

Christopher Peri, Ph.D.

Contribution by Bess Ho

Includes
Twitter
Development
for iPhone
and Android

Sams **Teach Yourself**

the **Twitter API**

in **24**
Hours

SAMS

www.allitebooks.com

Christopher Peri Ph.D.

Sams **Teach Yourself**

the **Twitter API**

in **24**
Hours

SAMS

800 East 96th Street, Indianapolis, Indiana 46240 USA

www.allitebooks.com

Sams Teach Yourself the Twitter API in 24 Hours

Copyright © 2011 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33110-7

ISBN-10: 0-672-33110-1

Library of Congress Cataloging-in-Publication Data

Peri, Christopher A., 1964-

Sams teach yourself the Twitter API in 24 hours / Christopher A. Peri, Bess P. Ho.
p. cm.

Includes index.

ISBN-13: 978-0-672-33110-7 (pbk. : alk. paper)

ISBN-10: 0-672-33110-1 (pbk. : alk. paper)

1. Application program interfaces (Computer software) 2. Twitter. I. Ho, Bess P., 1967-. II. Title. III. Title. Title: Teach yourself the Twitter API in 24 hours.

QA76.76.A63P47 2011

006.7'54—dc23

2011022576

Printed in the United States of America

First Printing June 2011

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearson.com

Associate Publisher

Mark Taub

Signing Editor

Trina MacDonald

Development Editor

Songlin Qiu

Managing Editor

Kristy Hart

Project Editor

Andy Beaster

Copy Editor

Barbara Hacha

Indexer

Erika Millen

Proofreader

Sarah Kearns

Technical Editors

Doug Jones

Ronan Schwartz

Ben Schupak

Publishing Coordinator

Olivia Basegio

Cover Designer

Gary Adair

Composition

Gloria Schurick

Contents at a Glance

Preface	xiii
HOUR 1 What Is Twitter?	1
HOUR 2 Twitter Out of the Box	11
HOUR 3 Key Issues to Consider When Developing Twitter Applications	21
HOUR 4 Creating a Development Environment	33
HOUR 5 Making Your First API Call	49
HOUR 6 Building a Simple Twitter Reader	59
HOUR 7 Creating a Twitter API Framework	73
HOUR 8 Twitter OAuth	81
HOUR 9 Building a Simple Twitter Client, Part I	95
HOUR 10 Building a Simple Twitter Client, Part II	105
HOUR 11 Expanding Our Client for More API Calls	113
HOUR 12 Direct Messages	125
HOUR 13 Lists	135
HOUR 14 Favorites and User Methods	147
HOUR 15 Search	161
HOUR 16 Trends and GEO	177
HOUR 17 Friendships, Notification, Block, and Account Methods	193
HOUR 18 Twitter Documentation	205
HOUR 19 Streaming API	219
HOUR 20 FailWhale and the Future of the API	229
HOUR 21 Getting Started in Twitter Android Application	241
HOUR 22 Building Android Applications with Twitter	255
HOUR 23 Getting Started with Twitter Using iOS	279
HOUR 24 Building an iPhone and iPod Touch Application with Twitter	293
Index	319

Table of Contents

HOURL 1: What Is Twitter?	1
What Twitter Offers You	1
A Brief History of Twitter—or Why 140 Characters?	2
Summary	7
Q&A	8
HOURL 2: Twitter Out of the Box	11
What Twitter Offers You	11
Registering Your Application	15
The Twitter Client	16
Summary	18
Q&A	18
HOURL 3: Key Issues to Consider When Developing Twitter Applications	21
Types of Twitter Users	21
Types of Twitter Applications	25
Platform	30
Summary	31
Q&A	31
HOURL 4: Creating a Development Environment	33
Background of LAMP Stacks	33
Setting Up a Local Web Server	34
Securing Your Web Server	38
Development Tools	41
Summary	45
Q&A	46
HOURL 5: Making Your First API Call	49
Making a Simple Twitter API Call	49
Making a Call in PHP	53

Summary	57
Q&A	58
HOUR 6: Building a Simple Twitter Reader	59
Building Our First Twitter Client	59
Twitter HTTP Response Codes	65
Summary	69
Q&A	71
HOUR 7: Creating a Twitter API Framework	73
Twitter API Parameters	73
Creating an API Function for Twitter Function Calls	75
Summary	80
Q&A	80
HOUR 8: Twitter OAuth	81
What Is a Class and Why Do We Want to Use It?	81
What Is OAuth?	82
How to Register Your Application	82
Creating the OAuth Twitter Class	83
PHP Library for Working with Twitter's OAuth API	84
Setting Up the twitterOAuth Class	85
How to Add New Functions to Your Twitter Class Object	90
How Our Class Deals with Twitter Connection Errors	92
Summary	93
Q&A	93
HOUR 9: Building a Simple Twitter Client, Part I	95
Expanding the Index File to Support Tabs	95
Adding Support for Home Timeline	97
Adding Support for Mentions	99
Adding Support for Direct Messages	101
Summary	102
Q&A	102

Teach Yourself the Twitter API in 24 Hours

HOUR 10: Building a Simple Twitter Client, Part II	105
Updating and Adding New Files to Support Input Text Field	105
Sending a Message to Twitter	108
API Call for Direct Messages	109
Sanitizing Messages	110
Summary	110
Q&A	111
HOUR 11: Expanding Our Client for More API Calls	113
Types of API Method Calls	113
Adding Tabs to Our UI	114
New Timeline API Calls: Retweeted	117
New Status API Calls: Retweeted	119
Summary	123
Q&A	123
HOUR 12: Direct Messages	125
Sending a Direct Message	125
Adding Direct Message API Support	127
Adding More Direct Message API Support	131
The Destroy API Method	132
Summary	133
Q&A	133
HOUR 13: Lists	135
What Is a List?	135
Implementing the List API into Our Application	137
Three Types of List Methods	142
Summary	144
Q&A	144
HOUR 14: Favorites and User Methods	147
Favorites API Methods	147
User API Methods	153
Summary	158
Q&A	159

HOURL 15: Search	161
History of Twitter Search API	161
Twitter's Stance on Search	161
The Lone Search API	162
A Quick Guide to More Information on Search from the Twitter Docs	170
Summary	173
Q&A	174
HOURL 16: Trends and GEO	177
What Is a Trending Topic?	177
Supporting Trends in Our Application	177
Understanding the GEO Tag	187
Summary	190
Q&A	190
HOURL 17: Friendships, Notification, Block, and Account Methods	193
Friendships Methods	193
Notification Methods	197
Block Methods	198
Account Methods	199
Summary	202
Q&A	202
HOURL 18: Twitter Documentation	205
The Twitter Dev Website	205
Dev.twitter.com/doc	211
Twitter Resource Page Overview	212
Summary	216
Q&A	216
HOURL 19: Streaming API	219
The Three Types of Streaming APIs	219
Streaming Methods	222
Summary	226
Q&A	226

Teach Yourself the Twitter API in 24 Hours

HOURL 20: FailWhale and the Future of the API	229
What Is Spotting the FailWhale?	229
Review of the Application We Just Built	231
Where Is the Twitter API Going?	236
Summary	237
Q&A	238
HOURL 21: Getting Started in Twitter Android Application	241
Introducing Android	241
Creating the Hello Android Project	243
Summary	251
Q&A	252
HOURL 22: Building Android Applications with Twitter	255
Using Twitter OAuth in Android	255
Importing Packages	261
Summary	276
Q&A	276
HOURL 23: Getting Started with Twitter Using iOS	279
Introducing iOS	279
Creating a Hello World Application	280
Summary	289
Q&A	290
HOURL 24: Building an iPhone and iPod Touch Application with Twitter	293
Introducing Twitter xAuth	293
Benefits of Using Twitter xAuth	294
Selecting Twitter Objective-C Libraries	294
Loading xAuth Token	302
Posting Tweet	304
Adding MGTwitterEngine Delegate Methods	305
Creating Objects in Interface Builder	308
Summary	315
Q&A	316
INDEX	319

About the Author

Dr. Christopher Peri received his Doctorate from the University of California, Berkeley, in Architecture. His focus was on Collaboration in Virtual Environments delving into methods that facilitate designers and engineers to improve communication over remote networks.

He started playing with the Twitter API very early in the API release, creating his own Twitter client called TwittFilter, which is geared more to the occasional user than someone who uses Twitter all the time. As time went on, he added more and more features and functions for his own personal use, until one day he realized he had a fairly sophisticated application and opened it up to the general public to use. He learned quite a bit about the Twitter API the hard way—by simply coding things up and seeing what happens. Although TwittFilter is still a personal project, he has already created a number of private Twitter applications, robots, and smaller projects like NewsSnacker.com, which is open to the public.

About the Contributing Author

Bess Ho is a UI Engineer in mobile, tablet, TV, and web with a strong background in data analytic and consumer behavior. She received her Master Degree from the University of California, Davis in Food Science and Technology. Her focus was on Consumer Sensory Science and Engineering. She is the winner of Nokia Open Screen Project Fund and was elected as Samsung Star in the Samsung Mobile Innovator worldwide program. She served as technical editor for the book titled **Building OpenSocial Apps: A Field Guide to Working with MySpace Platform** (Addison Wesley, 2009). She has presented mobile technology at Stanford University, O'Reilly Web20 Expo SF, Where20 Conference, Silicon Valley China Wireless Conference, and many developer events. Currently, she is Mobile Architect (EIR) for Archimedes Ventures. She also advises many early-stage startups in UI/UX design and mobile development in multiple platforms. She is actively teaching many mobile classes such as iOS SDK in Silicon Valley and online courses at Udemy.com. You can follow her at Twitter @Bess or Slideshare at www.slideshare.net/bess.ho. Her developer blog is at <http://www.bess.co>.

Acknowledgments

Christopher Peri—We would like to thank all the unknown coders on the interwebs who have contributed to not only Twitter's success, but creating mountains of technical information and code examples that allows a lowly hobby programmer, like myself, to learn how to work with Twitter API and one day...write a book on it. A number of people have helped with this book, but I want to call out three people specifically: @chiah for creating the foundation of Hour 1, @jon_wu for Hour 8 as well as helping with debugging and general feedback on technical issues, and @LanceNanek for debugging and researching Android in Hour 22.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: opensource@sampublishing.com

Mail: Mark Taub
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/title/9780672331107 for convenient access to any updates, downloads, or errata that might be available for this book.

Preface

This book on the Twitter API is geared to the programmer who is just a bit past beginner—who knows the basics of LAMP, including how to set up a basic server, PHP, JavaScript, HTML, and CSS. You do not have to be an expert programmer to use this book, but you should know how to look things up. In writing this book, we have tried to provide you with everything you need to get a simple Twitter client up and running. We include an hour on setting up your environment, as well as providing you with HTML and CSS codes to have something up and running. However, it's beyond the scope of this book to explain what is happening with these codes. Instead, we focus on the code surrounding the API calls, OAuth, and the returns. That does not mean that you could not use this book if you are a beginner programmer. Because we provide you with all the code and build an application up step by step, you can stop at any time and look up parts of the code you do not understand. However, if you have never coded anything before, you may find that this book moves far too fast. It may be better to get an introductory book on basic programming in PHP before reading this book.

In writing this book, we also kept in mind experienced programmers who have been asked to create a Twitter application or include Twitter support in a current application, even if they do not know much about Twitter. We believe it's important to understand what Twitter is, how it's being used, and what makes it different from other social media services. It's with this understanding that you will be able to approach your Twitter project with a more engaged understanding of what your application is trying to accomplish, which is the best way to not only satisfy product requirements, but also design future growth.

Sams Teach Yourself Twitter API in 24 Hours is a little different from most technical books in that the book is geared around creating a functional Twitter client, including all HTML, CSS, JavaScript, and PHP needed to create your own application. We also dedicated the last four hours of this book to getting you started with making API calls on the iPhone and Android OSes in case you want to make your own mobile Twitter application.

Unlike most books, this book was written as Twitter and the API set was going through major changes. As such, the book and the code used in the book have been edited many, many times. So much so that we expect there will be a technical oversight here and there. So be sure to check the book's website for changes and updates

Teach Yourself the Twitter API in 24 Hours

(<http://www.twitterapi24.com/>). Also, as much as we tried to keep up with all the changes happening with Twitter, we fully expect some details about the various API's to evolve from the time of the last edit to the time you have this book in your hands.

We hope you enjoy this book.

HOOR 1

What Is Twitter?

What You'll Learn in This Hour:

- ▶ What is Twitter?
- ▶ List of terms
- ▶ A brief history of Twitter
- ▶ How Twitter is different from other social media tools
- ▶ Example of how Twitter has been used

What Twitter Offers You

Twitter is a vast electronic conversation that is changing personal communications through the use of new social and mobile technologies. The idea is simple: The service enables users to post messages using 140 characters or fewer, resulting in short bursts of communication that can be transmitted through text, mobile apps, or the Web. Tweets can include links to video, photos, or other media hosted elsewhere on the Internet in addition to plain text. The text link URLs are included in the 140-character limit, so short URLs are obviously preferred.

Twitter is not designed to be any one thing; it's different things to different people. For some, it's a way to talk with their friends; for others, it's a way to broadcast out to the world, a way to consume information, or a way to share links. As such, the API has been designed and continues to be designed to be as agnostic as possible to how it's used.

For people who are not familiar with Twitter, it is a platform that allows one-to-many communication. It is a mashup of text, email, instant messages (IM), news, forum posts, social networks, public conversation, links, information sharing, and the world's biggest dinner party. The technology allows for almost instantaneous communication between an individual and a self-selected group. You can receive tweets through a variety of channels: the Twitter website, IM, SMS/text message, RSS, email, or third-party applications on computers and mobile devices.

A useful way to think about Twitter is to imagine that you are IMing or texting everyone you know, at the same time, in public.

The following is a list of common Twitter terms:

- ▶ **Twitter**—The service that allows you to communicate with anyone else who also signs up.
- ▶ **Tweets**—Messages of 140 characters or fewer that are sent through the Twitter service.
- ▶ **Follower**—Someone who opts in to receive your tweets.
- ▶ **Following**—The people whose tweets you opt in to receive.
- ▶ **@reply**—A public message sent as a tweet directed at one person, designated with @username typically as a response to a previous Tweet.
- ▶ **Direct messages (DMs)**—A message of fewer than 140 characters sent privately to one of your followers. You can send DMs only to people who are following you.
- ▶ **Private account**—An account whose tweets are not public. Only people who have accounts on Twitter *and* have been approved as a follower by the owner of the account can see what has been written.
- ▶ **Trending topics**—The most popular terms on Twitter at a moment in time.
- ▶ **Retweets (RTs)**—When users find an interesting tweet and share it with their followers.
- ▶ **Hashtag**—The convention of flagging a word with the hash character #topic. This was created on Twitter to aid with keyword search and the tagging of discussions. It came from users who used IRC regularly, where #topic indicates a channel where the topic is being discussed.

A Brief History of Twitter—or Why 140 Characters?

According to Dom Sagolla (www.140characters.com/2009/01/30/how-twitter-was-born/), Jack Dorsey came up with the idea of having a mobile text message-based communication tool for groups, because their podcasting startup Odeo was struggling to find a new direction. Text messages, also known as the Short Message System (SMS) protocol, are limited to 160 characters for historical reasons. In 1985, Friedhelm Hillebrand, chairman of the nonvoices services committee in the Global

System for Mobile Communications (GSM), tested his hypothesis that 160 characters were enough to communicate a complete thought. His group of researchers pushed forward their recommendation in 1986, and the modern text message length standard was born (<http://latimesblogs.latimes.com/technology/2009/05/invented-text-messaging.html>). Twitter's character limits resulted from that 160-character limitation: 20 characters are reserved for the username, leaving 140 characters for the message.

Even though Twtr, the original name for the project, was created in 2006, the service really took off a year later at SXSW Interactive in March of 2007. Attendees used it to keep track of other conference goers, and Twitter became the hit of the show, winning the SXSW Web Award in the Blog category.

One of the big reasons for Twitter's success is that it was first built as an SMS communication platform; only later did it turn into a web-based product with simple APIs. Because a very large user base of phones already existed that can only text, Twitter was often the only way to engage in social media without a computer. Keep in mind that this was before the iPhone and Android began their run to take over the phone market.

The service continued to gain popularity and obtained massive coverage from traditional media in the November 26 Mumbai attacks, when citizens on the ground used Twitter to relay eyewitness accounts well in advance of any reporters (see www.informationweek.com/blog/main/archives/2008/11/twitter_in_cont.html;jsession-id=4JPX5T2TTQKMHQE1GHPCKHWATMY32JVN). The resulting articles and TV news reports propelled the service into the mainstream. And the leap from technologists and bloggers continued, with celebrities like Ashton Kutcher and Oprah helping to highlight the service.

During the Iran elections, the use of Twitter by the opposition was deemed so critical that the U.S. government asked Twitter to delay an update to its services out of fear of compromising one of the few channels the opposition had to communicate and organize.

During the 2010 World Cup, traffic was so high that Twitter actually shelved one of its new features in order to focus time and resources on the spike in traffic.

How Is Twitter Different from Other Social Tools?

The newest social technologies take information that was once passed from one person to another and alter the format so the sharing is faster and more public. Now the news can spread through Twitter, Facebook, reddit, and Digg, taking personal, limited-distribution conversations and disseminating them to the entire world.

Although word-of-mouth news has existed since the beginning of spoken language

itself, it now accumulates in a written record, available to a much wider audience. In addition, social technologies not only make it easy for you to share with the people you know, it also allows the people you know to share with the people they know. What used to be a phone conversation, text, or IM can now propagate to a larger audience.

Social is also different from traditional media. Older media was a one-way communication channel in which a central authority sent out information for consumption by readers. Social media technologies allow unstructured conversations to happen, so information can flow both ways or be forwarded outward to others.

Twitter is a social networking site that is simple in format but allows each person to use the service differently. Other social sites have narrower applications: bookmarking sites, such as del.icio.us, or social news sites, such as Digg or reddit, are for sharing links. Media-sharing sites like YouTube or Flickr are for distributing videos or photos. Using Twitter, you can share any combination of links, news, photos, or videos with your network.

One of the biggest differences between Twitter and other networks is that the social relationship does not have to be symmetrical. You can opt in to see updates from other people by following them, and other users can see your updates by becoming followers. In other words, when you follow people, you receive their tweets or messages, and when they follow you, they receive your tweets or messages. As noted previously, you can choose to get these messages as text messages on your phone, tweets on the Web, or as output in a third-party application.

The two biggest social network platforms are Twitter and Facebook. They are different in two ways:

- ▶ Twitter has an inherent openness, unlike Facebook, and Twitter also offers nonreciprocal relationships that are very different from Facebook.
- ▶ Facebook began as a “walled garden,” or a system that people from only certain universities or colleges could join, and it still hasn’t lost that sense of protected information.

