder encodeBool:self.response forKey:@"response"];
}

- (id)initWithCoder:(NSCoder*)aDecoder {
    self = [super init];
    self.response = [aDecoder decodeBoolForKey:@"response"];
    return self;
}
```

As mentioned earlier, the code for these base communication classes is shared between each application.

Employee Application

FIGURE 13-2

Now that the communication fundamentals are complete, it's time to build the employee application. Employees running it can advertise themselves as available to assist customers. The application consists of a single view that enables employees to set their department and toggle their availability. When launched, the employee application resembles Figure 13-2.

The core of the employee application focuses around the publication of itself as a service and handling the communication with connecting devices. This functionality lives within the `Bonjour` class, a singleton that handles all `NSNetService`-related and `NSStream`-related details. Listing 13-5 outlines the `Bonjour` class definition; note the notification constants declared above the interface. These are the notification names broadcast using `NSNotificationCenter`. All code for the employee application can be found in the Chapter 13 download folder on the companion website.

LISTING 13-5: Bonjour Interface Definition (/Apps/Associate/associate-help/associate-help/Bonjour.h)

```
#import "HelpRequest.h"
#import "HelpResponse.h"

#define kNotificationResultSet              @"NotificationObject"
#define kPublishBonjourStartNotification    @"PublishStartNotification"
#define kPublishBonjourErrorNotification    @"PublishErrorNotification"
#define kPublishBonjourSuccessNotification  @"PublishSuccessNotification"
#define kStopBonjourSuccessNotification     @"StopSuccessNotification"
#define kHelpRequestedNotification          @"HelpRequestedNotification"

@interface Bonjour : NSObject <NSNetServiceDelegate, NSStreamDelegate>

+ (Bonjour*)sharedPublisher;

- (BOOL)publishServiceWithName:(NSString*)name;

- (void)stopService;

- (void)sendHelpResponse:(HelpResponse*)response;

@end
```

Although most methods declared in Listing 13-5 resemble the core `NSNetService` methods, the approach to wrap the `NSNetService` methods enables the model to abstract as much of the publication detail as possible from the front end. The `Bonjour` class in this example is intended for use with a single service type and an empty service domain. Although that may not work for all situations, this enables the model to expose a streamlined publication process in the form of the `publishServiceWithName:` method. You need to implement the methods declared in Listing 13-5, as shown in Listing 13-6.

LISTING 13-6: Bonjour Implementation (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (BOOL)publishServiceWithName:(NSString*)name {

    // setup the listening socket for connection attempts
    // and determine a port on which to advertise the service
    if (![self setupListeningSocket]) {
        return NO;
    }

    // create the service for publishing
    // this type should be registered - iana.org
    service = [[NSNetService alloc]
                   initWithDomain:@""
                             type:@"_associateHelp._tcp."
                             name:name
                             port:port];

    if (service == nil) {
        return NO;
    }

    service.delegate = self;

    // Publish service
    [Utils postNotification:kPublishBonjourStartNotification];
    [service publish];

    return YES;
}

- (void)stopService {
    [service stop];
}
```

The `publishServiceWithName:` method sets up a listening socket and lets the kernel assign a port, as shown previously in Listing 13-1, before instantiating an `NSNetService` object. If the service is successfully created, the delegate is set, and the UI is alerted just before broadcasting it to the network. If the service is successfully published, the `netServiceDidPublish:` delegate method is notified, which, in turn, notifies the front end so that it can update the interface accordingly, as shown in Listing 13-7. If there is an error publishing the service, the `netService:didNotPublish:` delegate method is called, which also alerts the front end, as demonstrated in Listing 13-7.

LISTING 13-7: NetService Publication Delegate Methods (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)netServiceDidPublish:(NSNetService *)sender {
    [Utils postNotification:kPublishBonjourSuccessNotification];
}

- (void)netService:(NSNetService *)sender
     didNotPublish:(NSDictionary *)errorDict {
```

```
    // typically you would pass along the errorDict
    // object or some form of error messaging
    [Utils postNotification:kPublishBonjourErrorNotification];
}

- (void)netServiceDidStop:(NSNetService *)sender {
    // reset port so a new one is assigned
    port = 0;
    CFRelease(ipv4socket);
    CFRelease(ipv6socket);

    [Utils postNotification:kStopBonjourSuccessNotification];
}
```

When the service has been successfully stopped, the `netServiceDidStop:` method is called where the application should reset the port to `0` so that subsequent publication requests assign a new port during socket creation. One of the most tedious steps in publishing a service that communicates with other devices is configuring the socket and connection handler callbacks. Because all connection requests originate from the customer application in this example, only the employee application needs to plan for this.

Listing 13-1 from the "Publishing a Service" section covers how to configure the sockets as well as how to allow the kernel to assign a port for you. Although you could hardcode a port for the service to use, you may encounter conflicts that render the application useless. Therefore, it is recommended that you allow the system to choose a port for you. As referenced in Listing 13-1, the application must implement a callback function that will be called when the socket receives a connection request. Listing 13-8 demonstrates how to implement the callback function set in Listing 13-1.

LISTING 13-8: Connection Request Callback Function (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
static void BonjourServerAcceptCallBack (CFSocketRef socket,
                                         CFSocketCallBackType type,
                                         CFDataRef address,
                                         const void *data,
                                         void *info) {

    Bonjour *server = (__bridge Bonjour*)info;
    if (type == kCFSocketAcceptCallBack) {
        // AcceptCallBack: data is pointer to a CFSocketNativeHandle
        CFSocketNativeHandle socketHandle
            = *(CFSocketNativeHandle *)data;

        CFReadStreamRef readStream = NULL;
        CFWriteStreamRef writeStream = NULL;
        CFStreamCreatePairWithSocket(kCFAllocatorDefault,
                                     socketHandle,
                                     &readStream,
                                     &writeStream);
```

continues

LISTING 13-8 *(continued)*

```
            if (readStream && writeStream) {
                CFReadStreamSetProperty
                    (readStream,
                     kCFStreamPropertyShouldCloseNativeSocket,
                     kCFBooleanTrue);

                CFWriteStreamSetProperty
                    (writeStream,
                     kCFStreamPropertyShouldCloseNativeSocket,
                     kCFBooleanTrue);

                NSInputStream *is = (__bridge NSInputStream*)readStream;
                NSOutputStream *os = (__bridge NSOutputStream*)writeStream;
                [server handleNewConnectionWithInputStream:is
                                              outputStream:os];

            } else {
                // encountered failure
                // no need for socket anymore
                close(socketHandle);
            }

            // clean up
            if (readStream) {
                CFRelease(readStream);
            }

            if (writeStream) {
                CFRelease(writeStream);
            }
        }
    }
```

Similar to Listing 13-1, Listing 13-8 may look intimidating, but it is actually straightforward. First, this function is called when the socket receives a connection request from the customer application. You could think of this as serving a similar function to that of a delegate method in Objective-C. When the function is called, the first step is to fetch a reference to `self`. Because C-functions do not have access to `self`, this is accomplished by casting the `info` function parameter to the `Bonjour` class. The `info` parameter was set using the `(__bridge void*)self` when creating the socket context used to configure the listening sockets in Listing 13-1.

If the callback `type` is `kCFSocketAcceptCallBack`, meaning connection requests are accepted and child streams will be passed to the callback function, the application creates read and write streams for the socket. If both streams are created, the application instructs each to close and release the underlying native socket when the stream is released. After the stream properties have been set, they are cast to their respective Objective-C types, and then `handleNewConnectionWithInputStream:outputStream:` is called, as implemented in Listing 13-9.