Most of the photos, status updates, or other content that you post on Facebook is accessible only to people who are connected to you. In contrast, you can always see users’ tweets on their Twitter page or at the URL www.twitter.com/username, provided they have not made their account private. This is further reflected in the limits of what interactions from an API perspective are supported.

In the beginning, the power of Twitter was in the conversations and the networks of people who choose to participate. Twitter is closer to the old IRC channels than to any other form of communication. This means that for a user, the service is not useful without a meaningful social network, and so it's hard for a new user to understand what to do with it. However, as the acceptance of Twitter by mainstream media has grown, more and more users are finding Twitter as a great information resource for news, Hollywood rumors, stock tips, and general yelling—mostly during sporting events. Twitter has become so useful for gaining information that often a story will break on Twitter before making it onto traditional media.

Like most things on the Web, after a tweet is sent out, there is no way to edit its content; the only thing that can be changed is that the tweet can be deleted. And even in that case, if the tweet went out to mobile devices or third-party tools, those copies are not deleted. So, as with anything information you put out to the Internet, if you would not say it in public, don't say it on Twitter.

Twitter Use Case Study: #blamedrewscancer

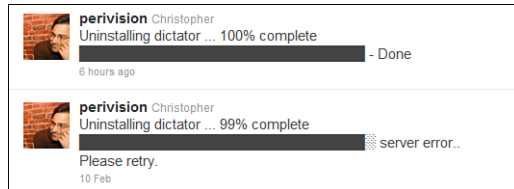
On May 20, 2009, Drew Olanoff was diagnosed with Hodgkin's Lymphoma (www.drewolanoff.com/post/117383549/thats-not-what-i-ordered), and he decided to use his online presence to create awareness of his cancer. He chose to write a blog post and use Twitter to share his experience. To make it interesting, he created a hashtag, #blamedrewscancer, and encouraged his friends to blame whatever went wrong in their lives on his cancer.

Soon, hundreds of people were tweeting about lost keys, getting stuck in traffic, Mondays, and anything else going wrong, all using this tag. A website was created that showed the tweets in a fun way, and news outlets started picking up the story. In just 100 days, more than 11,000 people blamed more than 25,000 things on Drew's cancer (www.twitip.com/blamedrewscancer-for-this-case-study/). What started out as a personal story of a cancer diagnosis became a phenomenon on Twitter. People connected over their own stories of unfortunate experiences.

Twitter Use Case Study: Global Politics

On February 11th, 2011, Mubarak stepped down from his post of President that he held since 1981. With the military taking over power, it seemed almost all of Egypt erupted in celebration. Almost as soon as his plane reached cruising altitude, the news broke and Twitter went nuts. Here is a Tweet I sent out the day before where he gave a speech where everyone expected him to step down and then a Tweet after he did the following day (see Figure 1.1).

FIGURE 1.1
Tweets sent during the Egyptian revolution.



This is not the first time that Twitter, Facebook, and other social media services have had an influence on world events. If you remember, back to April 10th, 2008, a UC Berkeley student sent out a single tweet that saved him from an uncertain outcome. He tweeted the word “Arrested”...just as he was taken into custody. That single Tweet was enough to let people know in Egypt, and back in the U.S., what had happened; to hire a lawyer and to demand his release. Although even back then, Twitter had already proven itself as a medium for rapid dissemination of information unlike anything we have seen in the past; no one could have foreseen the impacts yet to come.

Fast forward to the beginning of 2011. The number of people on Twitter, Facebook, and other social media climbed to the hundred of millions. Twitter and Facebook alone, combined, claim just under one billion users. Combine those numbers along with the explosion of online mobile devices now capable of accessing these services and you have a flattening of communications never before seen since the advent of the printing press, the consumer grade photocopying machine, and email. Each of these revolutions in communication has had its impact on society; the Twitter revolution is no different.

The reach of social media, especially Twitter (since it supports communication with increasingly popular text messaging), has become so prevalent that the normal tools used by regimes to manage their population have become compromised. Usage of information is a tool; information control is paramount to controlling a population. The more control over information you can impress, the greater the likelihood the population will believe and act on whatever information you provide; or conversely, ensure it never gets disseminated in the first place. Just in the past year alone (2010-11), we have seen exceptional examples of states that had some form of control over information (typically by controlling the press), but lost that control over information because of networked communications like Twitter and Facebook. Even with efforts to shut down Twitter and other social media platforms, information still seems to find a way out. For example; in Egypt, access to Twitter was blocked. In 24 hours, it was announced on the Google Blog, that the search giant has teamed up with the SayNow team and Twitter to create a simple speak-to-tweet service for people currently engulfed in the turmoil in Egypt. From the Google post...

“It’s already live and anyone can tweet by simply leaving a voicemail on one of these international phone numbers (+16504194196 or +390662207294 or +97316199855) and the service will instantly tweet the message using the hashtag #egypt. No Internet connection is required. People can listen to the messages by dialing the same phone numbers or going to twitter.com/speak2tweet.”

We hope that this will go some way to helping people in Egypt stay connected at this very difficult time. Our thoughts are with everyone there.

At the time of this writing (early 2011), demonstrators have clashed with police in the Yemeni capital Sanaa, riot police in Algiers dispersed thousands of people who had defied a government ban to demand that President Abdelaziz Bouteflika step down, and President Mahmoud Abbas will immediately ask Prime Minister Salam Fayyad to appoint a new cabinet. And in Iran, reports say several opposition activists have been arrested and international broadcasters are being jammed. In Libya, the control of the country is currently in doubt and sections of the country are no longer in government control.

As much as it seems that the “tools” of social media was the foundation of the revolutions we have been talking about, and those that seem to be coming, it’s not the service of Twitter, Facebook, YouTube, and Google but instead the change of thinking that these tools have helped evolve. By allowing people to exchange ideas and information quickly and easily and with greater reach, social media tools have given people a sense of community and strength. And it’s this ability to create and inform communities through social media that is the real power of Twitter, not just sending 140 characters.

Summary

This hour introduced you to Twitter, gave a brief history of the service, covered the basics of social media, and described how Twitter is different from other social platforms. The common terms used on Twitter were defined, and you should now have an understanding of the functionality of the platform and some of the ways people use the medium for communication. We also discussed an example of how someone used Twitter to create a community and illustrated some of the social norms at play and reflected on how such a simple idea like Twitter and all the programmers the help made it grow can have an effect on world.

Q&A

Q. What is an @ reply?

A. It's a way to specify a username on Twitter. Typically, this is used to respond to a tweet created by the user referenced.

Q. What is the character limit for a tweet?

A. 140 characters.

Q. What is a hashtag and why are they important?

A. Hashtags are a way to indicate a keyword by putting # in front of it. They are important because it allows people to tag tweets, search for them, and also organize all tweets from an event or chat. Think of it as a way to indicate the subject or subjects of a tweet.

Q. Do I have to already have a network of friends on Twitter before I begin to find the service useful?

A. No, many Twitter users send no more than a handful of messages a month. More people read messages on Twitter than create them. There are services that are focused on presenting Twitter messages (and the content of their links) as stand-alone application for reading only.

Workshop

Quiz

1. Why is there a character limit in a tweet?
 - A. Twitter decided that's long enough for a thought.
 - B. Twitter wanted to save on server space.
 - C. There is a hard-character limit on SMS.
2. True or False: There are two types of accounts on Twitter: one that is open and another that is closed.
3. What is a direct message, or DM?
 - A. A tweet that doesn't go through Twitter's servers.
 - B. A private tweet that goes only to the person you are sending it to.
 - C. A message that comes from Twitter corporate.

Quiz Answers

1. C. Twitter started off as a text or SMS system, and mobile phones can accept only 160 characters; 20 are reserved by Twitter for the username.
2. True. There are private accounts that are not open to anyone who doesn't have permission to follow.
3. B. A direct message is not shown in the public timeline and goes only to the person you are sending it to. You can send it only to someone who is following you.

Exercises

1. Visit www.twitter.com and create an account. Then follow a few of the suggested users.
2. Use search.twitter.com to find keywords that are interesting.

This page intentionally left blank

HOURL 2

Twitter Out of the Box

What You'll Learn in This Hour:

- ▶ What Twitter offers you
- ▶ Creating a new account
- ▶ Skinning your account
- ▶ Registering your application

What Twitter Offers You

Unlike almost any other API, Twitter exposes almost everything. Basically, if it's on Twitter, there is an API for it. That includes account setup and customization information. This is one of the great approaches of Twitter—focusing on the network and allowing others to create applications on top, and you have that right out of the box. After you set up your account, you are ready to go with almost every function that Twitter offers. There are limits, of course. Here are the current limits from (<http://dev.twitter.com/pages/rate-limiting>):

1,000 total updates per day, on any and all devices (web, mobile web, phone, API, and so on)

250 total direct messages per day, on any and all devices

150 API requests per hour

OAuth calls are permitted 350 requests per hour

Whitelisting

Here is the current policy from Twitter.com on whitelisting.

(http://groups.google.com/group/twitter-development-talk/browse_thread/thread/1acd954f8a04fa84?pli=1)

**Watch
Out!**

There is no general idea of a whitelist for the Search API as with the REST API. However, under extraordinary circumstances, Twitter will work with developers to raise rate limiting for Search requests.

“Beginning in February 2011, Twitter no longer grants whitelisting requests. We will continue to allow whitelisting privileges for previously approved applications; however, any unanswered requests recently submitted to Twitter will not be granted whitelist access.

Twitter whitelisting was originally created as a way to allow developers to request large amounts of data through the REST API. It provided developers with an increase from 150 to 20,000 requests per hour, at a time when the API had few bulk request options and the Streaming API was not yet available.

With authentication, an application can make 350 GET requests on a user’s behalf every hour. This means that for every user of your service, you can request their timelines, followers, friends, lists, and saved searches up to 350 times per hour. Actions such as Tweeting, Favoriting, Retweeting, and Following do not count toward this 350 limit. Using authentication on every request is recommended, so that you are not affected by other developers who share an IP address with you.”

Setting Up Your Account

Odds are that you have already done this. However, in an effort to be complete, we are going to briefly walk through setting up and configuring a new account.

Open up a web browser and go to www.twitter.com; then click on ‘Sign Up’.

Account Information (shown in Figure 2.1) is pretty clear; thus, I will not go over every field. Keep in mind that these fields, with exception to email, are most likely not checked for proper format. For example, you can see that I used “the bay” for my location. Although there are plenty of other sites with the name of the city I live in, back when I first set up my account, I was being a little more cautious. Yes, there was a time when Twitter was yet just another startup. What is great about Twitter (and now other services are seeing the value of this) is the fact that your username is unique and part of your Twitter URL. In this case, my unique Twitter username is Perivision. Thus, my unique Twitter address is <http://twitter.com/perivision>.

**Did You
Know?**

Many system admins will set up an account using a user’s name and adding a 1 or a 123 after it. For example, consider the username: myusername123. This is common practice, and hackers look for it. Do not set up an account for a new user using this technique. Many systems now create a completely random string of letters and numbers and then email the password. This is a more secure procedure.



FIGURE 2.1
Example of the Twitter setup page.

We want to reinforce that you use a somewhat cryptic password when you get to this screen (see Figure 2.2). However, if you are setting up accounts for other users and use something simple for them with the expectation they will change it, double-check that they did change the password.

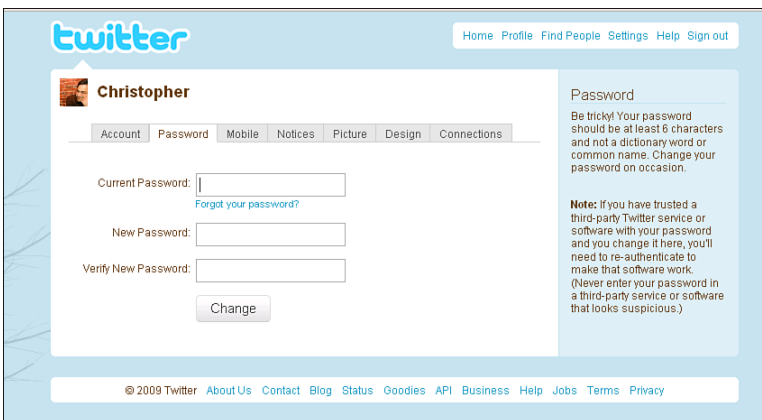


FIGURE 2.2
Screenshot of the password page.

Did you Know?

Many people believe that one reason for Twitter’s popularity is that usernames are unique and, therefore, the vanity address is unique. At the time of this writing, many other services, such as Facebook, LinkedIn, and Google, have moved to “vanity” URLs.

Twitter first started out as a text-messaging system only. Although most of the interactions with Twitter are through the Web, text messaging is still an option (see Figure 2.3). Be careful, though, if you do not have unlimited SMS messaging with your plan; it can get out of control, and thus very expensive, very quickly.

Although you can get New Follower email alerts and Direct Message email alerts, you can no longer get email alerts for mentions. There are third-party services that can do this, however.

FIGURE 2.3 Screenshot of the mobile options on Twitter.



Did You Know?

In the screenshot shown in Figure 2.4, the text “Direct Text Emails” is used to refer to “direct message.” This terminology is a holdover from when Twitter started as an SMS service.

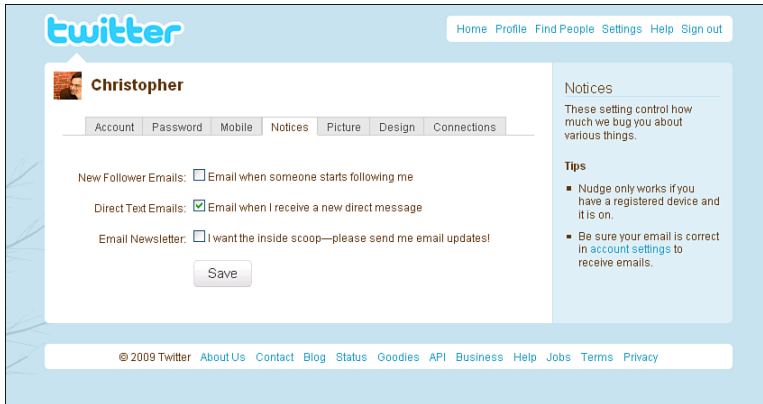


FIGURE 2.4
Notices screen
in Twitter.com.

Registering Your Application

With Twitter, you can register your application at (<http://twitter.com/apps/net>) so a user can share OAuth credentials with you (we will cover this in later hours of this book) and that when someone gets a tweet, the person can see what application it came from. If you are going to create any type of Twitter client that can send messages, it's worth your time to set this up. Wait until you have a beta version of your site alive and running. You cannot register a nonfunctioning site. In addition, it is possible that Twitter may review your site for promotion on Twitter.com. So, make sure your beta is working well.

Twitter originally allowed a username-password combination for registering a new application, but no longer. The following is from the Twitter site:

“We originally allowed applications to create a source parameter for non-OAuth use but that has been discontinued. Applications pre-OAuth source parameters will remain active, but new registrations are no longer accepted.”

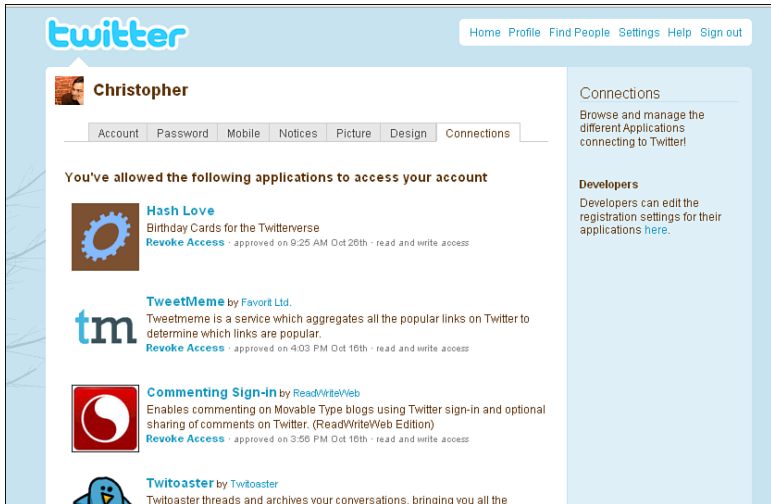
The registration process is simple enough: Provide the name of your application, OAuth information (<http://oauth.net/about/>), a short description, and a logo, and you are ready to go. As you can see in the screenshot (see Figure 2.5), I registered TwittFilter. I can, however, register more than one application if I choose.

You can also authorize other applications to have access to your account. Under the Settings and then Connections tab, you can see all applications you have authorized. It's a good idea to keep an eye on this for your own account as well as accounts you manage. Figure 2.6 is an example of applications registered to have access to the Perivision account.

FIGURE 2.5
Example of a registered application.



FIGURE 2.6
List of applications that have access to the Perivision Twitter account.



The Twitter Client

The default page of Twitter.com has changed a few times over the years, so what this page will look like by the time of this printing is unknown. However, as you can see from the screenshot shown in Figure 2.7, the folks at Twitter seem to be committed to making search and topic trending a major part of Twitter's offering. As such, when you are developing your application, understanding this direction is important so that you do not develop something that later becomes a native functionality within Twitter.

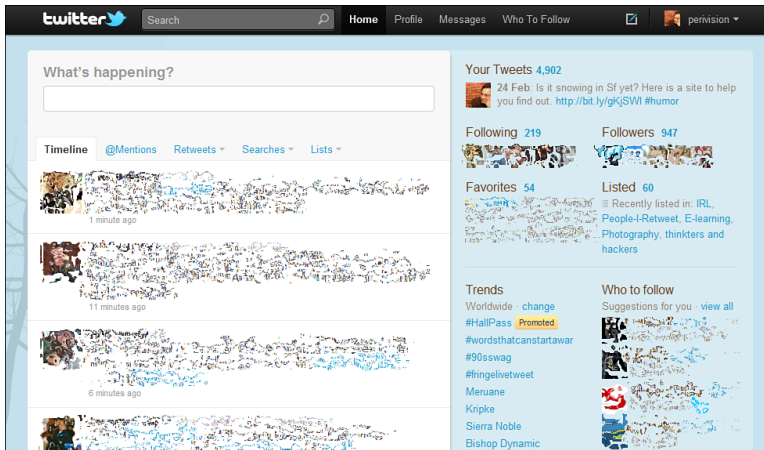


FIGURE 2.7
Example of the
Twitter home
page as of
early 2011.

The web client that Twitter provides is bare bones and purposely so. However, most of the features and functions you need to fully engage with Twitter are here: reading your timeline, mentions, direct messages; managing your following and followers; as well as managing the recently added lists. Although the Twitter.com website is “bare bones,” it serves as a great example of the minimum functionality a Twitter user would expect from a client, which is the following:

- ▶ Create a new tweet
- ▶ Create a new direct message
- ▶ Read your latest messages from your timeline
- ▶ Read your latest mentions
- ▶ Read your latest direct messages
- ▶ Read your lists
- ▶ Respond, reply, and retweet messages
- ▶ Reply to a direct message (which is different from replying to a public or mention)
- ▶ Search Twitter
- ▶ Edit your lists
- ▶ Follow/unfollow a tweeter
- ▶ Block a tweeter
- ▶ Create or remove Favorites

Notice that I did not include in the list anything about setting up and managing your account. Although this is becoming more common in mature Twitter applications, it is still not considered a basic feature. This is because it is assumed you can manage your account via [Twitter.com](https://twitter.com).

There is also a form of convention on how to display a message, which you can see in Figure 2.7. Typically, the Twitter image of the person who sent the message is displayed on the left, and the message box is normally wide enough to display three lines, thus lining up nicely with the image. The date and source is normally displayed in a smaller font as the fourth line. You may also notice that certain words are colored blue, indicating they are hyperlinked. The current Twitter convention is to provide a link to any Twitter user's account that is found within the post, any term with a # as the first character, and any term with `http://` as the lead characters.

Summary

Good job. You should have your account set up. You should also have a good idea of what basic features and functions your client or customers may expect if you decide to create a Twitter client. Although it's not necessary, skinning your Twitter page is a worthwhile exercise.

Q&A

Q. *What is a vanity URL?*

A. A vanity URL is a URL unique to a user that employs the username in the URL structure.

Q. *We know that there is a 150 API call limit. What are the other two limits at this time?*

A. 1,000 total updates per day, on any and all devices (web, mobile web, phone, API, and so on) and 250 total direct messages per day, on any and all devices.

Q. *Can I use my login and password when I register my application?*

A. No, you need to have information from setting up OAuth within your application, including the URL that twitter will use to verify your application.

Workshop

Quiz

1. True or False: Twitter's main objective is to build a full-featured super client application.
2. True or False: It is never a good idea to create passwords that are easy to guess, even if you expect it to be changed later.

Quiz Answers

1. False. Twitter wants people to build interesting products on their service. The goal with twitter.com is to focus on an enjoyable experience.
2. True. Bots exist that try to guess common passwords.

Exercises

1. If you have not done so already, set up a Twitter account.
2. Take a look at various Twitter page designs. Can you figure out how they were done?
3. If you have an idea of what type of Twitter application or widget you want to build, what percentage of its features and functions are already supported on the Twitter.com site?

This page intentionally left blank

HOUR 3

Key Issues to Consider When Developing Twitter Applications

What You'll Learn in This Hour:

- ▶ Different types of Twitter users and how they impact code design
- ▶ Different types of Twitter applications and program architecture
- ▶ Things to consider if you are not building a web-based application

Types of Twitter Users

As one would expect with an API system as open as Twitter, and the explosion of interesting applications people have developed, we have also seen the development of different types of Twitter users. Understanding these types of users and knowing which of them we are trying to reach will inform how we may want to build our Twitter application framework. As with any large user base, there are a number of ways to set up categories. In this hour, we will break down and discuss the users in the following categories or types.

The News Reader

Twitter is a great source of breaking news, whether it's politics, business, sports, or following celebrities. Most users use searches to find what they are interested in, or they follow Twitter feeds that act like RSS readers. For example, BreakingNews is what you would guess it would be—a Twitter account publishing breaking news. Most news outlets have such accounts: CBSNews, ABC, BBC, and so on. The screenshot of NewsSnacker, an application created by the author (shown in Figure 3.1) is a good example of a Twitter application that focuses on the news.

FIGURE 3.1
Screenshot of
NewsSnacker.



Although making search and Twitter account API reads from Twitter does not require authentication, you can still get dinged going over the API limit because Twitter will limit calls from an IP address. So, you still need to keep in mind how often you make calls. In the case of NewsSnacker, we use a white-listed account because the user could exceed the API calls-per-hour limit since each news service is a separate call. Suppose that the user has 10 sources and refreshes every 30 minutes. That is 200 calls in an hour, which is over the current limit of 150 for non logged in users. This does not include normal calls to check for new mentions or direct messages from the user's chosen Twitter client application. An alternative approach is to create a list of twitter news accounts and then call that list. However since newsSnacker removes duplicate posts, a large number of returns on the list call would be required. Both approaches have their merits however; one feature of newsSnacker is to allow a custom list of sources. This can be done by having the user log into the application and then select which of their lists they would like to call thus the second approach is being pursued in the next version of the application.

Chatters

Twitter does allow for people to have conversations; it's called a *direct message*. However, many people like to hold their conversations in public and a big attraction for these people is conversation threading. This is a very complicated proposition, so much so that new APIs are being created to deal with this situation. We will cover

retweeting in later hours, but this could cause quite an impact on your code's structure because of older reply techniques that use the letters RT for conversations instead of recent API methods that support replies formally. So, supporting Twitter conversation is a decision you will want to make early in your product's design.

Power Users and PR Managers

Although you will have a drag-out fight between the two because one is personal messaging and the other is more professional, the impact on product design is not that much different. PR (public relations) managers, power users, and anyone who consumes or monitors a lot of Twitter information will put special requirements on you as a product developer. Like the limits to the number of API calls mentioned in the section discussing the user group news readers, the issues with the power users and PR managers group will be the same, with the added requirements of being able to sort and search the stream of messages that come in. They may also need to send messages on a schedule or from people using the same account. There is usually no simple way around this issue other than to start thinking of a well laid-out database up front. You may want to also explore having your server make the API calls and relay the information to your Twitter application in the form of automatic processes or bots. Furthermore, set up your architecture to deal with a wide variety of API calls. We will cover this later in the hour. PR managers will want more than just searching the Twitter stream; they will want to make sense of it and make sense of who is on that stream and their influence. The API has just expanded to handle retweets, but not all Twitter clients will be updated to work with this API. As such, you still need to pay attention to RT (the current convention for a retweet) and hashtags. Plan for this up front. Also, plan to keep some of the user information in your database; you will want to use it for user profile and relationship analysis. Although the number of power users, compared to typical Twitter users, is quite low, having a power user using (and advocating) your application is highly desirable, and although every power user you talk to will have a different list of features and functions, there are some things you must be able to support—for example, dynamic search. Just providing a call and return to the search API is not good enough anymore. The current and future power users of Twitter are going to demand just as much power and feedback as they get using Google search. For example, power users would want links with the tweets that are returned to be followed and analyzed in some manner. Perhaps you should show a thumbnail of the site, or display the title and the first 50 words of the link. Be sensitive to nonstandard protocols, such as searching stock quotes using the \$ sign in front of the stock market ID. For example, \$aapl for Apple. Power users are going to demand speed and customization and will fully expect that your application understand the nonstandard features (social conventions) of Twitter.

Microbloggers

Microbloggers will want to take the time to craft each tweet carefully. Pay attention to the ease of creating a message—that is, allowing them to save as drafts, sending to multiple Twitter accounts, spell checking (yes, spell checking), and although this is not easy, a quick look up of the other tweeters or access to a list of tweeters. Especially for PR users, you may want to have a look at simple web-based CRM products to give you ideas. A new API to Twitter is the capability to store lists of tweeters. This is useful to all power users as well as microbloggers.

High-Frequency Users (TwitterHolics)

The current rules of the API system allow only 350 calls per hour if you are logged in, 150 if not. This may seem like a lot, but based on what features you are providing to your users, this can go very quickly. It's not unlikely that you could have five API calls per user action if you need to make follow-up calls. If they are high-frequency users, they may find themselves approaching the 350-call limit pretty quickly. Although there are calls that do not require credentials, you could still run up against this limit because Twitter does count the number of calls from an IP. As such, be sure you monitor the number of calls the user has left and deal with it accordingly. The good news is that an API call exists for checking how many API calls the user has left which does not count against your API limit. However, calling it over and over again too often (every 5 seconds, for example) could trigger other traffic limit controls.

New Users

This is less an API architecture question than a GUI issue. Although GUI design is not addressed directly in this book, consider using clear terms and common metaphors (like an email system, for example) for the layout and functionality of your application. Do not assume that your users will understand various social conventions in Twitter, so explain it up front and design your functions' intent clearly using tool tips for icons for example. If you are making an application that reflects some aspects of the Twitter.com site, be sure to follow the conventions Twitter uses.

Bots

Bots (programs that perform automated tasks), including creating spam or setting up phishing attacks, will always be an issue. A sophisticated Twitter application will be aware of some of these bots and try to protect users. You may, however, need to

create your own bots (for good, not evil). For example, you might take a RSS feed and republish it to Twitter after passing it through a business rules filter which is something the main Author of this book does. Because a bot is nothing more than “rules” you have for dealing with reading or creating Twitter messages or lists, you will find creating automated processes very easy with the Twitter API.

Types of Twitter Applications

Normally, when I’m about to start writing a Twitter application, I already know what I want it to do. Thus, based on the features and functions I have in mind, I already know what platform and category of users I’m targeting. Because we cannot know what you, the reader, have in mind, we will try to set up a basic framework for thinking about the various things you can do with Twitter as we go through this book. Part of Twitter’s success is its simplicity and wide-open API. As such, people have developed powerful, sophisticated applications, mashups, and simple widgets that run in other apps or on web pages. However, the approach you will take building a full-on application is different from building a simple mashup or widget.

A mashup is a web page or application that takes two or more data sources and combines them into a new service. Typically, mashups create a functionality not envisioned by the creators of the original sources. Twitter is a very popular mashup source.

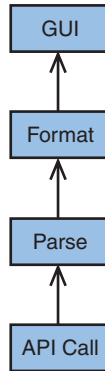
**By the
Way**

Building a feature-rich Twitter application takes some planning. Although we will walk you through various examples of how to build apps around specific APIs, we want to bring focus, too. There is an overall approach you should determine before you write line one.

Widget

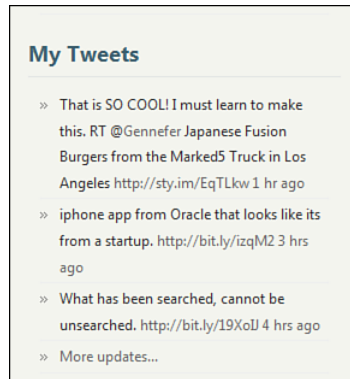
Let’s talk about architecture around a simple widget. Suppose our simple widget is going to display the results of a search or the latest tweets from a user. This is the easiest to build. All we have to think about is four steps: make an API call to Twitter, parse the return, format it, and display it. That’s it. We diagrammed this simple architecture in Figure 3.2.

FIGURE 3.2
Example of a simple Twitter API diagram.



All API systems work this way, but what’s great about Twitter is that the results are already of value. Quite often, blogging sites (mostly personal) have this type of widget. I have a widget like this on my blog (see Figure 3.3).

FIGURE 3.3
Screenshot of a Twitter widget on www.perivision.net/wordpress.



Mashup

Because a mashup can be the combination of anything, and that’s kind of the point of mashups, we are going to think about our architecture a bit differently. Although technically, a mashup can be just two sources of information or very complex number and relationship of sources, we are going to stick with the spirit of what is considered a mashup by just thinking about mixing two data sources. For example, we can take our Twitter search feed and weather data and display tweets from places that are raining versus tweets from where it’s sunny. In this case, we need to store our returns from Twitter somewhere while we get weather data. Then we need to perform some business logic on those returns.

Business logic is a nontechnical term generally used to describe the functional algorithms that handle information exchange between a database and a user interface. It is distinguished from input/output data validation and product logic.
From Wikipedia, the free encyclopedia

In this case, we need to hold our returns in an array so that when we get the weather data, we can reorganize our data. Because tweets are small, discrete messages, it makes sense to create a multidimensional array object that we can easily explore. So now, we will add one more layer to our diagram. As you can see in Figure 3.4, we are using arrays to store our parsed return so that we can apply some rules (business logic) to create a more valuable dataset.

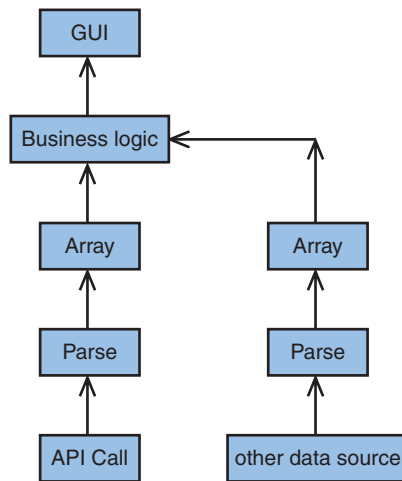


FIGURE 3.4
Example of
combining two
data streams.

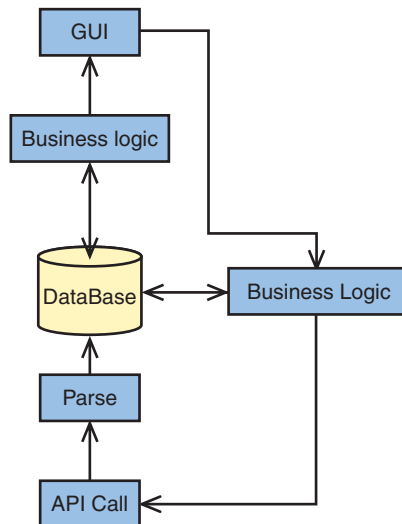
Twitter Application

I would expect that only a small percentage of readers of this book are intending to build a full-featured Twitter client, but if you are, you want to approach building your application like any other application. Think about your calls to Twitter almost like calls to a database where you provide a set of parameters with your call and get a filtered response that can then be analyzed or applied to a set of rules. It is also well worth your time to set up your Twitter calls in a separate class to deal with errors and changes to the API. You should also set up another class to deal with converting your Twitter calls into multidimensional arrays and/or storing them in a database. The reason for this is that Twitter is still changing. Even during the writing of this book, we had to make adjustments to the book's index as new methods were introduced and other calls were deprecated. By keeping these two processes in standalone classes, you're going to save yourself some headaches down the road. If you are

planning on building a full-scale Twitter app, we recommend bookmarking the website for this book and the Twitter API site. Really! It changes and grows that much.

Also, somewhat like a database, you can store information in Twitter. For example, a much-overlooked feature is favorites. This API call allows you to save tweets. This can be quite useful as a means of understanding which tweeters and types of tweets a user tends to favor. New to the API list is lists. This is a list of tweeters a user creates. Again, it's a powerful bit of information that can be quite useful in understanding users' preferences. What is more interesting, though, is using these two API calls as storage devices if your user is under your control—a corporate account, for example. Because the user of that account does not interact with the account personally, you can use these API calls to store tweets and lists that can have greater meaning than originally intended. For example, suppose you have a corporate account for company X. We can store in the favorite list all tweets that match a certain rule, like any tweet that has an unfavorable term in the tweet. Now you have a list of tweets that public relations can examine using other than the application you developed. Also, remember you have access to user bio, location, and other elements. Again, because of how open Twitter is with its API, you can use these fields for anything—for example, including the updating of the Twitter background based on the latest message from the company, or perhaps updating the location field if you're a mobile food van, or changing the profile image based on the time of day or your mood. Instant database functionality ... of sorts! Now this does not mean you should not have a database if you intended on storing anything beyond the simple examples provided here. Also, it is not recommended to abuse this open access by placing unrelated data in these fields. Most applications will follow a simple structure, as illustrated in Figure 3.5.

FIGURE 3.5
Example of an architecture placing a database between API calls and an application's business logic.



Pure Chat

This class of Twitter application is concerned with creating tweets, reading incoming tweets, searching Twitter, retweeting, setting/getting favorites, and displaying simple user account information. Everything can be done as a standalone command, meaning you do not need to store information outside of Twitter. Each command has only one or two API calls. The current Twitter.com main web page is this type of application. Since we do not need to keep track of a state or store data, we can create this application using nothing more than a simple collection of PHP calls. For this class of application, we want to think about our application as a series of standalone pages. It would be a good idea to use cookies on the user's computer in case you need to store last-seen dates or other simple pieces of information.

Structured Display

Very common with Twitter applications are the capabilities to save groups, perform more advanced searches, display only new information and some threaded conversations, and so on. Although some of these structured displays can be somewhat complex, the approach you would take as a programmer is not that much different. Many of these structured displays can be achieved without storing information on the server but by using API calls and cookies instead. Consider the following example: Suppose we want to display a column of unread tweets, tweets from our “top 10” friends, three or four saved searches, and your current favorites lists. All of these can be achieved by passing variables within the existing API calls. You will actually work far harder at the UI than the backend coding. For this class of application, we want to set up our code as a series of calls that are more or less self-contained. This will make dealing with the GUI less troublesome as redesign is requested or required.

Twitter Statistics

Collecting statistics from Twitter data provides great promise for research, improved discover and communications. However, this class of Twitter application is a bit harder. TwittFilter, another application created by this author is in this class as shown in Figure 3.6.

This class of Twitter application depends on creating new information through analyzing the return or returns from past Twitter calls, and storing or modifying this information on the server, typically in a database. This, however, is where we find Twitter to be the most interesting; because Twitter has such a large user base, you can gather enough data to infer information that does not have a direct user correlation. Did someone say mashup? For example, a very popular and now API-supported feature called “Trends” in Twitter is nothing more than a constant search

across all messages being sent to Twitter, displaying the terms with the highest rate of occurrence. However, because of the large user base and ease of creating tweets, Trends tends to be one of the first places that news breaks.

FIGURE 3.6
User Scoring
screen of
TwittFilter.



Because we need to store information as well as grab details for analysis, we need to think about how we structure our program differently. For this class of application, we want to think of Twitter as more of a database source. Setting up our arrays that allow for ease of use within formulas, as well as pulling and pushing into databases, will be a great benefit as our analytics become more and more complex. However, if you are not white listed, you will run into the API call limit quite quickly. It's recommended that if you plan to do statics that require large sample sets or recursive calls, that you explore the streaming API.

Platform

Now that we know the class of application we want to develop, we need to think about the delivery platform. If you are going to develop for UI-hosted apps, such as native mobile apps or Adobe Air, you may again want to modify how to approach your coding. Typically, when creating an app for the iPhone or other mobile platform, many of our UI elements are going to be handled on the device. You also will want to minimize the amount of traffic going back and forth as much as possible. Therefore, you should design your application around the output, which could be XML, JSON, or some custom bitcode. In this case, the organizational structure you choose will dictate how you structure your backend code. Because we have the luxury of storing information on our target platform, we can focus on speed and ease of architecture. Even though our backend code is not responsible for the presentation

layer (display), you still need to follow the basic tenants of good programming design by keeping the business logic separate from the API calls; don't fall into the trap of making each call from your application as a separate instance, as you may with the Pure Chat approach. You never know when your application starts to take on more features than you planned.