LISTING 13-9: handleNewConnectionWithInputStream:outputStream: Method (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)handleNewConnectionWithInputStream:(NSInputStream*)istr
                                outputStream:(NSOutputStream*)ostr {
    inputStream = istr;
    outputStream = ostr;

    inputStream.delegate = self;
    outputStream.delegate = self;

    [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                           forMode:NSDefaultRunLoopMode];
    // output stream is scheduled in the runloop when it is needed


    if (inputStream.streamStatus == NSStreamStatusNotOpen) {
        [inputStream open];
    }

    if (outputStream.streamStatus == NSStreamStatusNotOpen) {
        [outputStream open];
    }

}
```

The `handleNewConnectionWithInputStream:outputStream:` method sets two instance variables used for stream comparison later in the process and then sets the delegate. Streams are not prescheduled in a run loop, so you must schedule them so that they can monitor for the various `NSStreamEvents`. The method then checks the status of each stream and opens them if necessary. As streams are opened the delegate is notified with the `NSStreamEventOpenCompleted` event, as shown in Listing 13-10, which simply logs the event to the console.

LISTING 13-10: stream:handleEvent: Open Completed Event (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)stream:(NSStream *)aStream
   handleEvent:(NSStreamEvent)eventCode {

    switch (eventCode) {
        case NSStreamEventHasBytesAvailable:
            if (aStream == inputStream) {
                if (receiveData == nil) {
                    receiveData = [[NSMutableData alloc] init];
                }
                uint8_t buf[1024];
                unsigned int len = 0;
                len = [(NSInputStream *)aStream read:buffer
                                            maxLength:1024];

                if(len) {
```

continues

LISTING 13-10 *(continued)*

```
                [receiveData appendBytes:(const void *)buffer
                                  length:len];

                bytesRead = [NSNumber
                             numberWithInt:([bytesRead intValue]+len)];

                if (![inputStream hasBytesAvailable]) {

                    // you could optionally keep the 'transaction'
                    // state stored so that you could determine
                    // which object you are expecting.
                    HelpRequest *request;
                    @try {
                        request =
                        [NSKeyedUnarchiver
                         unarchiveObjectWithData:receiveData];

                        NSDictionary *info =
                        [NSDictionary
                         dictionaryWithObject:request
                         forKey:kNotificationResultSet];

                        [[NSNotificationCenter defaultCenter]
                         postNotificationName:
                         kHelpRequestedNotification
                         object:nil
                         userInfo:info];

                    }
                    @catch (NSException *exception) {
                        NSString *msg =
                        @"Exception while unarchiving request data.";
                        NSLog(@"%@", msg);
                    }
                    @finally {
                        // clean up
                        receiveData = nil;
                        bytesRead = nil;
                    }
                }
            } else {
                NSLog(@"No data found in buffer.");
            }
        }
        break;

...

case NSStreamEventOpenCompleted:
    if (aStream == inputStream) {
        NSLog(@"Input Stream Opened");
    } else {
        NSLog(@"Output Stream Opened");
```

```
                }
                break;

            ...
        }
    }
```

Listing 13-10 also includes the logic to read data from the input stream. The stream delegate is notified with the `NSStreamEventHasBytesAvailable` event when the connected system receives data. The logic processing that event continues to append data from the input stream buffer until there are no more bytes available. Because the only inbound data to the employee application will be a `HelpRequest`, after all data has been read, the application attempts to create a `HelpRequest` object using `NSKeyedUnarchiver` and `NSCoding`, as discussed in Listing 13-3. If the object creation is successful, the application notifies the front end, which triggers an alert to the employee, as shown in Figure 13-3.

FIGURE 13-3

After the employee responds to the `HelpRequest` (refer to Figure 13-3), the `sendHelpResponse:` method on `Bonjour` is invoked. Listing 13-11 details the implementation of `sendHelpResponse:`.

LISTING 13-11: sendHelpResponse: Implementation (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)sendHelpResponse:(HelpResponse*)response {
    if (sendData == nil) {
        sendData = [[NSMutableData alloc] init];
    }
    NSData *responseData =
        [NSKeyedArchiver archivedDataWithRootObject:response];

    [sendData appendData:responseData];
    [outputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                            forMode:NSDefaultRunLoopMode];

    // associate is going to help customer
    // stop the service so they aren't discoverable
    // while with the customer
    if (response.response == YES) {
        [self stopService];
    }
}
```

The `sendHelpResponse:` method creates a data representation of the response using `NSKeyedArchiver` and adds it to a queue of data pending transmission. In more sophisticated applications, a more advanced queue such as an `NSMutableArray` of `NSData` may be warranted. After the `HelpResponse` data has been added to the queue, the application schedules the `outputStream` in the current run loop so that it begins actively monitoring for the `NSStreamEventHasSpaceAvailable`

event, which notifies the `stream:handleEvent:` delegate method. After the `outputStream` has been scheduled in the run loop, the service is stopped so that other customers do not attempt to request help while the employee is busy. When the employee is done helping the current customer, she will republish availability.

Listing 13-12 indicates that if there is data to be sent, the delegate writes it to the stream buffer 1 kilobyte at a time, which will be read from the stream by the customer application. After all data has been transmitted, the application cleans the queue and removes the `outputStream` from the run loop. It is rescheduled when there is additional data to transmit.

LISTING 13-12: NSStreamEventHasSpaceAvailable Logic (/Apps/Associate/associate-help/associate-help/Bonjour.m)

```
- (void)stream:(NSStream *)aStream
   handleEvent:(NSStreamEvent)eventCode {

    switch (eventCode) {
        ...
        case NSStreamEventHasSpaceAvailable: {
            if (aStream == outputStream) {
                // send data if there is some pending
                if ([sendData length] > 0) {
                    uint8_t *readBytes =
                        (uint8_t *)[sendData mutableBytes];

                    // keep track of pointer position
                    readBytes += [bytesWritten intValue];
                    int data_len = [sendData length];

                    unsigned int len =
                        ((data_len - [bytesWritten intValue] >= 1024) ?
                         1024 : (data_len-[bytesWritten intValue]));

                    uint8_t buffer [len];
                    (void)memcpy(buffer, readBytes, len);
                    len = [(NSOutputStream*)aStream
                                write:(const uint8_t *)buffer
                            maxLength:len];

                    bytesWritten =
                        [NSNumber
                         numberWithInt:([bytesWritten intValue]+len)];

                    if ([sendData length] == [bytesWritten intValue]) {
                        sendData = nil;
                        [outputStream
                            removeFromRunLoop:[NSRunLoop currentRunLoop]
                                      forMode:NSDefaultRunLoopMode];
                    }

                    if ([bytesWritten intValue] == -1) {
                        NSLog(@"Error writing data.");
```

```
                    }
                }
            }
            break;
        }
        ...
    }
}
```

Customer Application

With the host component complete and employees broadcasting their ability to help, the customer application is needed to initiate requests. The customer application consists of a two-view tab bar controller. The first view provides customers with the ability to shop and browse from their mobile device. For this example the shop and browse functionality has not been implemented. The second view presents customers with a list of available employees. This example assumes that customers always connect to the same network as employees. In practice you should provide a backup means to request help, such as a simple form. If customers are not connected to the network, it would be a better experience to disable the feature all together.

FIGURE 13-4

FIGURE 13-5

Figure 13-4 shows the shop view, and Figure 13-5 shows the help view in which a single employee is currently available. As mentioned with the employee app, all presentation tier code can be found in the Chapter 13 downloads folder. However, as with the employee application, all Bonjour browse, connection, and communication functionality has been abstracted from the presentation tier, making display and interaction with services straightforward.

The main class in the model is `BonjourBrowser`, as defined in Listing 13-13. `BonjourBrowser` is also implemented as a singleton. Similar to the employee application, interaction between the model and the front end is done via `NSNotificationCenter`. Notification constants are defined before the `@interface` declaration.

LISTING 13-13: BonjourBrowser Interface Definition (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.h)

```
#import "HelpRequest.h"
#import "HelpResponse.h"

#define kNotificationResultSet      @"NotificationObject"

#define kBrowseStartNotification    @"BonjourBrowseStartNotification"
#define kBrowseErrorNotification    @"BonjourBrowseErrorNotification"
#define kBrowseSuccessNotification  @"BonjourBrowseSuccessNotification"

#define kConnectStartNotification   @"BonjourConnectStartNotification"
```

continues

LISTING 13-13 *(continued)*