Summary

In this hour, you were introduced to various types of Twitter users. Depending on your product's target market, you may need to think about how you will approach the design architecture of your product.

This was not intended to be an exhaustive list, nor an absolute one. One could easily break this list into smaller pieces or roll it up into more general categories; instead, it's to provide a framework to think about the application you intend to build. We broke this up into two sections because we want to make a distinction between the type of use and type of application. However, do not think you can explore one without the other. When designing any application, you should always start with the user. What is the value proposition you are offering users in order for them to use your application? Once you understand that, you can then move to the type of application you want to create. So, we started out with an exploration of types of Twitter users, and then types of Twitter applications. We ended this hour with a short conversation about platforms. If you are developing for anything other than the desktop, you are most likely already aware of these points, but we included them for less-experienced developers as good to know.

Now, hold on to your hats because in the following hours, we are going to start building code!

Q&A

Q. *Should I apply for a white-list account before I start coding?*

A. No. White listed accounts are currently not available. However, you will find that having 350 calls per hour is plenty as you learn how to develop your program.

Q. *I plan to make a simple Twitter application now, but I may expand it later. Should I bother setting up a separate twitterAPI class?*

A. Yes. If you have any plans, even just thoughts of doing something beyond a few different types of API calls, set up a separate class for your API calls. In addition to new APIs, current API calls can change.

Workshop

Quiz

1. What is meant by thinking about Twitter as a type of database?
2. I check my Twitter account only a few days a week on my iPhone. What kind of Twitter user am I?
3. Is it illegal to create bots?
4. What is the easiest type of Twitter application to create?

Quiz Answers

1. This is a two-part answer: 1) Although Twitter exposes everything, you still can only get detail data on users one at a time although this is changing. Thus, thinking about accessing user statistics as if you were accessing a database is a useful way to think about what you can do with Twitter. 2) If you have control over the Twitter account(s), you can use the fields in Twitter to store information instead of on your database.
2. You are a news reader. Even if you are reading only your timeline (people you follow), you are more of a consumer of information than a creator.
3. No—and not all bots are bad. However, the good folks at Twitter do actively look for automated processes that abuse the system.
4. A pure chat widget.

Exercises

1. Describe your typical target user and then determine the class of application you feel is appropriate for your user.
2. If you plan to create an automated process, write down each step and then count the number of times you will need to call Twitter to get information. What happens if the user hits refresh 10 times in 10 minutes? Will you go over the 150-API call limit if they are not logged in?

HOUR 4

Creating a Development Environment

What You'll Learn in This Hour:

- ▶ What is a LAMP stack?
- ▶ Setting up a local web server
- ▶ How to secure your web server
- ▶ How to choose the right development tools

Background of LAMP Stacks

If you've ever built a dynamic web application, it's no news to you that you need a web server to run your code; you can't just open files straight from Firefox like static HTML files. Most of the code we write in this book is in PHP, so you'll need an Apache web server with PHP installed to run the examples. If you already have a development environment in place, you can skip this hour.

So, before we get started, what is LAMP? From Wikipedia:

LAMP is an acronym for a solution stack of free, open source software, originally coined from the first letters of Linux (operating system), Apache HTTP Server, MySQL (database software), and PHP, Python, or Perl (scripting language), principal components to build a viable general purpose web server.

Although the "P" in LAMP stands for PHP, Python, or Perl, it most commonly refers to PHP.

PHP originally stood for Personal Home Page.

***Did You
Know?***

LAMP stack packages have become popular because configuring Apache, PHP, MySQL, and all the necessary components is no easy task. They offer developers a quick and easy way to get a web server running with everything they need on their local machines. Even if you have an existing web host with everything you need, it is often faster and more convenient to develop code locally.

Although there are entire books about LAMP, this hour will serve as a quick intro or refresher so you can follow along for the rest of the book, even if you're new to PHP. Before we set up our own server, here's an overview of each component:

- ▶ **Apache**—The most popular HTTP server on the Web since 1996; it currently serves the majority of sites on the Web.
- ▶ **MySQL**—A relational database management system (RDBMS) used for persistent storage in many applications.
- ▶ **PHP: Hypertext Processor**—A popular general-purpose scripting language generally used to create dynamic web applications often running on top of Apache and using a MySQL database for persistent storage.

Did You Know?

*AMP is a term most commonly denoting the use of Apache, MySQL, and PHP regardless of the operating system. There are variants such as LAMP, WAMP, and MAMP for Linux, Windows, and Mac, respectively. All components are available on all major operating systems, although the majority of web servers running Apache use Linux.

Setting Up a Local Web Server

LAMP is a common term describing a web server using Apache, MySQL, and PHP, but there are a number of ways to set up a LAMP server. Although you could install Apache with PHP and a MySQL server independently, getting all the right components set up and working together can be a tricky process. The easiest way to get a web server up and running is to use one of many LAMP packages available, which will help you install everything you need to start testing your PHP code.

Running a local web server makes development easier and faster because files can be edited directly from your computer and do not have to be uploaded to another server. This is sometimes referred to as the *sandbox*. In many cases, you can also test your applications without Internet access, although you will need connectivity if you are making any calls to the Twitter API. After you have a live site, developing locally will also give you a chance to test out your code in a sandboxed environment before making changes on your actual site. However, you should also set up a testing area

in the same environment as your live server because your local web server's configuration will differ from your webhost's.

Although LAMP packages are convenient and easy to install, they are typically used by developers looking to get a development environment up and running and may not be optimized for the best performance or security. Most production environments are set up carefully by an IT professional.

**By the
Way**

Introducing XAMPP

Our LAMP package of choice is called XAMPP, where the X is cross-platform and the extra "P" stands for Perl support (which we won't be utilizing in this book). It is one of the most popular LAMP distributions because it installs everything you'll need for most development with minimal effort, including phpMyAdmin, a popular web-based MySQL administration tool. Because it runs on Windows, Mac, Linux, and Solaris, you can be sure you'll have a consistent experience no matter what platform you're using.

Installing XAMPP

The XAMPP installation is so simple that it tricks you into thinking anybody could set up a web server. At the time of this writing, the current version is 1.7.2. Because XAMPP is frequently updated, your experience may differ slightly from what we describe here. To download the package, follow these steps:

1. Visit www.apachefriends.org.
2. Go to the XAMPP project page.
3. Click on the XAMPP icon at the top of the page to get to the links for the download page for your platform.
4. Depending on your platform, different options may be available. For your convenience, we've outlined quick installation tips for Windows, Mac, and Linux in the sections that follow.

Regardless of your operating system, installing XAMPP will overwrite any existing XAMPP installations. If you already have XAMPP, you can skip this section.

**Watch
Out!**

Windows

Windows users will see download options—XAMPP, XAMPP Lite, and XAMPP Add-Ons. For the purposes of this book, XAMPP Lite is fine. One of the biggest differences

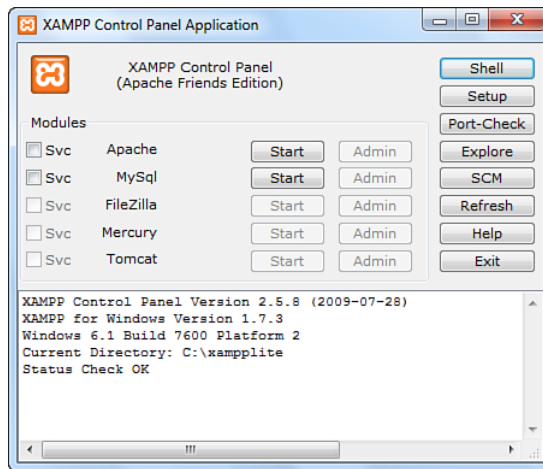
is that it doesn't include the FileZilla FTP Server or the Mercury Mail Transport System. The easiest option is to download the XAMPP Lite EXE, which includes an installer. Compared to the ZIP download, it is about half the download size because of better compression.

If you don't want XAMPP making changes to the Registry (required for all installed Windows applications), you can follow the instructions on the download page to install the ZIP version. You don't need the Mercury Mail Transport System to allow your applications to send emails. This is normally done using an existing SMTP server, such as one provided by Gmail or your web host.

Installing XAMPP Lite from the EXE installer is straightforward:

1. Fire up the downloaded XAMPP Lite EXE and click Install. The default option will extract all the files to C:\xampplite.
2. After the files are extracted, choose the default options in the command prompt windows that follow, until you reach the final menu, which has no default.
3. From the final menu, choose 1 to start the XAMPP control panel (shown in Figure 4.1).

FIGURE 4.1
XAMPP Control Panel (Windows).



4. Check the Svc boxes for both Apache and MySQL and confirm the prompts that follow.
5. Click Start next to both Apache and MySQL to fire up both servers.
6. You may now exit the command prompt menu and the XAMPP control panel.

A service is a program that starts with Windows and is often automatically run by the system even before a user logs in. If Apache and MySQL are not installed as system services, they must be started manually.

Mac

On the Mac download page, grab the Universal Binary—the file containing Apache, MySQL, and PHP. You do not need the developer package.

After you've downloaded the file, installing XAMPP on a Mac is just like installing any other application from a DMG file:

1. Open the downloaded DMG file.
2. Drag and drop the orange XAMPP folder into your Applications folder.
3. Open XAMPP Control in /Applications/XAMPP and start Apache and MySQL.

Linux

Download XAMPP Linux—the file containing Apache, MySQL, and PHP. If you're using Linux, we'll assume you pretty much know what you're doing, but we've outlined the installation steps to make sure we're all on the same page:

1. Open a terminal and switch users to root or run all the subsequent commands with `sudo`.
2. Extract the downloaded file to /opt:

```
tar xvfz xampp-linux-1.7.4.tar.gz -C /opt
```

3. Start XAMPP using the following command:

```
/opt/lampp/lampp start
```

If everything went okay, you should see something like the following output:

```
Starting XAMPP 1.7.4...
```

```
LAMPP: Starting Apache...
```

```
LAMPP: Starting MySQL...
```

```
LAMPP started.
```

```
Ready. Apache and MySQL are running.
```


If you encountered any errors during the preceding steps, make sure you were running all commands as root or prepending each command with `sudo`. Otherwise, take a look at the Linux FAQ at www.apachefriends.org/en/faq-xampp-linux.html#start.

Does It All Work?

Presumably, you now have a local PHP-enabled web server and a MySQL database. To see if it all works, open up a web browser and go to `http://localhost`.

If everything is working correctly, you'll be redirected to `http://localhost/xampp/splash.php`. If you see nothing, something is wrong. To troubleshoot, visit the download page for your platform on the XAMPP website for some tips, or try reinstalling XAMPP again from scratch using the directions on the XAMPP website. If you had an existing web server running on your machine, try uninstalling it and reinstalling XAMPP because they will conflict with each other if they are both using the default settings. You may also encounter a conflict if Skype is running because it uses ports 80 and 443 as alternative ports for incoming connections. If you are having trouble running XAMPP and have Skype installed, search the Skype documentation for "conflicts."

Take note of where XAMPP is installed. Inside the XAMPP root folder is a folder called `htdocs`. This is your web root, and files inside it are accessible via `http://localhost`. We'll be referring to the `htdocs` folder for the rest of the book; it's where all the code goes.

Securing Your Web Server

XAMPP is designed to be a convenient package to help web developers get a server running as easily as possible, but in many cases, convenience means risk. XAMPP's defaults leave your local web server open to risks on many fronts. Most people are behind a router that typically uses Network Address Translation (NAT) to share a single Internet connection. By default, NAT acts as a firewall by discarding all incoming requests to your network, including those bound for your web server.

Did You Know?

If you are using a home router and want others to be able to access your web server, forward TCP requests on port 80 (HTTP) to the local IP of your computer on your router's port forwarding configuration page. In some cases, your ISP may block incoming requests to port 80, so you may want to try another port, such as 8080, to port 80 of your local IP. When setting up port forwarding, you should also assign your server a static IP to prevent it from changing and breaking your forwarding rules.

The following are security issues with the default installation of XAMPP:

- ▶ The MySQL root account has no password.
- ▶ MySQL is accessible via network.
- ▶ phpMyAdmin is accessible via network.
- ▶ The XAMPP demo page is accessible via network.

In general, you'll want to ensure that all your passwords are secure—containing some combination of uppercase and lowercase letters, numbers, symbols, and no words or names that could be found in a dictionary.

For the rest of this book, we will use the following credentials for everything, but feel free to use your own secure usernames and passwords:

- ▶ **Username**—twitter or root (MySQL)
- ▶ **Password**—s0m3Th1ng

Note that we are using the same credentials everywhere so that this book is easy to follow. On your own sites, it's a good idea to use different secure credentials for everything. If you're wondering why there are no symbols in our password, at the time of this writing, there is a bug in the XAMPP security console for Windows that prevents you from setting passwords with symbols in them.

XAMPP Security Console

Although we need to fix a number of things, XAMPP makes it easy for us to take care of everything at once. The XAMPP security console will guide you through securing your XAMPP installation on all OSes, although each package may differ because each uses slightly different components.

Next, we outline some steps you can take to secure your web server. If you'd like more detail about the security issues the XAMPP security console addresses, we explain the main issues in detail in the next few sections. Otherwise, you can skip ahead to the "Development Tools" section.

Windows

The Windows version of XAMPP has a web-based security console to help you secure your local development server. Go to <http://localhost/security/> in your browser, and you will see a page with a title like "XAMPP SECURITY [Security Check 1.1]," which will give you a quick overview of security issues with your current configuration.

From here, you can click the link to <http://localhost/security/xamppsecurity.php>, which you'll be able to use in the following sections to secure MySQL and the XAMPP pages.

In the **MYSQL SECTION**, set the MySQL password to `s0m3Th1ng` using defaults for the rest of the options. If you'd like, you can store the password you set here in a text file by checking the last box before submitting the form. The saved password file cannot be accessed directly from your web server because it does not reside in `htdocs`.

Next, we need to secure the XAMPP directory in the second section. For simplicity's sake, we'll use the same login information we used for MySQL—`twitter` for the username and `s0m3Th1ng` for the password. Again, you can check the box if you want this login info saved to a text file.

After you've changed both passwords, you can go back to <http://localhost/security/>, and you'll notice that XAMPP has been secured. You may notice an **UNKNOWN** status by Tomcat because we have not installed that add-on.

Mac and Linux

Run `/Applications/XAMPP/xamppfiles/xampp security` (Mac) or `/opt/lampp/lampp security` (Linux) to secure your server. We assume that you use `twitter` for all usernames except for MySQL, which uses `root`, and `s0m3Th1ng` for all passwords.

XAMPP Pages

By default, all the pages under <http://localhost/>, including the demo pages at <http://localhost/xampp/>, are exposed and available to anybody who can see your computer on the network. Although the security page is available only from your local machine, other potentially sensitive information such as the `phpinfo()` page is still accessible without a password.

Did You Know?

The `phpinfo()` function outputs all details of your current PHP configuration and is useful when reconfiguring PHP, checking to see if a certain extension is enabled, or checking the values of certain server-side variables. Although XAMPP includes a page already, all it takes is a PHP file containing `<?php phpinfo()`.

MySQL

There are two MySQL issues that we'll point out. The most urgent issue is that there is no root password, so anybody can log in to your MySQL server and make any changes if the person knows your IP address or hostname. Our other potential issue is that the MySQL server is accessible from other IP addresses. For the purposes of this

book, no other computer will need access to your MySQL server, so we could restrict access to only localhost. However, because we have configured MySQL with a secure password, the default settings should be good enough for development purposes.

The root user is the default superuser for many systems, including MySQL and, most commonly, Linux.

Did You Know?

phpMyAdmin

Leaving phpMyAdmin open is just as bad as leaving MySQL open to attack, because it is a web-based interface to your MySQL database. The good news is that with the default options, securing MySQL means that your phpMyAdmin installation is secured because you'll need to use your new MySQL login for phpMyAdmin. We'll discuss phpMyAdmin some more in the next section, "Development Tools."

Development Tools

Having a web server set up is a good first step, but it's about as useful as having a foundation to a house without any tools to build the house with. Seasoned developers have their favorite tools, but if you're just getting started or want to try something new, we've listed some of our favorites in this section.

Firefox

One of the most important tools in any web developer's kit is Firefox. Sure, it might be obvious that you need a web browser to test your code, but Firefox is really valuable to developers because it has a plethora of powerful extensions available to help out with debugging. In the end, you'll want to make sure your pages work in Firefox because it's the second most popular browser used today, with about one-fourth market share.

Firebug

Firebug is a must in every web developer's toolbox. It gives you tools for HTML, CSS, and JavaScript and allows you to edit your pages in the browser and preview changes live, debug code, and optimize performance. You can easily find the HTML for any element on the page and edit it live or modify CSS styles on the page so you can quickly see what the changes do before you hardcode them. In JavaScript, you can see scripts in one place, whether they're inline or in external files. The JavaScript debugger can watch variables, inspect objects, and set breakpoints so that you can step through your code when you run into complex issues. For performance

optimization and AJAX development, the Net panel in Firebug allows you to monitor all network requests and see how long they look. This includes a visual graph of load and render time for all the initial elements on the page as well as all subsequent AJAX requests that occur in the background. Although Firebug adds important features to Firefox, it also slows it down significantly so it should only be enabled when needed.

Chrome

Google Chrome is quickly growing in popularity and like Safari, it is based on the Webkit rendering engine which is also used on many mobile devices. The two browsers account for more than 1 in 5 desktop users. Although it is important to test on all supported browsers, Chrome will often render sites similarly to Safari as well as the Android and iPhone browsers. Many web developers like Chrome too because it offers many developer tools that offer functionality similar to Firebug without the need for any add-ons or extensions.

Internet Explorer

Even if you think nobody uses Internet Explorer anymore, it still accounts for over one-third of all browser traffic on the Internet. If you're using Firefox to test most of the time, you'll often have to verify that everything works and looks the way it's supposed to from Internet Explorer. If you want to maximize your site's exposure, you should ensure that your site works in at least Internet Explorer (IE) 6 and later, since many machines running Windows XP still have IE 6.

phpMyAdmin

One of the most popular graphical MySQL administration tools is phpMyAdmin because it is open source, free, and web-based. It enables you to create and administer MySQL databases with minimal knowledge of MySQL, but it still has features useful to novices and professionals alike. As a result of phpMyAdmin's popularity, you will find it preinstalled on many web hosts. If not, installing it is as simple as extracting the files into a folder on a web server and entering the hostname and login credentials for your MySQL server.

For the rest of this book, phpMyAdmin will be our MySQL administration tool of choice, but if you already prefer using the command-line interface or some other tool, that will work equally well.