```
#define kConnectErrorNotification   @"BonjourConnectErrorNotification"
#define kConnectSuccessNotification @"BonjourConnectSuccessNotification"

#define kServiceRemovedNotification @"BonjourServiceRemovedNotification"
#define kSearchStoppedNotification  @"BonjourSearchStoppedNotification"

#define kHelpRequestedNotification  @"HelpRequestedNotification"
#define kHelpResponseNotification   @"HelpResponseNotification"

@interface BonjourBrowser : NSObject <NSNetServiceDelegate,
                                      NSNetServiceBrowserDelegate,
                                      NSStreamDelegate>

+ (BonjourBrowser*)sharedBrowser;

- (void)browseForHelp;

- (NSArray*)availableServices;

- (void)connectToService:(NSNetService*)service;

- (void)sendHelpRequest:(HelpRequest*)request;

@end
```

When customers select the Help tab, the application calls `browseForHelp`, which instantiates an `NSNetServiceBrowser` object and begins a search for the type `_associateHelp._tcp.`, which is the type specified during service publication from the employee application. When the search has been initiated, the method informs listeners of the event, as shown in Listing 13-14.

LISTING 13-14: browseForHelp Implementation (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)browseForHelp {
    if (browser == nil) {
        browser = [[NSNetServiceBrowser alloc] init];
    }

    browser.delegate = self;
    [browser searchForServicesOfType:@"_associateHelp._tcp."
                            inDomain:@""];

    [Utils postNotification:kBrowseStartNotification];
}
```

If there are issues performing the search, the `netServiceBrowser:didNotSearch:` delegate method is notified, which broadcasts a message to the front end and stops the search, as demonstrated in Listing 13-15. After the search has stopped, the `netServiceBrowserDidStopSearch:` delegate

method is called. As shown in Listing 13-15, the application takes this opportunity to clear the `browser` instance variable and notify the front end so that any related status indicators can be updated.

LISTING 13-15: netServiceBrowser:didNotSearch: and netServiceBrowserDidStopSearch: Implementations (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
             didNotSearch:(NSDictionary *)errorDict {
    // alert the user and stop the browser
    [Utils postNotification:kBrowseErrorNotification];
    [browser stop];
}

- (void)netServiceBrowserDidStopSearch:
        (NSNetServiceBrowser *)aNetServiceBrowser {

    // clears browser and delegate
    // a new browser will be created
    // if search is initiated again
    browser = nil;
    [Utils postNotification:kSearchStoppedNotification];
}
```

As services are discovered the `netServiceBrowser:didFindService:moreComing:` delegate method is invoked, which adds the service to an `NSMutableArray` instance variable that maintains available services and notifies the front end, as shown in Listing 13-16. Pay particular close attention to the `moreComing` indicator prior to notifying listeners of updates because it is called once per discovered service. In this example there should not be more than a couple of dozen services, assuming two or three per department, but searching for printers on a large corporate network could be a different story. Although this example does not sort the service collection, this is one place to do so, just before broadcasting the notification.

LISTING 13-16: netServiceBrowser:didFindService:moreComing: Implementation (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)netServiceBrowser:(NSNetServiceBrowser *)aNetServiceBrowser
           didFindService:(NSNetService *)aNetService
               moreComing:(BOOL)moreComing {

    if (![services containsObject:aNetService]) {
        [services addObject:aNetService];
    }

    if (moreComing == NO) {
        [Utils postNotification:kBrowseSuccessNotification];
    }
}
```

The front end can fetch currently available services from the model by calling `availableServices`. This enables the model to add and remove services that the front end can retrieve as needed without having to maintain its own copy. `availableServices` returns an `NSArray` of `NSNetService` objects that can be displayed in a `UITableView` (refer to Figure 13-5). When customers are ready to request help, they tap on the appropriate service. Selecting a service begins the resolution process by passing the chosen service to `connectToService:` and then presenting the customers with a view prompting for additional information, as shown in Figure 13-6.

FIGURE 13-6

If there is an issue resolving the service, the `netService:didNotResolve:` delegate method is called, which posts a notification to the front end at which point the customer is informed. As the service is resolved, the `netServiceDidResolveAddress:` delegate method is called. Listing 13-17 covers the implementation of the `netService:didNotResolve:` and `netServiceDidResolveAddress:` delegate methods. As mentioned in the "Resolving a Service" section, developers should code defensively and ensure that the necessary address information has been resolved prior to retrieving the preconfigured streams or connecting manually. Because that has already been covered, it has been left out of this listing for the sake of brevity.

LISTING 13-17: netService:didNotResolve: and netServiceDidResolveAddress: Implementations (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)netService:(NSNetService *)sender
     didNotResolve:(NSDictionary *)errorDict {
    [Utils postNotification:kConnectErrorNotification];
}

- (void)netServiceDidResolveAddress:(NSNetService *)sender {
    NSInputStream *tmpIS;
    NSOutputStream *tmpOS;
    BOOL error = NO;

    // this application requires both streams
    // if you don't get them both, that poses
    // a problem
    if (![sender getInputStream:&tmpIS outputStream:&tmpOS]) {
        error = YES;
    }

    // Input Stream
    if (tmpIS != NULL ) {
        inputStream = tmpIS;
        inputStream.delegate = self;
        [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                               forMode:NSDefaultRunLoopMode];
```

```
            if (inputStream.streamStatus == NSStreamStatusNotOpen) {
                [inputStream open];
            }

        } else {
            error = YES;
        }

        // Output Stream
        if (tmpOS != NULL ) {
            outputStream = tmpOS;
            outputStream.delegate = self;
            //output stream is scheduled in runloop when it is needed
            if (outputStream.streamStatus == NSStreamStatusNotOpen) {
                [outputStream open];
            }

        } else {
            error = YES;
        }

        if (error == NO) {
            [Utils postNotifification:kConnectSuccessNotification];
        } else {
            [Utils postNotifification:kConnectErrorNotification];
        }
    }
```

The `netServiceDidResolveAddress:` method attempts to fetch the preconfigured streams using the `getInputStream:outputStream:` method on `NSNetService`. For each stream returned, it assigns it to its instance variable counterpart, `inputStream` or `outputStream`; sets the delegate; and then checks the `streamStatus` to determine whether it should be opened. `inputStream` is scheduled with the run loop, but `outputStream` is not. `outputStream` will be scheduled with the run loop when it is needed, as was done in the employee application.

After the customer enters the necessary information on the request view and taps Submit, the application calls `sendHelpRequest:`, which converts the `request` parameter to `NSData` using `NSKeyedArchiver`, appends the data to the applications outbound queue, and then schedules the `outputStream` with the current run loop, as shown in Listing 13-18.

LISTING 13-18: sendHelpRequest: Implementation (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)sendHelpRequest:(HelpRequest*)request {

    if (sendData == nil) {
        sendData = [[NSMutableData alloc] init];
    }

    // convert the request to NSData (using NSKeyedArchiver/NSCoding)
    NSData *requestData = [NSKeyedArchiver
```

continues

LISTING 13-18 *(continued)*

```
                            archivedDataWithRootObject:request];

    [sendData appendData:requestData];
    [outputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
                            forMode:NSDefaultRunLoopMode];
}
```

After the `outputStream` is scheduled with the run loop, the application begins actively listening for the `NSStreamEventHasSpaceAvailable` event from the connected peer. When the device receives the event, it notifies the `stream:handleEvent:` delegate as discussed previously in the Employee Application section. Listing 13-19 demonstrates how to send data to the employee application. This process is the same as covered in Listing 13-12.