Text Editors

Although technically you can use any text editor (not a word processor) to write your code, using an editor with programming-specific features can help you code faster and more efficiently with features like syntax highlighting specific to each language. Hundreds of great text editors exist, and we've listed a few of our favorites for each platform:

- ▶ Notepad++ (Windows) is a lightweight text editor for Windows with many useful features for programmers, such as syntax highlighting in every major language and format, auto-completion, tabbed editors, and more. It comes with a variety of plug-ins, such as a simple FTP plug-in to let you edit remote files.
- ▶ TextMate (OS X) is a GUI editor much like Notepad++. Because of its rich feature set and extensive support for almost every language, many people who prefer a GUI-based editor and use a Mac stand by TextMate, even though it costs about \$60 at the time of this writing.
- ▶ Vim (all platforms) is a powerful modal text editor that is most popular among Linux users who often prefer its command-line interface over GUI-based text editors. Although it is most popular on Linux, it has been ported to every major operating system, including Windows and Mac. Users accustomed to GUI editors might find it difficult to use because it makes use of many customizable keyboard shortcuts for much of its functionality. However, most people who want an editor like vim probably already use it or something similar; therefore, we do not recommend it for beginners.

Integrated Development Environments (IDEs)

Although a simple text editor is lightweight, fast, and gets the job done, many developers prefer IDEs for their additional features, such as syntax checking and intelligent code completion. These features often function like a spell checker for your code—identifying and underlining syntax errors before you even execute your code. Some other common features include integration with revision control, bug tracking, and the capability to upload files directly to a web server from your IDE.

There are a handful of popular IDEs supporting PHP, but Eclipse and Netbeans are our favorites, and they're both free and cross-platform. Both also support a variety of programming languages, including Java and C/C++, but they are also great for web programming:

- ▶ Eclipse is a popular IDE that supports development for many platforms, languages, and devices. If you're looking to try out an IDE for the first time, we'd recommend giving Eclipse for PHP Developers a shot; it is a copy of Eclipse that comes with all the necessary plug-ins needed for PHP and general web development. Speaking of plug-ins, there are hundreds of thousands available to extend Eclipse's abilities from support for new languages, to support for revision control systems, to task and bug management. Because Eclipse is popular outside of web development, getting comfortable with it while programming in PHP may help you in other unrelated endeavors, such as Android mobile development.
- ▶ Netbeans has been around for a long time but recently added support for PHP. It offers many of the same features that you'll find in Eclipse but would be well suited for anybody who has already used it with other programming languages. However, it's a close match with Eclipse, so we suggest giving it a shot and deciding what works best for yourself.

Revision Control Systems

Revision control systems, also known as source code management (SCM) systems, are used to store and manage changes to files or documents. They are most commonly used in software development to manage source code. Because every revision of each file is stored, developers can always revert changes back to an earlier version of the code if things don't go as planned. If multiple developers make edits to the same file, the revision control system will automatically merge the changes and, if needed, facilitate resolving any conflicts.

There are many more uses for revision control, and we encourage you to research using one of the revision control systems listed next—especially if you're working with other people. Even if you're working alone, you can save yourself the frustration of making a change that breaks everything and not knowing what code caused the problem:

- ▶ Subversion (SVN) is one of the most popular revision control systems today and is designed to improve on the older Concurrent Versions System (CVS).
- ▶ Git is much like SVN for basic revision control, but one of its strengths is that there is a local repository, so changes can be committed locally even if there is no network access to the main server. Another advantage of Git is access to GitHub, which is a social coding website that allows you to host, share, and collaborate on your code. At the time of this writing, if your code is open source, it's a free service; otherwise, there is a monthly fee.

Many other differences exist between Git and SVN; however, it is difficult to discuss them without going into more depth on version control. If you're interested, search online for the differences and do some research on your own. In general, you'll find that Git is more powerful than SVN and is rapidly becoming more popular, but it may be more confusing to new users. If you have no experience using version control, we recommend you start with SVN because it is widely used and there are clients for more platforms and programs than Git.

Our Recommended Toolbox

If you're new to web development, we don't want to scare you off with all these possibilities. If you're just starting off, we recommend that you install the following tools (described in the previous sections):

- ▶ **Firefox**—The second most popular browser in use today; it works on all platforms and supports a variety of extensions to aid you in web development and testing.
- ▶ **Firebug**—A popular Firefox extension for web developers; it helps you debug issues with HTML, CSS, and JavaScript.
- ▶ **Chrome**—The most popular Webkit based desktop browser has many integrated development tools to help you debug and tune your site and has similar behavior to Safari and many mobile devices.
- ▶ **Eclipse for PHP Developers**—This powerful IDE will help you program with code completion and help you quickly find mistakes with syntax checking.

We've recommended these tools because they'll work on all platforms and should get you through the entire book. However, you should always test all web pages in Internet Explorer because it is the most popular browser.

After you're comfortable with these tools, try using revision control to avoid irreversible changes and prepare yourself for working on a team of developers. Revision control systems are used by almost all professional software developers.

Summary

In this hour, we set you up with everything you need to start writing and testing PHP code. You learned about LAMP stacks and how to set up your own local web server for development purposes. You also learned about a few security vulnerabilities you'll

encounter with a fresh XAMPP install and how to fix them. After configuring and testing your local web server, you learned about some of our favorite web development tools, which we will use for the rest of the book.

Q&A

Q. *Why are LAMP stacks so popular?*

A. It's no coincidence that LAMP stacks are popular in the web world. Because all of the technologies in LAMP are open source, free, and widely supported, they are cheap and easy to deploy.

Q. *If I already have an *AMP server set up, is there any reason to install XAMPP?*

A. In theory, almost any *AMP configuration with a current version of PHP should work with the examples in this book. However, because there are so many options to configure within Apache, MySQL, and PHP, we cannot guarantee that things will work unless you're using XAMPP.

Q. *Why do you recommend an IDE for beginners over a simple text editor?*

A. People generally take two sides when deciding what editor to recommend to beginners. There's no doubt that an IDE can offer useful tips and can help you find errors in your code faster, but some people argue that an IDE can become a crutch and others feel strongly against installing a large program with many features they will not use. For us, our recommendation is simple. We're here to teach you about the Twitter API, not to teach you PHP, HTML, CSS, or JavaScript. We want your full attention on the Twitter API, not the nuances of programming and markup languages we're using in this book; using an IDE can help you worry less about the code, and more about the API itself.

Workshop

Quiz

1. What does LAMP stand for?
 - A. Linux Apache MySQL Python
 - B. Linux Apache MySQL PHP
 - C. Linux Apache MySQL Perl
 - D. All of the above

2. In XAMPP, what is the name of the folder containing the files or scripts that you can access from `http://localhost`?
3. True or False: When testing a web page, you can safely assume that it will look the same in all web browsers as long as you are using valid HTML.

Quiz Answers

1. D. Although LAMP most commonly refers to the use of Linux, Apache, MySQL, and PHP, the “P” can also stand for Perl or Python—other popular scripting languages.
2. `htdocs` is the folder sometimes referred to as the web root, which is where you put all the files that are to be accessible from the web server. If you aren’t using XAMPP, it’s sometimes named `www`.
3. False. All web pages should be tested in at least Internet Explorer, Firefox, and Chrome as pages may render differently in every browser and on every operating system. If you want to be thorough, you should try your web pages in Safari as well as on Windows, Mac, and Linux. As a final check, you should test your site on a variety of popular mobile devices.

Exercises

1. Create your own `phpinfo()` page and access it from your browser. Read over the page as it contains interesting information about your server’s configuration and environment variables. When you begin to develop more complex applications, this information will come in handy, especially if your application requires special PHP extensions.
2. Set up your development and testing environment, including an editor, Internet Explorer, Firefox, Chrome, Safari, and the Firebug browser extension we mentioned in this hour. When we get around to building, you don’t want to be distracted by having to get your toolbox in order.
3. Try setting up a version control system. We recommend Git for more experienced users, but if you’re just getting started and Git looks too complicated for you, try SVN first until you’re comfortable with it. However, it is important to familiarize yourself with Git because it and other distributed version control systems are quickly gaining popularity for new projects due to their power, speed, and flexibility.

This page intentionally left blank

HOUR 5

Making Your First API Call

What You'll Learn in This Hour:

- ▶ How to make a simple URL call
- ▶ How to make a call in php

Making a Simple Twitter API Call

OK, it's time make a call to Twitter and get a response. Let's dive right in and make a simple call to the Twitter service to get the latest public timeline. Depending on your browser and its configuration you may need to view source to see the proper formatting.

Open a web browser, type in the following, and press Enter:

```
http://twitter.com/statuses/public_timeline.xml
```

You should get something like the following:

```
<statuses type="array">
<status>
  <created_at>Thu Dec 31 03:31:08 +0000 2009</created_at>
  <id>7220013867</id>
  <text>Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  eiusmod tempor
  incididunt ut labore et dolore magna aliqua.</text>
  <source>&lt;a href=&quot;http://apiwiki.twitter.com/&quot;
  rel=&quot;nofollow&quot;&gt;API&lt;/a&gt;</source>
  <truncated>>false</truncated>
  <in_reply_to_status_id></in_reply_to_status_id>
  <in_reply_to_user_id></in_reply_to_user_id>
  <favorited>>false</favorited>
  <in_reply_to_screen_name></in_reply_to_screen_name>
  <user>
    <id>11710512</id>
    <name>Christopher</name>
```

```

    <screen_name>perivision</screen_name>
    <location>the bay</location>
    <description>I make stuff. Currently I'm making twittFilter.com Check
    ↪ it out and
    tell me what you think.</description>
    <pro
file_image_url>http://a1.twimg.com/profile_images/61029076/
chrisperi2_normal.jpg</profile_image_url>
    <url>http://www.perivision.net</url>
    <protected>>false</protected>
    <followers_count>830</followers_count>
    <profile_background_color>C6E2EE</profile_background_color>
    <profile_text_color>663B12</profile_text_color>
    <profile_link_color>1F98C7</profile_link_color>
    <profile_sidebar_fill_color>DAECF4</profile_sidebar_fill_color>
    <profile_sidebar_border_color>C6E2EE</profile_sidebar_border_color>
    <friends_count>172</friends_count>
    <created_at>Tue Jan 01 09:29:37 +0000 2008</created_at>
    <favourites_count>26</favourites_count>
    <utc_offset>-28800</utc_offset>
    <time_zone>Pacific Time (US & Canada)</time_zone>
    <pro
file_background_image_url>http://s.twimg.com/a/1262113883/images/themes/the
me2/bg.gif</profile_background_image_url>
    <profile_background_tile>>false</profile_background_tile>
    <notifications>>false</notifications>
    <geo_enabled>>false</geo_enabled>
    <verified>>false</verified>
    <following>>false</following>
    <statuses_count>3416</statuses_count>
    </user>
    <geo/>
</status>
<status>
    <created_at>Thu Dec 31 03:30:37 +0000 2009</created_at> ...

```

Quite the mess, isn't it? Well, no worries. We are going to clean this up a bit so that you can see what's going on. First, we made a call request in XML format. We could have made that call in JSON, RSS, or ATOM, but we are sticking with XML for now.

The first thing you should notice is the nesting of information. To make this a bit easier to explore, let's remove the content and simplify this output:

Statuses

```

    created_at
    id
    text
    truncated

```

```
in_reply_to_status_id
in_reply_to_user_id
favourited
in_reply_to_screen_name
user
    id
    name
    screen_name
    location
    description
    url
    protected
    followers_count
    profile_background_color
    profile_text_color
    profile_link_color
    profile_sidebar_fill_color
    friends_count
    created_at
    favourites_count
    utc_offset
    time_zone
    profile_background_tile
    statuses_count
    notifications
    following
    verified
geo
```

Although this is a bit easier to read, we can pare this down even further by focusing on the key fields we would use to display a typical message:

Statuses

```

    created_at
    id
    text
    user
      id
      name
      screen_name
      utc_offset
  ..

```

Now we have something a bit easier to deal with. These will be the values we will want to read for each message, or statuses, that we get back from Twitter. So, let's go over each node of the XML schema quickly:

Created_at—This is the time the message was created in UTC Coordinated Universal Time.

Id—This is the numerical ID number of the message.

Text—This is the contents of the message.

User—This is the Twitter user who created the message. This node also has children. The nodes inside “user” contain information about the user.

User->id—This is the numerical id of the writer.

User->name—This is the name of the writer.

User->screen_name—This is the writer's screen name. The writer's name and screen_name are sometimes the same thing.

User->utc_offset—This is the time zone offset that is relative to the creator of the message. Because the created_at value is always at UTC, we need this value in order to display the created time relative to our own time zone.

Watch Out!

A Twitter screen name is restrictive. You cannot use the word “twitter,” cannot have spaces, and certain characters are not allowed.

Making a Call in PHP

Now that we have connected to Twitter, it's time to do it using PHP code. You should have your development environment ready go, as outlined in Hour 4, "Creating a Development Environment." Let's create a new php file called `get_public_timeline.php`. The first line will read as follows:

```
<?php
```

This lets the server know that all the following lines are to be treated as PHP code. Next, type in the following line:

```
$api_url = 'http://twitter.com/statuses/public_timeline.xml';
```

Here we are setting the variable `$api_url` to the string `http://twitter.com/statuses/public_timeline.xml`. The next few lines might be a bit confusing for those who are not used to PHP. It's not in the scope of this book to go into great detail on the cURL library, but we will touch on it. Type these lines into your file:

```
$curl_handle = curl_init();  
curl_setopt($curl_handle, CURLOPT_URL, $api_url);  
curl_setopt($curl_handle, CURLOPT_RETURNTRANSFER, TRUE);  
$twitter_data = curl_exec($curl_handle);  
curl_close($curl_handle);
```

What Is cURL?

cURL, sometimes written as curl, is a set of C-based libraries in PHP that support http "get" and "post" protocol communication with a server. The name "curl" is the contraction of "client" and "URL." The outcome of this project is a set of routines referred to as the *libcurl*, which is shorthand for curl libraries. These libraries support FTP, FTPS, HTTP, HTTP POST, and many more. The library is also very well supported on various platforms: Windows, Mac OS X, Linux, Solaris, OpenBSD, and more. Even better, libcurl is free, thread-safe, and supports IPv6.

What's important to us is the PHP libcurl, which is typically included in any well-rounded PHP installation. This library and the functions within are how we are going to manage our communications with Twitter. So, let's go through the lines we just added to our `get_publictimeline.php` file.

This line creates a reference to the curl library and assigns that reference to the variable `$curl_handle`:

```
$curl_handle = curl_init();
```


These next two lines set options within our curl object. It's important to notice here that the CURLOPT_URL is using a variable called `$api_url`:

```
curl_setopt($curl_handle, CURLOPT_URL, $api_url);
curl_setopt($curl_handle, CURLOPT_RETURNTRANSFER, TRUE);
```

These last two lines should be clear. The first makes the actual curl call to Twitter.com and assigns `$twitter_data` to results. The last line closes our reference object:

```
$twitter_data = curl_exec($curl_handle);
curl_close($curl_handle);
```

Okay, only two more lines to go. Type these in:

```
echo 'Public Stream <br>';
echo '<pre> '.htmlentities($twitter_data);
?>
```

The first line uses the command `echo` to print 'Public Stream' in your browser and the '`
`' is HTML to end that line and start a new one. The following line also uses `echo` to print information in the browser. The `<pre>` HTML tag and the `htmlentities()` function are there to make the display more readable for us.

Save your file. Make sure your local PHP server is up and running, and open a web browser. Open your file `get_public_timeline.php` in the browser.

Did you get an output of text similar to the text below? Great! This is pretty much the same thing we saw when we made this API call from the browser in the beginning of this hour:

```
<statuses type="array">
<status>
  <created_at>Thu Dec 31 03:31:08 +0000 2009</created_at>
  <id>7220013867</id>
  <text> Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  ➤ eiusmod tempor
  incididunt ut labore et dolore magna aliqua.</text>
  <source>&lt;a href=&quot;http://apiwiki.twitter.com/&quot;
  rel=&quot;nofollow&quot;&gt;API&lt;/a&gt;</source>
  <truncated>>false</truncated>
  <in_reply_to_status_id></in_reply_to_status_id>
  <in_reply_to_user_id></in_reply_to_user_id>
  <favorited>>false</favorited>
  <in_reply_to_screen_name></in_reply_to_screen_name>
  <user>
    <id>11710512</id>
    <name>Christopher</name>
    <screen_name>perivision</screen_name>
    <location>the bay</location>
  ...
```

Congratulations! You just made your first Twitter API call from PHP using cURL. Now, let's make another API call to a specific Twitter account.

User_timeline API

The API call to get the public timeline is `http://twitter.com/statuses/public_timeline.xml`. The API call to get a specific twitter user's timeline is `http://twitter.com/statuses/user_timeline/<users screen name>.xml`. For example, if I wanted to see the timeline for perivision, I would type the following into a web browser:

```
http://twitter.com/statuses/user_timeline/perivision.xml.
```

However, we want to do this in our code. All we have to do is make one change in our code. So, let's create a new php file and call it `get_user_timeline.php`. Copy the lines from `get_public_timeline.php` into our new file.

Next, change this line:

```
$api_url = 'http://twitter.com/statuses/public_timeline.xml';
```

To this:

```
$api_url = 'http://twitter.com/statuses/user_timeline/perivision.xml';
```

Make the change in your code and give it a try. You should see basically the same XML schema we have been looking at throughout this hour, but with different content.

However, suppose you want to read *your* timeline. There is an API call for that too, called `../home_timeline`. However, this XML structure is a little different from the other two. Also, readers who have used the Twitter API may be wondering why we are not using `../friends_timeline`. The `friends_timeline` API call is still functional, but it's being deprecated in favor of `../home_timeline` because this new API call will include replies. Let's see what's different about `../home_timeline`.

As you might recall, the simplified `public_timeline` XML schema from the beginning of this hour looks something like this:

Statuses

created_at

id

text

user

id

```

name
screen_name

utc_offset

```

...