LISTING 13-19: stream:handleEvent: NSStreamEventHasSpaceAvailable Implementation (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)stream:(NSStream *)aStream
   handleEvent:(NSStreamEvent)eventCode {

    switch (eventCode) {
        case NSStreamEventHasSpaceAvailable: {
            if (aStream == outputStream) {
                if ([sendData length] > 0) {
                    uint8_t *readBytes =
                        (uint8_t *)[sendData mutableBytes];

                    // keep track of pointer position
                    readBytes += [bytesWritten intValue];
                    int data_len = [sendData length];

                    unsigned int len =
                        ((data_len - [bytesWritten intValue] >= 1024) ?
                         1024 : (data_len-[bytesWritten intValue]));

                    uint8_t buffer[len];
                    (void)memcpy(buffer, readBytes, len);
                    len = [(NSOutputStream*)aStream
                                write:(const uint8_t *)buffer
                            maxLength:len];

                    bytesWritten =
                        [NSNumber
                         numberWithInt:([bytesWritten intValue]+len)];

                    if ([sendData length] == [bytesWritten intValue]) {
                        sendData = nil;
                        [outputStream
                            removeFromRunLoop:[NSRunLoop currentRunLoop]
                                      forMode:NSDefaultRunLoopMode];
```

```
                    }

                    if ([bytesWritten intValue] == -1) {
                        NSLog(@"Error writing data.");
                    }
                }
            }
            break;
        }
        ...
}
```

After the data has been transmitted, the outbound queue is cleared, and the `outputStream` is removed from the run loop. It is rescheduled the next time the customer sends a request to an employee.

After the request has been sent, the customer application waits for a response from the employee. That response is delivered over the `inputStream`, and the `stream:handleEvent:` delegate method receives the `NSStreamEventHasBytesAvailable` event when there is a response. Listing 13-20 demonstrates how to handle the inbound response data.

LISTING 13-20: stream:handleEvent: NSStreamEventHasBytesAvailable Implementation (/Apps/Consumer/consumer-help/consumer-help/BonjourBrowser.m)

```
- (void)stream:(NSStream *)aStream
   handleEvent:(NSStreamEvent)eventCode {

    switch (eventCode) {
        ...
        case NSStreamEventHasBytesAvailable:
            if (aStream == inputStream) {
                if (receiveData == nil) {
                    receiveData = [[NSMutableData alloc] init];
                }
                uint8_t buffer[1024];
                unsigned int len = 0;
                len = [(NSInputStream *)aStream read:buffer
                                            maxLength:1024];

                if(len) {
                    [receiveData appendBytes:(const void *)buffer
                                      length:len];

                    bytesRead = [NSNumber
                                 numberWithInt:([bytesRead intValue]+len)];

                    if (![inputStream hasBytesAvailable]) {

                        // you could optionally keep the 'transaction'
                        // state stored so that you could determine
                        // which object you are expecting.
                        HelpResponse *response;
```

continues

LISTING 13-20 *(continued)*

```
                        @try {
                            response =
                                [NSKeyedUnarchiver
                                 unarchiveObjectWithData:receiveData];

                            NSDictionary *info =
                                [NSDictionary
                                 dictionaryWithObject:response
                                 forKey:kNotificationResultSet];

                            [[NSNotificationCenter defaultCenter]
                             postNotificationName:kHelpResponseNotification
                             object:nil
                             userInfo:info];

                        }
                        @catch (NSException *exception) {
                            NSLog(@"Exception unarchiving data.");
                            NSLog(@"Possible missing / corrupt data.");
                        }
                        @finally {
                            // clean up
                            receiveData = nil;
                            bytesRead = nil;
                        }