Now, if a message was in reply to a previous message, a new child node will be presented. Here is the reply node in full:

```

<retweeted_status>
  <created_at>Wed Nov 18 18:36:34 +0000 2009</created_at>
  <id>5833513351</id>
  <text>
    Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
    ↪tempor
    incididunt ut labore et dolore magna aliqua.</text>
    <source><a href="http://www.hootsuite.com"
    ↪rel="nofollow">HootSuite</a></source>
    <truncated>false</truncated>
    <in_reply_to_status_id></in_reply_to_status_id>
    <in_reply_to_user_id></in_reply_to_user_id>
    <favorited>false</favorited>
    <in_reply_to_screen_name></in_reply_to_screen_name>
    <user>
      <id>2558...</id>
      <name> jfieuu uremd</name>
      <screen_name>jfieuu uremd</screen_name>
      <location></location>
      <description>jfisoje jilsifijis jifle</description>
      <profile_image_url>http://a3.twimg.com/profile_images/11621.../
      ↪hjkuyi_73x73_normal.jpg</profile_image_url>
      <file_image_url>
        <url>http://www.trdytf.com</url>
        <protected>false</protected>
        <followers_count>1653473</followers_count>
        <profile_background_color>08a9e7</profile_background_color>
        <profile_text_color>000000</profile_text_color>
        <profile_link_color>ee0077</profile_link_color>
        <profile_sidebar_fill_color>ffee9a</profile_sidebar_fill_color>
        <profile_sidebar_border_color>ffcc66</profile_sidebar_border_color>
        <friends_count>406</friends_count>
        <created_at>Fri Mar 20 22:30:24 +0000 2009</created_at>
        <favourites_count>2</favourites_count>
        <utc_offset>-18000</utc_offset>
        <time_zone>Eastern Time (US & Canada)</time_zone>
        <profile_image_url>http://a1.twimg.com/profile_background_images/
        ↪6859.../bgpage.g
      </profile_image_url>
    </user>
  </retweeted_status>

```

```

    <profile_background_tile>true</profile_background_tile>
    <statuses_count>1740</statuses_count>
    <notifications>>false</notifications>
    <geo_enabled>>false</geo_enabled>
    <verified>>false</verified>
    <following>>false</following>
  </user>
  <geo/>
</retweeted_status>

```

Let's simplify this a bit more:

```

<retweeted_status>
  <created_at>Wed Nov 18 18:36:34 +0000 2009</created_at>
  <id>5833513351</id>
  <text>
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
tempor
incididunt ut labore et dolore magna aliqua </text>
  <user>
    <id>255897...</id>
    <name>hkhukg</name>
    <screen_name>hjhkhjiiu</screen_name>
  </user>
  <geo/>
</retweeted_status>

```

We can see here that the reply node has some things in common with the message node structure we have seen before. However, for now, we are going ignore this node, but we will revisit it in later hours.

In addition to an additional node within the `user_timeline` API call, there is a new requirement we have not seen before: You must provide user credentials. When we first started writing this book, you could simply add your username and password along with the cURL request. This is no longer supported because Twitter has moved to an OAuth authentication protocol only. But don't worry—we will be covering that in Hour 8, "Twitter OAuth."

Summary

Great! We have just made two API calls in this hour, both from the browser and from PHP using cURL. Now it's time to take our return from Twitter and parse it so that we can display our returns in a format a little more user friendly than what we have seen so far.

Q&A

Q. Can we make only API calls in XML?

A. No. You can use JSON, RSS, or ATOM. However, we have decided to stick with XML here because of how well known it is.

Q. What is the difference between a message and a status within Twitter?

A. There is no difference. Although it can be confusing, a status is actually a message, not the actual status of the user.

Q. Do I have to pay for the cURL libraries in PHP?

A. No, cURL is free.

Workshop

Quiz

1. What is the difference between a user ID, username, and user screen_name?
2. Why is the friends_timeline API call being deprecated for the new home_timeline API call?

Quiz Answers

1. The user ID is a numeric identifier for the user. The username is the name the user used when he or she registered. The user screen name is the actual Twitter handle used within the Twitter system.
2. The home_timeline API call now includes information about the original status if the status being viewed is a reply.

Exercise

Create an HTML text field that enables you to enter in a Twitter name and then make a Twitter API call to get the timeline of the Twitter name entered.

HOUR 6

Building a Simple Twitter Reader

What You'll Learn in This Hour:

- ▶ How to set up a basic file structure for our Twitter client application
- ▶ How to parse returned data from Twitter
- ▶ What is an HTTP response code?

Building Our First Twitter Client

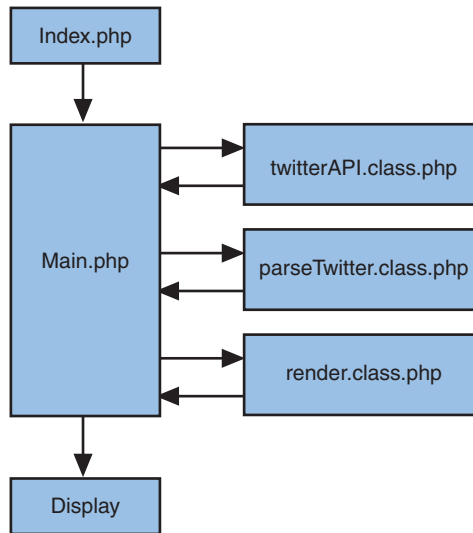
We are in Hour 6 and, believe it or not, we already have the basics required to create our first simple Twitter Client. In this hour, we will create the basic file structure that you would see with any typical website: the HTML, CSS, and PHP code to create the content.

In the previous hour, we created a single standalone PHP file to make calls to and get data back from Twitter. However, this is not the proper way to structure a program. It is the goal of this book to have you create a Twitter client by the end of the 24 hours. As such, we are going to take a more structured approach to how we create our application framework. If it's been a while since you read Hour 3, "Key Issues to Consider When Developing Twitter Applications," I recommend that you go back and give it a quick review.

As we go through this Hour and the rest of the Hours in the book, more advanced programmers may be wondering why we took certain approaches to our coding or choose to use what would seem overly simple techniques. After much debate by the writers and advisors, we decided that at the price of elegant coding, we would try to make things more clear to the beginning programmers.

Let's put our simple framework in place. It's a basic framework compared to more sophisticated applications. Our basic setup will have an HTML file that is called when the user first comes to the site, which will load `main.php`, as well as call three other files, each designed to perform a certain task, as you can see in Figure 6.1.

FIGURE 6.1
A simple client application framework.



We are going to create six files in this hour: the HTML page, the CSS file, and four PHP files. Three of the PHP files will live in a subdirectory called `includes`.

Creating `index.php` and `main.css`

First, let's create the HTML code. We are not going to go into detail about the HTML nor the CSS files, which is not in the scope of this book. You may have noticed that we are using `index.php` instead of `index.html`. Depending on your server's configuration, you can use either one, however convention is to use `index.php` if you have a PHP code in your file, and we very much do in our example. Create a new file called `index.php` in your editor and type in the following code:

Watch Out!

Recall That HTML Processors Ignore Extraneous Whitespace

Although most environments will allow you to mix HTML and PHP code by simply calling out the PHP code using `<?php ... ?>`, this is not true with all environments. If you find that your PHP codes are being displayed on your web page, change the name of `index.html` to `index.php`.

```

<meta charset="utf-8">
<html>
  <head>

```

```

    <title>twitterAPI 24 hours</title>
    <link href="css/main.css" rel="stylesheet" />
</head>

```

Here we will make a PHP call to load the various files we need to run our application. As a means of clarity, we are going to place some of our new files into a folder called includes. We use this convention to refer to files and the functions that are not called on their own:

```

<?php
    include 'includes/twitterAPI.php';
    include 'includes/parseTwitter.php';
    include 'includes/render.php';
    ?>
<body>
    <div class="header">
        <div class="tweet">TwitterAPI24</div>
    </div>
    <div class="container">

```

Next, we make a call to main.php, which we will use to load our content:

```

    <? include 'main.php'; ?>
</div>

</body>
</html>

```

Creating main.css

It's not in the scope of this book to explain how CSS works, so we will simply provide the CSS code here. Create a folder called css and then create a file in that folder called main.css. Type in the following code:

```

/* Global Layout */
body {
    margin:0;
    padding:0;
    font:12px Arial, Helvetica, sans-serif;
    color:#999;
    width:100%;
}
/* End of Global Layout */

/* Global Styles */
/* Disable CSS Text Decoration Property */
a {
    text-decoration:none;
    color:#667;
}

```



```

/* Enable CSS Text Decoration Property and select underline */
a:hover {
    text-decoration:underline;
}

/* End of Global Styles */

/* Section: Header */
.header {
    margin:0 auto;
    padding:10px 10px;
    background:#333;
    overflow:hidden;
}
.tweet {
    padding:0 10px;
}

/* Section: Table */
/* Format the Detail Table with border */
#detail_table td{
    width:330px;
    padding-top:2px;
    padding-left:5px;
    padding-bottom:2px;
    vertical-align: top;
    color:#333;
    border-bottom:1px solid #DBDBDB;
    font-family: Trebuchet MS, Arial, Helvetica, sans-serif;
    valign: top;
}

/* Section: Messages */
/* Format the Message container */
.container{
    float:left;
    width:320px;
}
/* Format the Message image */
.mess-pic {
    background-color:#eee;
    padding-right: 5px;
    font-size: 10px;
    float:left;
}
/* Format the Message container */
.mess-container{
    float:left; width: 250px; padding-left:10px;
}
/* Format the Message content */
.mess-row-text{
float:right; padding-left:10px; width:250px;
}

```

Great. Now that we have the presentation layer out of the way, we can get to the actual code. We are going to create four more files: `main.php`, `parseTwitter.php`, `render.php`, and `twitterAPI.php`.

Creating `main.php`

First, let's start with `main.php`. This file will make calls to all other functions we need to support our simple Twitter client application. The `main.php` script should not do any tasks in and of itself, but instead serve as the traffic cop for calling other functions. In this example, `main.php` is going to make three calls: first to `callTwitter()` from `twitterAPI.php` to get the latest messages from the Twitter servers; then it will call `parseTwitterReply()` from `parseTwitter.php` to convert the returned data into something more usable within PHP; finally, it sends the formatted HTML code of `renderTweets()` of `render.php` to the user's browser.

To get started, let's create `main.php` and type in our first line of code:

```
<?php
```

In the next line, we manually defined `$twitterName`. `BREAKINGNEWS` is the name of a Twitter account. You could replace this with any valid Twitter account with public tweets. We appended `.xml` to let Twitter know we want the response to our request to be in XML format:

```
    $twitterName =  
'http://api.twitter.com/1/statuses/user_timeline/BREAKINGNEWS.xml';
```

Next, we make a call to a function called `callTwitter()` in our `twitterAPI.php` file. It is here that we make the actual call to the Twitter servers. We will create this file later in this hour:

```
    $twitterRequest = callTwitter($twitterName);
```

After we get our return from the `callTwitter()` function, we call `parseTwitterReply` in our `parseTwitter.php` file to convert the XML text reply we got from Twitter into a structured PHP object that is easier to work with. Again, do not worry; we will create this file later in this hour:

```
    $twitterRequest = parseTwitterReply($twitterRequest);
```

After we have our reply from Twitter in a format we can use, we make a call to `renderTweets` in `render.php` to create the HTML code we will send to the user's browser:

```
    echo renderTweets($twitterRequest);
```

```
?>
```

Creating twitterAPI.php

Now that we have our main file, let's create the file `twitterAPI.php` that will be placed in the `includes` folder. We are going to use this file only twice. After this hour and the following hour, we will replace it with a collection of files that support OAuth authentication with Twitter. The `twitterAPI.php` file basically takes our request and makes a cURL call to Twitter for us, just as we did in Hour 5:

```
<?php
```

```
function callTwitter($api_url){
```

First, we create a curl object to allow us to communicate with the Twitter servers. That object is assigned to `$curl_handle`:

```
    $curl_handle = curl_init();
```

Next, set options for our new curl object. These are metadata elements we will send along with our request to the Twitter servers:

```
        curl_setopt($curl_handle, CURLOPT_URL, $api_url);
        curl_setopt($curl_handle, CURLOPT_RETURNTRANSFER, TRUE);
```

Here we set the variable `$twitterResponseData` to contain the data that Twitter has sent back to us:

```
    $twitterResponseData = curl_exec($curl_handle);
```

After a message is sent back to our cURL object, we can find out what the HTTP response code is. We can then use that code to determine if there was a problem with our request, what the problem was, and what we want to do about it. Don't worry about what an HTTP response code is at the moment; we will cover this later in this hour:

```
    $errCode=curl_getinfo($curl_handle, CURLINFO_HTTP_CODE);
```

Although we could set an action for each error code, we are only interested in whether the call was successful. So, if the response code is anything but 200, we will display an error message with whatever other information Twitter has sent us:

```
        if(!strstr($errCode,'200')) {echo 'err '.$errCode; return; }
```

Now that we are done with our API call, we do not need the connection any longer. So, here we close it:

```
        curl_close($curl_handle);
```

Assuming a successful API call, we return our data from Twitter to the function that called it:

```
        return $twitterResponseData;
    }
?>
```

You may have noticed that we have not entered any authentication information. That is because the public and user API calls are public API calls, not requiring us to send login information.

**By the
Way**

Twitter HTTP Response Codes

Every time you make a restful call to the Twitter API service, you will get an 'HTTP response code' in addition to whatever data you requested. These response codes are important for letting the application, and the user, know when something did go as expected. Here is a list of the currently supplied codes from Twitter. These codes are based on the current standard for HTTP response codes; they are not specific to Twitter with exception to one:

200 OK—Success!

304 Not Modified—There was no new data to return.

400 Bad Request—The request was invalid. An accompanying error message will explain why. This is the status code will be returned during rate limiting.

401 Unauthorized—Authentication credentials were missing or incorrect.

403 Forbidden—The request is understood, but it has been refused. An accompanying error message will explain why. This code is used when requests are being denied due to update limits.

404 Not Found—The URI requested is invalid or the resource requested, such as a user, does not exist.

406 Not Acceptable—Returned by the Search API when an invalid format is specified in the request.

420 Enhance Your Calm—Returned by the Search and Trends API when you are being rate limited.

500 Internal Server Error—Something is broken. Please post to the group so the Twitter team can investigate.

502 Bad Gateway—Twitter is down or being upgraded.

503 Service Unavailable—The Twitter servers are up, but overloaded with requests. Try again later.

Depending on the amount of time you want to invest, you could write an exception for each code and present the user with a custom message, as well as perform follow-up actions. For example, codes 500, 502, and 503 have to do with Twitter being unable to fulfill a request. If you get one of these codes back, you can provide a message to your users that Twitter is unavailable. You could create a custom page to show the famous FailWhale as shown on the creators website (<http://www.whatisfail-whale.info/>). In our case, we will simply display whatever message Twitter gave back to us if the response code is not 200.

Did You Know?

The HTTP response code 420 is not a standard response code. Instead, Twitter created it for their own use. We will cover this more in Hour 20.

Creating `parseTwitter.php`

Now let's create `parseTwitter.php`. This file should be in the `includes` folder. This file is focused on making calls to and parsing the returns from Twitter:

```
<?php
function parseTwitterReply($messages){
```

The variable `$messages` has the reply we got from Twitter. Because our reply is in XML, we are going to use a standard PHP 5 function called `SimpleXMLElement()` that will take our XML reply and convert it into an object we can work with more easily within PHP. Explaining the `SimpleXMLElement` object is not in the scope of this book. However, you can learn more at <http://php.net/manual/en/book.simplexml.php>.

```
    $twitterReturn = new SimpleXMLElement($messages);
    $i=0;
```

Now that we have our response in an object, we can get the values we are interested in. We will create a simple 'foreach' loop and grab the values for `<created_at>`, `<text>`, `<profile_image_url>`, and `<screen_name>`. These are the only four values we need to create our simple display.

In the following loop, we are going to create a set of array objects to hold each tweet as we loop through, the following are the values we are looking for:

`$updateTime`: the time a tweet was created.

`$update`: the content of the tweet.

`$profile_image_url`: the user's profile image.

`$screen_name`: the screen name of the user.

```

foreach($twitterReturn->status as $status){
    $updateTime[$i] = $status->created_at;
    $update[$i] = $status->text;
    $profile_image_url[$i] = $status->user->profile_image_url;
    $screen_name[$i] = $status->user->screen_name;
    $i++;
}

```

We now have all the tweets that Twitter gave us back into arrays. You may have noticed that we defined our array using `$i`. Not every element returned by Twitter will have a value, so we use `$i` to ensure that the arrays for the first tweet have information for the first tweet returned, and the second set of arrays contains information from the second tweet, and so on.

Now we will create a new array object called `$parseReturn` to hold the arrays we just created:

```

    $parsedReturn = array();
    $parsedReturn['updateTime']=$updateTime;
    $parsedReturn['update']=$update;
    $parsedReturn['profile_image_url']=$profile_image_url;
    $parsedReturn['screen_name']=$screen_name;

    return $parsedReturn;
}
?>

```

Creating render.php

Next, let's create `render.php`. This file will take the returns from `parseTwitter.php` and format them in HTML to be displayed on our page. This will also be in the `includes` folder:

```
<?php
```

```
function renderTweets($parsedReturn){
```

In this function, we are going to loop through each set of array objects that we created in `parseTwitter`. We will then create an object that will contain the HTML codes we need to display our tweets:

```

    $num = count ($parsedReturn['update']);
    $output="";

```

Notice that we are using `$i` again to control our loop. This is to make sure we get the correct values for each tweet. The first part of the loop will get the values from our `$parseReturn` object:

```

    for($i=0; $i<$num; $i++){
        $updateTime = $parsedReturn['updateTime'][$i];
        $update = $parsedReturn['update'][$i];
        $profile_image_url =
➔$parsedReturn['profile_image_url'][$i];
        $screen_name = $parsedReturn['screen_name'][$i];
        $textBody = $update.' '.$updateTime;

```

Because a screen name can be pretty long, we are going to cut it off at nine characters:

```

        $screen_name_abv="<a
➔href='http://www.twitter.com/$screen_name'
➔target='_blank'>".strtolower(substr($screen_name,0,8))."</a>";

```

Now that we have our values, we can populate `$output` with HTML code:

```

        $output.= "<tr><td><div id='$screen_name' class='mess-
➔pic' >
            <img src='$profile_image_url' / width='48px'
➔height='48px'>
                <br>$screen_name_abv</div>
                <div class='mess-container'>
                <div class='mess-row-text'>$textBody</div></div>
                <div style='clear: both; padding-top:
➔10px'></div></td></tr>";
        }

        return $output;
    }

?>

```

So, let's test out our code. Go to your browser and open `index.html`. You should see a stream of tweets from BREAKINGNEWS similar to Figure 6.2. Congratulations! You have made your first step in creating a basic Twitter client by creating a Twitter reader!

Did You Know?

Did you notice the `+0000` after the date and time? This indicates the time zone that the displayed date is based on. `+0000` means the time based on GMT—Greenwich Mean Time, also known as Coordinated Universal Time (UTC).



FIGURE 6.2
Example of
tweets from
BREAK-
INGNEWS.

Summary

In this hour, we created a basic Twitter reader. This will serve as the foundation for the rest of the examples we will use in the book. You were also introduced to file structure, making calls to Twitter, parsing the reply, and displaying the data in a web-friendly format.

PHP Code Created in This Hour

We created quite a number of files and lots of code in this hour. As such, we are showing all the code for this hour again for your convenience.

index.php:

```
<meta charset="utf-8">
<html>
  <head>
    <title>twitterAPI 24 hours</title>
    <link href="css/main.css" rel="stylesheet" />
  </head>
```

main.php:

```
<?php
    $twitterName =
'http://api.twitter.com/1/statuses/user_timeline/BREAKINGNEWS.xml';
    $twitterRequest = callTwitter($twitterName);
    $twitterRequest = parseTwitterReply($twitterRequest);
    echo renderTweets($twitterRequest);
```


?>

twitterAPI.php:

```
<?php
function callTwitter($api_url){
    $curl_handle = curl_init();

    curl_setopt($curl_handle, CURLOPT_URL, $api_url);
    curl_setopt($curl_handle, CURLOPT_RETURNTRANSFER, TRUE);
    $twitterResponseData = curl_exec($curl_handle);

    $errCode=curl_getinfo($curl_handle, CURLINFO_HTTP_CODE);
    if(!strstr($errCode,'200')) {echo 'err '.$errCode; return; }
    curl_close($curl_handle);

    return $twitterResponseData;
}
?>
```

parseTwitterReply.php:

```
<?php
function parseTwitter($messages){

    $twitterReturn = new SimpleXMLElement($messages);
    $i=0;
    foreach($twitterReturn->status as $status){
        $updateTime[$i] = $status->created_at;
        $update[$i] = $status->text;
        $profile_image_url[$i] = $status->user->profile_image_url;
        $screen_name[$i] = $status->user->screen_name;
        $i++;
    }

    $parsedReturn = array();
    $parsedReturn['updateTime']=$updateTime;
    $parsedReturn['update']=$update;
    $parsedReturn['profile_image_url']=$profile_image_url;
    $parsedReturn['screen_name']=$screen_name;

    return $parsedReturn;
}
?>
```

render.php:

```
<?php
function renderTweets($parsedReturn){
```

```

$num = count ($parsedReturn['update']);
$output="";

for($i=0; $i<$num; $i++){
    $updateTime = $parsedReturn['updateTime'][$i];
    $update = $parsedReturn['update'][$i];
    $profile_image_url =
↳$parsedReturn['profile_image_url'][$i];
    $screen_name = $parsedReturn['screen_name'][$i];
    $textBody = $update.' '.$updateTime;

    $screen_name_abv="<a
↳href='http://www.twitter.com/$screen_name'
↳target='_blank'>".strtolower(substr($screen_name,0,8))."</a>";

    $output.= "<tr><td><div id='$screen_name' class='mess-
↳pic' >
        <img src='$profile_image_url' width='48px'
↳height='48px'>
        <br>$screen_name_abv</div>
        <div class='mess-container'>
        <div class='mess-row-text'>$textBody</div></div>
        <div style='clear: both; padding-top:
↳10px'></div></td></tr>";
    }

    return $output;
}
?>

```

Q&A

Q. *Can I request my data from Twitter to be in any other format than XML?*

A. Yes. The API presently supports the following data formats: XML, JSON, and the RSS and Atom syndication formats, with some methods accepting only a subset of these formats.

Q. *In theory, I could use a class object for almost everything in PHP. When should I use a class and when not?*

A. There is no hard and fast rule for when to use a class versus defining functions in separate PHP files and calling them using the include statement. In general, you should use class if what you are creating is going to be shared by other programmers or, as in our case, the functions and procedures are self-contained, numerous and may be replaced in the future

- Q. Do I have to keep my class file in the same directory as my main files?**
- A.** No. In fact, it's better to keep as many files out of your first level (sometimes referred to as root) directory as possible. Standard practice is to create a folder called /Classes and place your file there.

Workshop

Quiz

1. True or False: With PHP, you have to break up your application into as many separate files as you can.
2. Can you make API calls to Twitter without authentication?
3. What does HTTP response code 502 mean?

Quiz Answers

1. False. If you wanted to, you could write a complete application in one file. Files are broken up to make it easier on the programmer to manage the code and to make it easier for other programmers to read and work on the code. As such, breaking key functions, like making calls to Twitter, is easier to modify if it's a single file.
2. Yes and no. There are only a few calls that you can make without providing Twitter with authentication; these include `public_timeline`, `user_timeline`, and search API calls.
3. Code 502 means Bad Gateway. This indicates that the Twitter service is down or cannot respond. It's normally a bad sign.

Exercise

Twitter returns an HTTP response code when a call is made. Create your equivalent of a FailWhale page for the correct HTTP response codes that indicate Twitter is unable to fulfill a request.

HOUR 7

Creating a Twitter API Framework

What You'll Learn in This Hour:

- ▶ What are Twitter API parameters?
- ▶ How to create an API function for Twitter method calls

Twitter API Parameters

In Hour 6, “Building a Simple Twitter Reader,” we created a simple Twitter reader application. There are some shortcomings, however. Remember that we had to manually create the API call to the Twitter servers.

Here is the line just to remind you:

```
$name = 'http://api.twitter.com/1/statuses/user_timeline/BREAKINGNEWS.xml';
```

We created the API call, the value BREAKINGNEWS, and the return type .xml manually in one line. However, to make our system more useful, we need a better way to create this API call. In addition, we need a more dynamic framework for the various API calls we will want to support. However, it's not just API calls we want to support. There is something called *parameters* that can be sent along with an API call.

Not only can we make calls to the Twitter API to get messages, but we can also pass along parameters to refine that call. For example, here are the parameters that Twitter currently accepts for our *statuses/user_timeline* API call (the following are all optional parameters. Not all of these links will work directly in the browser anymore since basic authentication has been disabled):

- ▶ **ID:** Specifies the ID or screen name of the user's timeline. Example:
`http://api.twitter.com/1/statuses/user_timeline/11710512.xml` or
`http://api.twitter.com/1/statuses/user_timeline/perivision.json`.
- ▶ **User_ID:** Specifies the ID of the user's timeline. This is useful when a valid user ID is also a valid screen name. Example: `http://api.twitter.com/1/statuses/user_timeline.xml?user_id=11710512`.
- ▶ **Screen_name:** Specifies the screen name of the user's timeline. This is useful when a valid screen name is also a user ID. Example:
`http://api.twitter.com/1/statuses/user_timeline.xml?screen_name=perivision`.
- ▶ **Since_ID:** Returns statuses with an ID greater than the specified ID.
Example:
`http://api.twitter.com/1/statuses/user_timeline.xml?since_id=2000000` (this example does not work via webpage).
- ▶ **Max_ID:** Returns statuses with an ID less than or equal to the specified ID.
Example:
`http://api.twitter.com/1/statuses/user_timeline.xml?max_id=123456` (this example does not work via webpage).
- ▶ **Count:** Specifies the number of statuses to return. This cannot be greater than 200. Note: The number of statuses returned might be fewer than requested because retweets are stripped out. Example:
`http://api.twitter.com/1/statuses/user_timeline.xml?count=200`.
- ▶ **Page:** Specifies the page of results to retrieve. Keep in mind that there are pagination limits. Example:
`http://api.twitter.com/1/statuses/user_timeline.rss?page=3`.

As you can see, there are a number of parameters at our disposal for tuning the reply we get from the Twitter servers.

Watch Out!

Twitter has been growing so fast that the example numbers you see here for message ID are far larger than simply a five-digit number. At the time of this writing, a typical message ID is as high as 20000000000. That's 20 billion!

Because this is your first time creating a Twitter application, you might be wondering why we need so many parameters. Not every API call has these parameters—some have less, and some have more, but by having these parameters, we can set up some fairly complex and refined interaction with Twitter.

For example, if we keep track of the last message ID a user has seen, we can then request only the newest messages for that user by passing the last-known message ID

plus one. This is very useful for keeping the number of replies from Twitter to a minimum, as well as letting us know if there are any new messages to act on or if Twitter provides an empty set of new messages.

Another example is pagination. If we ask for the last 20 messages, and then the user wants to look back 20 more messages, we can simply pass the value of '2' to the page parameter.

Although the current default reply is 20 messages on most API calls, this is not the case with all calls. If a tweet within the set is a retweet or deleted, it will not appear but will still count toward the returned set. In addition, Twitter could change this value in the future.

**Watch
Out!**

Creating an API Function for Twitter Function Calls

To manage all these parameters, we are going to expand our twitterAPI.php file, but before we show you the code for this API call, let's revisit some code from Hour 6. Remember that we had a function called callTwitter() in our php file main.php:

```
$twitterName = 'http://api.twitter.com/1/statuses/user_timeline/
BREAKINGNEWS.xml';
$twitterRequest = callTwitter($twitterName);
```

The callTwitter() function call would pass our \$twitterName API request to a cURL function in twitterAPI.php. Now we need to update our twitterAPI.php file to support the various parameters we might want to pass to it.

Open the file twitterAPI.php and add the following code at the end of the file but before the '?>':

```
function getUserTimeline($format, $id = NULL, $count = NULL) {
```

Here we begin constructing our API statement. The following 'if' statement checks to see if a user ID was passed. If so, we append the user ID (in this case, 'BREAKINGNEWS') to the end of the API statement. We also append the end of the statement with a '.' and 'xml':

```
    if ($id != NULL) {
        $api_call =
sprintf("http://api.twitter.com/1/statuses/user_timeline/%s.%s", $id,
$format);
    }
    else {
        $api_call =
```

```
sprintf("http://api.twitter.com/1/statuses/user_timeline.%s", $format);
    }
    if ($count != 20) {
        $api_call .= sprintf("?count=%d", $count);
    }
}
```

The next lines append the parameter options. You should already be familiar with '?' and '&' for sending requests over URL, but just to be complete, a '?' after a URL statement indicates to the receiving server that parameters are being passed. The '&' allows us to pass more than one set of parameters.

Here we are using an echo statement so that you can see the API statement we have created. You can remove this later:

```
echo "<h4>$api_call</h4>";
```

Finally, we return our results back to main.php:

```
return callTwitter($api_call);
}
```

Save this file and close it.

Because we have a function to create our API call for us, we no longer need to manually construct our API statement. So, open main.php and make a modification.

Delete these two lines:

```
$name =
'http://api.twitter.com/1/statuses/user_timeline/BREAKINGNEWS.xml';
$twitterRequest = callTwitter('xml', $name);
```

And replace them with these two:

```
$name = 'BREAKINGNEWS';
$twitterRequest = getUserTimeline('xml', $name);
```

Save the file and close it.

Now let's give our new code a test. Open your web browser and load index.php in your development environment. You should see a set of returns, shown in Figure 7.1, that look a lot like what we saw in Hour 6.

We now are making a call to getUserTimeline() and passing two values, as follows:

- ▶ **'xml'**—We want our reply to be in XML format. We could have used JSON if we wanted.
- ▶ **\$name**—We redefined \$name to be just BREAKINGNEWS. We could have used any value Twitter name.



FIGURE 7.1
Example of tweets from BREAKINGNEWS.

Now let's open up main.php again and make a slight change to our code.

Modify this line:

```
$twitterRequest = getUserTimeline('xml', $name);
```

To read:

```
$twitterRequest = getUserTimeline('xml', $name, '3');
```

With this change, we now are making a call to getUserTimeline() and passing three values, as follows:

- ▶ **'xml'**—We want our reply to be in XML format. We could have used JSON if we wanted.
- ▶ **\$name**—We redefined \$name to be just BREAKINGNEWS. We could have used any value Twitter name.
- ▶ **'3'**—Here we are passing our first parameter: the number of tweets we want to get back. If we had left this empty, we would get the default 20 messages back.

We are done with our changes to main.php, so save and close the file.

Next, let's open our Twitter application and see what we get.

You should see something like Figure 7.2. Notice the following line at the top of the page:

`http://api.twitter.com/1/statuses/user_timeline/BREAKINGNEWS.xml?count=3`

FIGURE 7.2
Example of three latest tweets from BREAKINGNEWS.



This is the actual request we are sending Twitter. Using ‘echo’ to see what commands are being sent to Twitter is a great debugging tool. We will leave this in for now, but do remember to remove it once you feel comfortable with what these calls look like.

Let’s add one more API function.

Creating `getPublicTime` API Function

First, let’s see what parameters Twitter supports for this API call. In this case, there is just one: ‘since_id.’

This makes our function call pretty simple. Open up `twitterAPI.php` and add the following lines at the end of the file but before ‘?’:

```
function getPublicTimeline($format, $since_id = 0) {
    $sapi_call =
    sprintf("http://api.twitter.com/1/statuses/public_timeline.%s",
    $format);
    if ($since_id > 0) {
        $sapi_call .= sprintf("?since_id=%d", $since_id);
    }
    echo "<h4>$sapi_call</h4>";
    return callTwitter($sapi_call);
}
```

This should seem quite simple compared to all the parameters we had to worry about with 'user_timeline.'

Save the file and close it.

Now to test this new function call, all we have to do is change one line in main.php. Open main.php and delete (or comment out) the following line:

```
$twitterRequest = getUserTimeline('xml', $name, '3');
```

Then add this line:

```
$twitterRequest = getPublicTimeline('xml');
```

Save your file and close it. Now let's try our application again.

You should see the latest public tweets from Twitter, as shown in Figure 7.3. And all you had to do was change one line!



FIGURE 7.3
Example of latest public tweets.

This is going to be the basis of our framework going forward. We will create a UI that will enable users to click tabs to choose which API calls they want to make. We will get to that in later hours.

Summary

In this hour, we took the first steps in creating the twitterAPI portion of our framework. We also introduced the concept of parameters and a structured way of dealing with them. In addition, we discovered in this hour that we can make different API calls to Twitter by changing one line within our framework.

Q&A

- Q. *Can I use multiple parameters in a call? For example, can I use both `since_id` and `max_id` together?***
- A.** Technically, you can, but it's a bad idea. Twitter sometimes has issues with `since_id` and `max_id`, especially with search, so it's normally a good idea to use only one or the other. There are no known issues with mixing other parameters however, always test first.

Workshop

Quiz

1. True or False: You have to use at least one parameter when making an API call.
2. True or False: The XML return for all Twitter API calls is the same.

Quiz Answers

1. False. If you do not pass any parameters outside the normal API call, Twitter will assume default values for you.
2. False. Although the XML returns are similar and reflect the same basic constructs, there are subtle differences resulting from the ever-evolving API requirements and modifications.

Exercises

1. Try various calls with different parameters to see what you can do. There is a way to confuse and get nothing back from Twitter by asking for conflicting parameters. Try to do this.
2. Toward the end of this hour, we added support for the parameter `since_id` but did not actually use it. Try to find the most recent message ID and then make the call again. Try a number in the future and see what you get as a return.

HOUR 8

Twitter OAuth

What You'll Learn in This Hour:

- ▶ What is a class and why do we use it?
- ▶ What is OAuth? (Briefly)
- ▶ How to create a simple Twitter class object
- ▶ How to add new functions to your Twitter class object
- ▶ How our class deals with Twitter connection errors

What Is a Class and Why Do We Want to Use It?

The rewrite of PHP5 from PHP4 includes updating and upgrading the underlying Object model, which includes the class object. First, what is an object? Object-oriented programming, or OOP, has been around since the 1960s but did not really enter the mainstream until the early 1990s. OOP programming is pretty common in today's languages, and although we are using PHP in this book, most languages you may use to build your Twitter application will likely support the OOP constructs. So, what is an object? Let's look to Wikipedia.

An object is actually a discrete bundle of functions and procedures, all relating to a particular real-world concept such as a bank account holder or hockey player in a computer game. Other pieces of software can access the object only by calling its functions and procedures that have been allowed to be called by outsiders. Isolating objects in this way makes their software easy to manage and keep track of. (http://en.wikipedia.org/wiki/Object-oriented_programming_language)

In our case, the object is a collection of Twitter API calls. The functions and procedures we will isolate will be the individual API calls, as well as a few procedures to deal with Twitter instability. A class is a type of object. Because it's not in the scope of this book to define OOP in depth, we will continue on from here.

The advantage of using a class is that it gives us a clean and clear way to think about our code, as well as ensuring that if you are working with others and thus sharing this code, that no confusion or duplication errors are introduced into the code set.

Now that we know what a class is, and the advantages it gives us, let's have a look at a class for handling the Twitter API.

What Is OAuth?

From the OAuth.net site:

OAuth provides a method for clients to access server resources on behalf of a resource owner (such as a different client or an end-user). It also provides a process for end-users to authorize third-party access to their server resources without sharing their credentials (typically, a username and password pair), using user-agent redirections...

In the traditional client-server authentication model, the client uses its credentials to access its resources hosted by the server. With the increasing use of distributed web services and cloud computing, third-party applications require access to these server-hosted resources.

OAuth introduces a third role to the traditional client-server authentication model: the resource owner. In the OAuth model, the client (which is not the resource owner, but is acting on its behalf) requests access to resources controlled by the resource owner, but hosted by the server. In addition, OAuth allows the server to verify not only the resource owner authorization, but also the identity of the client making the request.

(<http://tools.ietf.org/html/rfc5849>)

How to Register Your Application

To get the OAuth information for your application, you need to register it with Twitter.

Registering an application is fairly simple. Open your browser and go to this address: <http://dev.twitter.com/apps/new>.

Fill out your application as appropriate. If you are not sure, use the following:

Application Type: 'Browser'

Default Access Type: Read & Write

Don't worry; you can change these settings at any time.

After you have filled out the application, you should get something like the screenshot in Figure 8.1.

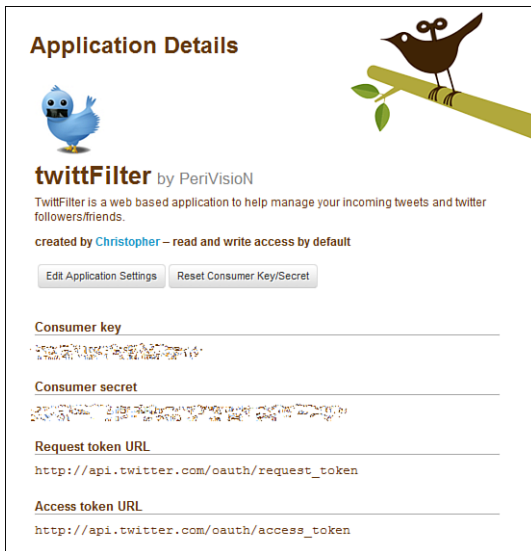


FIGURE 8.1
Twitter
Application
Registration
screen.

Creating the OAuth Twitter Class

So, now that we know what a class is and we know what OAuth is, we need to create an OAuth class. However, creating our own OAuth class is outside the bounds of this book, so instead we will get one that is recommended from the Twitter.com site.

The example we are using here came from a class written by Abraham Williams | abraham@poseurte.ch | <http://abrah.am> twitter name: @abraham.

PHP Library for Working with Twitter's OAuth API

Documentation: <http://wiki.github.com/abraham/twitteroauth/documentation>

Source: <http://github.com/abraham/twitteroauth>

Twitter: <http://apiwiki.twitter.com>

It is not in the scope of this book to explain what OAuth is in detail, much less deconstruct Abraham's code. There are whole books dedicated to the subject. Instead, we are going to walk through the steps needed to get this working for us and then learn how to write API calls within this class.