                    }
                } else {
                    NSLog(@"No data found in buffer.");
                }
            }
            break;
        ...
    }
}
```

After ensuring the stream being processed is the `inputStream`, the application initializes an `NSMutableData` object to store all received data. Then the application pulls 1 kilobyte off the inbound stream. If data is actually read, it is appended to the `NSMutableData` variable created earlier, and a byte counter is updated to reflect the number of bytes actually read. If the `inputStream` doesn't have any more bytes available, the application attempts to create a `HelpResponse` object using `NSKeyedUnarchiver` and then notifies the front end and passes the `response`. If there are additional bytes to be read, the delegate continues to receive the `NSStreamEventHasBytesAvailable` event until the buffer has been exhausted.

When the employee response is received, it is presented to the customer, as shown in Figure 13-7, and the request view is dismissed. However, in the help request view, after the employee submits a request, the application disables each button in the navigation bar until either an error occurs or the employee responds. More sophisticated applications may want to review other options that

enable the customer to continue to use the application while the employee responds to the request. One possible option is to add the listener to the `AppDelegate`. This becomes even more important if the service you provide allows customers to queue requests, similar to a deli ticket system, in which there may be several customers ahead of them in line.

You should now have two fully functional applications; one that publishes itself as a host and can be discovered via Bonjour, and another that uses Bonjour to browse and connect with available services. When connected, the two applications should be able to communicate seamlessly with each other.

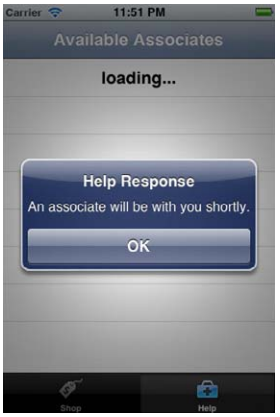


FIGURE 13-7

SUMMARY

Bonjour is a great technology to facilitate ad-hoc networking to share data between devices on the same network. This chapter has taken you through the background of Bonjour, the steps required to successfully publish a service on the network, and the process to browse and ultimately connect a host and peer. Use of Bonjour is not limited to iPhones and iPads; you can use it to discover any type of device that can publish a Bonjour service. Devices range from printers and DVR boxes to home automation tools.

The employee and customer applications provide a good framework to integrate Bonjour into your applications. However, there are other networking tools available that you should review to ensure Bonjour is best for your needs. One option is Game Kit, which is covered in Chapter 12. Chapter 8 covers low-level network programming functionality that may also be something to consider for certain requirements.

This is the final chapter in this book. All three of us thank you for buying the book, allowing us the opportunity to share our experiences with you, and for making it to the end. We hope it's been informative. Thank you, again.

— Jack Cox, John Szumski, and Nathan Jones

INDEX

A

C

D

E

F

G

H

L

M

N

Q

R

S

T

U

V

W

X

Z

Programmer to Programmer™

Connect with Wrox.

Participate

Take an active role online by participating in our P2P forums @ **p2p.wrox.com**

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble

Join the Community

Sign up for our free monthly newsletter at **newsletter.wrox.com**

Wrox.com

Browse the vast selection of Wrox titles, e-books and blogs and find exactly what you need

User Group Program

Become a member and take advantage of all the benefits

Wrox on twitter

Follow @wrox on Twitter and be in the know on the latest news in the world of Wrox

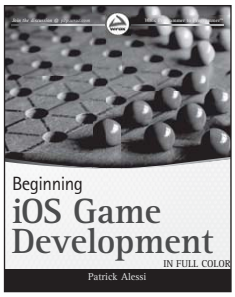
Wrox on facebook

Join the Wrox Facebook page at **facebook.com/wroxpress** and get updates on new books and publications as well as upcoming programmer conferences and user group events

Contact Us.

We love feedback! Have a book idea? Need community support?
Let us know by e-mailing wrox-partnerwithus@wrox.com

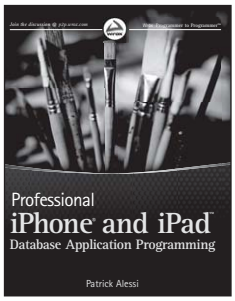
Related Wrox Books



Beginning iOS Game Development

ISBN: 978-1-118-10732-4

No matter your experience level with iOS programming, this beginner's guide covers the technologies you need to know to get started creating fun iOS games. Learning how to create games should be nearly as much fun as playing them, so this book offers a complete, playable game in nearly every chapter. Each game is created in simple, easy-to-understand parts, building to a full game by chapter's end.



Professional iPhone and iPad Database Application Programming

ISBN: 978-0-470-63617-6

As the iPhone and iPad grow in popularity, there is a growing demand for applications that are focused on data. Developers need to know how to get data onto these devices, deal with and create data, and communicate with external services—this book satisfies that need. The in-depth coverage of displaying and manipulating data, creating and managing data using Core Data, and integrating your applications using web services puts you on your way to implementing data-driven applications for the iPhone or iPad.