First, the general overview of how the OAuth API works: the following steps come directly from the documentation within Abraham's code set. I added a few notes where I felt clarification would be useful in our case.

▼ Flow Overview

1. Build TwitterOAuth object using client credentials.

This is where we start. Typically, you would store this information either in the cookie or in a database. In our case, we are not going to support a persistent login, so after the browser is closed, or the OAuth session is closed by some other means, the user will have to reauthenticate.

2. Request temporary credentials from Twitter.
3. Build authorize URL for Twitter.
4. Redirect user to authorize URL.

This URL is found in the `config.php` file.

5. User authorizes access and returns from Twitter.
6. Rebuild TwitterOAuth object with client credentials and temporary credentials.
7. Get token credentials from Twitter.
8. Rebuild TwitterOAuth object with client credentials and token credentials.
9. Query Twitter API.

A more detailed explanation of the preceding flow overview can be found in Abraham's documentation.

Setting Up the twitterOAuth Class

Now that we have an introductory understanding of what OAuth is and how it works, let's take Abraham's code, place it in our development environment, and test it out.

First, if you have not done so already, download his code from github. You should have the following:

images/

darker.png

lighter.png

twitteroauth/

OAuth.php

twitteroauth.php

DOCUMENTATION

LICENSE

README

callback.php

clearsessions.php

config.php

connect.php

html.inc

index.php

redirect.php

test.php

Copy these files into the root directly of your development environment. Now we need to modify a few things:

Open config.php.

In this file, you are going to put your CONSUMER_KEY and CONSUMER_SECRET that we got when registering our application.

The OAUTH_CALLBACK is the location of our root directory. For example, on my local build, it's the following:

```
define('OAUTH_CALLBACK', 'http://localhost/callback.php');
```

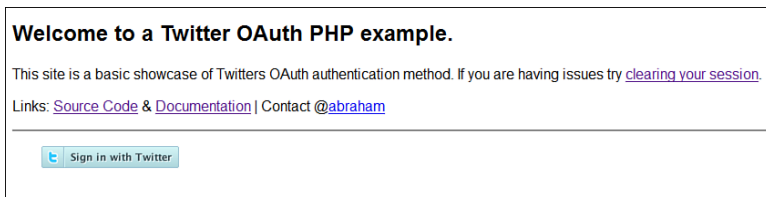
Note: Using localhost as a callback does not always work. If that happens, try using the following workaround: `http://127.0.0.1:8000/twitter_callback`.

If this does not work, try this. You can use bit.ly, a URL-shortening service. Just shorten the url "`http://localhost:3000/twitter_callback`" and register the shortened URL as the callback in your Twitter app. For this method, you have to create another Twitter OAuth app for development so that the callback URLs can differ. (Thanks to Toni for this workaround: <http://www.tonyamoyal.com/2009/08/17/how-to-quickly-set-up-a-test-for-twitter-oauth-authentication-from-your-local-machine/>.)

Save and close.

Now open a web browser and click `index.php`. You should see something like the screenshot in Figure 8.2. If you do not, there is one more thing you may want to check. Sometimes you need to set your system to GMT time. Another possibility is that you need to create a new Twitter application if you were trying to use an existing twitter application.

FIGURE 8.2
Abraham OAuth
login screen.



Click the Sign In with Twitter button, and you will see a message from Twitter asking: An application would like to connect to your account. Select Allow. Next, you should see something like Figure 8.3.

If you got an error saying the curl could not be initiated, be sure you have the `php_curl` extension uncommented. For example:

Fatal error: Call to undefined function `curl_init()` in
C:\xampplite\htdocs\api\twitteroauth\twitteroauth.php on line **199**
`extension=php_curl.dll`

Welcome to a Twitter OAuth PHP example.

This site is a basic showcase of Twitters OAuth authentication method. If you are having issues try [clearing your session](#).

Links: [Source Code](#) & [Documentation](#) | Contact [@abraham](#)

```

stdClass Object
(
    [profile_text_color] => 663B12
    [followers_count] => 928
    [show_all_inline_media] =>
    [follow_request_sent] =>
    [lang] => en
    [geo_enabled] =>
    [created_at] => Tue Jan 01 09:29:37 +0000 2008
    [profile_sidebar_fill_color] => DAECF4
    [description] => I make stuff. Currently I'm making twittFilter.com Check it out and tell me what you think.
    [status] => stdClass Object
        (
            [in_reply_to_user_id] =>
            [text] => Crazy iPad case http://t.co/lbWbhzp via @perivision
            [favorited] =>
            [created_at] => Thu Jan 06 06:38:02 +0000 2011
            [in_reply_to_status_id_str] =>
            [contributors] =>
            [geo] =>
            [in_reply_to_screen_name] =>
            [in_reply_to_user_id_str] =>

```

FIGURE 8.3
Your account
information.

Great. Now that we have our OAuth class in place, we are going to add to it to support our API calls. But first, a little tweak.

Open `index.php` and save it under a new name. Call it `oauth_index.php`. We will be changing `index.php`. This will allow us to create our own `index.php` file. By the way, I used an underscore as a personal habit when I rename a core file from another framework.

Edit `oauth_index.php`

If you do not have it open already, open the file `oauth_index.php`. Change the variable `$connection` to `$twitter` in this line:

(old)

```
$connection = new TwitterOAuth(CONSUMER_KEY, CONSUMER_SECRET,
    $access_token['oauth_token'], $access_token['oauth_token_secret']);
```

(new)

```
$twitter = new TwitterOAuth(CONSUMER_KEY, CONSUMER_SECRET,
    $access_token['oauth_token'], $access_token['oauth_token_secret']);
```

Do the same for the following line:

(old)

```
$content = $connection->get('account/verify_credentials');
```

(new)

```
$content = $twitter->get('account/verify_credentials');
```

Finally, comment out the last line:

(old)

```
include('html.inc');
```

(new)

```
//include('html.inc');
```

We do not have to make this change. However, it's easier to read that you are opening a connection to Twitter. Thus, just for readability, we are changing the variable name. Save and close this file.

We have one more file to edit.

Edit twitteroauth.php

Open the file twitteroauth/twitteroauth.php.

Change the following line:

(old)

```
public $format = 'json';
```

(new)

```
public $format = 'xml';
```

We are making this change because it's easier to read raw XML than raw JSON.

Next, find the function `oauthRequest()`.

We are going to change this line:

```
$url = "{$this->host}{$url}.{$this->format}";
```

To this line:

```
$url = "{$this->host}{$url}";
```

We are doing this because in later hours, we are going to make JSON calls, so we will define when to use XML and when to use JSON in another part of our code.

Now we are going to add some new lines.

At the end of the file, but *before* the last “}”, add the following:

```
#####
#####          New Code #####
#####

function getHomeTimeline($format, $id = NULL, $count = 60, $since
=>= NULL) {
    if ($id != NULL) {
        $api_call =
sprintf("statuses/home_timeline/%s.%s", $id, $format);
    }
    else {
        $api_call =
sprintf("statuses/home_timeline.%s", $format);
return $this->get($api_call);
    }
}
}
```

We have added a function that will make the API call `home_timeline` for us. The reason we want to have a separate function here in the class is to give us maximum flexibility for default actions on an API call, as well as having all our calls in one place in case we need to make any changes or worry about scope. Do not worry about what this function is doing; we will cover it later in the hour.

Save and close.

Great! We have a basic working Twitter class object that we can call. So now, let's write some code to call it. Create a new file and call it `firstcall.php`. In this file, put the following code:

```
<?php
    ini_set('display_errors', '1');
    include 'oauth_index.php';
    global $twitter;
    $messages=$twitter->getHomeTimeline('xml');
    echo 'Home Timeline <br> ';
    print_r($messages);
?>
```

This first line is for reporting errors; this is useful for testing:

```
ini_set('display_errors', '1');
```

This line is where we initiate our OAuth sessions:

```
include 'oauth_index.php';
```

Now we set our `$twitter` object within scope of our function:

```
global $twitter;
```

And finally, we make our first API call using the OAuth class we put in place.

```
$messages=$twitter->getHomeTimeline('xml');
```

Save and close.

Open the file `firstcall.php` in your web browser, and you should see a stream of content from Twitter. Congratulations! You have a working Twitter class.

How to Add New Functions to Your Twitter Class Object

Now that we have a basic class, we can easily add more functions to it. So, let's add a function to allow us to get a user's timeline using the API call `'users_timeline'`.

Edit `twitteroauth.php`

Let's open `twitteroauth.php` again and add a new function. At the end of the file, but *before* the last `"}`", add the following code:

```
function getUsersTimeline ($format, $id = NULL, $since = NULL) {
    if ($id != NULL) {
        $api_call = sprintf("statuses/user_timeline/%s.%s", $id,
➡$format);
    }
    else {
        $api_call = sprintf("statuses/user_timeline.%s",
➡$format);
    }
    if ($since != NULL) {
        $api_call .= sprintf("?since=%s", urlencode($since));
    }
    return $this->get($api_call);
}
```

Next, open `firstcall.php` and find this line:

```
$messages=$twitter->getHomeTimeline('xml');
```

Comment out this line:

```
// $messages=$twitter->getHomeTimeline('xml');
```

Below the line that you commented out, add the following:

```
$messages=$twitter->getUsersTimeline('xml', 'perivision');
```

This new function we created in our Twitter class does look a little like the 'getHomeTimeline' function. Next, we need to look at a few new lines. Let's look at the second and third lines:

```
        if ($id != NULL) {
            $api_call = sprintf("statuses/user_timeline/%s.%s", $id,
➤$format);
        }
```

If we do not pass a value for \$id in our function call, we will skip the following line because the curl object via OAuth already knows who we are when we first created the object. However, we will provide a value for \$id, so let's look at the next line:

```
$api_call = sprintf("statuses/user_timeline/%s.%s", $id, $format);
```

Remember the new line we added in firstcall.php?

```
$messages=$twitter->getUsersTimeline('xml', 'perivision');
```

This line is going to call our new getUsersTimeline function in the Twitter class and pass two values: 'perivision', which is assigned to \$id, and 'xml', which is assigned to \$format. So, the variable \$api_call will have the value of this string: statuses/user_timeline/perivision.xml.

Now if we did not provide a value for \$id, the following lines in our new function will handle that contingency by assigning the viable \$api_call with the string value of statuses/user_timeline.xml:

```
        else {
            $api_call = sprintf("statuses/user_timeline.%s",
➤$format);
        }
```

The Twitter API call user_timeline will provide the timeline of whichever credentials were passed if an ID is not provided.

The next lines in our new function deals with an attribute of the Twitter API that allows us to decide how far back in time to look for messages:

```
        if ($since != NULL) {
            $api_call .= sprintf("?since=%s", urlencode($since));
        }
```

In this case, we did not pass a value for \$since, so this attribute will be ignored and the default value of 20 will be used.

The final line in our new function is something you have already seen before. This line calls the private function that makes the actual API call to twitter.com:

```
return $this->APICall($api_call, true);
```

Save both files and open or refresh the web page firstcall.php. You should get a stream of information, but this time from the Perivision Twitter account.

How Our Class Deals with Twitter Connection Errors

Twitter is a great service, but it's not perfect and sometimes is unavailable. When this first started to happen, Twitter.com provided an error page, sometimes with the picture of a whale trying to be supported by a collection of small birds, a.k.a the FailWhale.

So, what do we do when we try to make a call and Twitter does not give us a reply we expected? We place some code in our Twitter class to deal with this exception.

**By the
Way**

An exception is a generally accepted reserved word that deals with errors encountered by the program. In this case, we are taking exception a bit out of context because a return from Twitter is not an error as much as a message saying the system is not available.

Twitter.com's API provides a variety of replies, as do most HTTP-based services. However, in this hour, we will focus on whether we got a good reply.

Because Twitter provides a status code for each reply, we can check that status value to see if we got a good return. The status code for a good return from Twitter is '200'. So, all we need to do is check to see if we got '200' as a status code.

Insert the following code into twitteroauth/twitteroauth.php below these lines:

```
curl_setopt($ci, CURLOPT_URL, $url);
$response = curl_exec($ci);
$this->http_code = curl_getinfo($ci, CURLINFO_HTTP_CODE);
$this->last_api_call = $url;
```

Code to insert:

```
$http_status = curl_getinfo($ci, CURLINFO_HTTP_CODE);
echo 'status ' . $http_status . '<br>';
if ($http_status!=200) return $http_status;
```

These three lines are fairly straightforward. The first line gets the status code from the curl object. The next line “prints” the value of `$http_status` on the screen. The third line is our ‘if’ statement. If the value of `$http_status` is *not* 200, then we return whatever the value is.

In a proper application, we will want to deal with this exception in a more proper manner; however, in our case, we simply want to know if the call worked.

There is no reason to keep this code because we will deal with this more properly toward the end of the book, but you can keep it for now if you like.

Summary

In this hour, we implemented our OAuth class object that we will rely on and expand upon throughout the rest of this book. Understanding the class object is not only useful to our goals here in building Twitter applications, but good programming in general. We also only touched on the basics of OAuth; however, there are many great sources of detailed information on how OAuth works. You will find many such links on docs.twitter.com.

Q&A

- Q.** *When I first started programming with the Twitter API, I was using a simple username and password, and only recently it stopped working. Will they ever bring it back?*
- A.** Very unlikely. If you have any past code that used ‘Basic Auth’, you would be advised to go back and update that code. With exception to public streams and the various forms of search, access to Twitter requires the use of OAuth only.

Workshop Quiz

1. What is a class object?
2. What is a callback URL?
3. What does the status code 200 mean?

Quiz Answers

1. An object is actually a discrete bundle of functions and procedures, all relating to a particular real-world concept, such as a bank account holder or player info in a computer game.
2. This is the URL Twitter will call when trying to verify your OAuth process.
3. It means the API cURL call you made resulted in a successful return.

Exercises

1. Try changing the search from “perivision” to some other Twitter name.
2. Knowing the HTTP response code is very important when building a Twitter client, try to find where the HTTP response code is returned within the `twitter.class` and display it to the viewer using `echo`.

HOUR 9

Building a Simple Twitter Client, Part I

What You'll Learn in This Hour:

- ▶ How to create a simple framework for our Twitter client
- ▶ How to put tabs in HTML
- ▶ How to support parameters in your framework

Expanding the Index File to Support Tabs

First, we need to add a few tabs to our `index.php` file so that we can select between reading `home_timeline`, mentions, and our direct messages. So, let's create a new file called `header.inc`, and we will put it in the `includes` folder. By the way, it is not necessary to label `header.inc` with a `.inc` extension. This is a convention used by this Author.

To know what page the user wants, we passed that page identifier in the URL link and picked it up using the `$_GET` command:

```
<? $page = $_GET['page']; ?>
```

```
<div id="header">
    <div class="header-bg">
        <div class="header">
            twitterAPI in 24 hours. <br>
```

You can add the following line optionally to help keep track of what hour you are currently working on. Remember to update this as you move from hour to hour:

```
Hour 9 <br>
    </div>
</div>
```

```
<!-- main navigation -->
<div id="nav-box">
    <ul id="main-nav">
```

Here you can see where we set the \$page value:

```
        <li><a href="/" >Home</a></li>
        <li><a href="/?page=mentions"
>Mentions</a></li>
        <li><a href="/?page=direct" >Messages</a></li>
    </ul>
</div>
</div>
```

Save and close this file. Now let's open the file `index.php` and add this file we just created.

After this line:

```
include 'includes/render.php';
```

Add the following line:

```
include 'includes/header.inc';
```

Great. Save the file and close. Load `index.php`, and we should see our new tabs; they should look something like Figure 9.1.

FIGURE 9.1
Tabs now added
to the top of
the web page.



Adding Support for Home Timeline

First, we need to set a switch to call the correct set of functions.

Open `includes/parseTwitter.php` in your editor and add the following code at the end of the file but before the `?>`:

```
function getTwitterData($command) {
    global $twitter;
    switch ($command) {
        case 'timeline':
            { $messages=$twitter->getHomeTimeline('xml'); return
↳call_timeline($messages); }
            break;
        case 'mentions':
            { $messages=$twitter->getMentions('xml'); return
↳call_timeline($messages); }
            break;
        case 'direct':
            { $messages=$twitter->getMessages('xml'); return
↳call_direct($messages); }
            break;
    }
}
```

This is a simple case routine that will not only make the correct API call for us, but also pass the returned data to the correct parsing function. Now we need to define the functions this case will call:

```
function call_timeline($messages){
```

We are going to use a PHP5 call called `SimpleXMLElement()`. Again, it's not in the scope of this book to explain PHP, but if you would like to know more, go to this site: <http://us3.php.net/simplexml>:

```
    $twitterReturn = new SimpleXMLElement($messages);
```

We are again using a counter (`$i`) to make sure our arrays are aligned:

```
    $i=0;
    foreach($twitterReturn->status as $status){
        $updateTime[$i] = parseDate($status->created_at);
        $update[$i] = $status->text;
        $profile_image_url[$i] = $status->user->profile_image_url;
        $screen_name[$i] = $status->user->screen_name;
        $i++;
    }
```

Now that we have Twitter attribute values in arrays, we are going to create another associative array to hold the ones we just defined:

```

        $parsedReturn = array();
        $parsedReturn['updateTime']=$updateTime;
        $parsedReturn['update']=$update;
        $parsedReturn['profile_image_url']=$profile_image_url;
        $parsedReturn['screen_name']=$screen_name;

        return $parsedReturn;
    }

```

Save your file and now open main.php.

Delete everything between the `<?php` and the `?>` and replace them with these two lines:

```

    $twitterReturn = getTwitterData($page);
    echo renderTweets ($twitterReturn);

```

This is the last file to edit. Open the file 'twitteroauth/twitteroauth.php' in your editor and add the following function at the bottom of the file but *before* the last "`}`": This will look a lot like what we used in Hour 7 with the following exceptions:

```

function getHomeTimeline($format, $id = NULL, $count = 60, $since
➡= NULL) {
    if ($id != NULL) {

```

Notice here that we have changed the first annotation to the `$api_call` variable. This is because this part of the API call `http://api.twitter.com/1/` is handled in the OAuth class:

```

        $api_call =
sprintf("statuses/home_timeline/%s.%s", $id, $format);
    }
    else {
        $api_call =
sprintf("statuses/home_timeline.%s", $format);
    }
    if ($since != NULL){
        $api_call .= sprintf("?since_id=%s",
urlencode($since));
        $count=0;
    }

```

Notice that both if statements offer to add a question mark. A more proper programming technique would be to use switch or some control to change the question mark to an ampersand if more than one variable is passed to this function.

However, we want to keep this simple to get the idea across. In addition, it's not a great idea to stack parameters—the results are not always guaranteed:

```
if ($count != 60 AND $count!='') {
    $api_call .= sprintf("?count=%d", $count);
}
// Now we return this API call.
```

Here is the second change. We are now calling 'get' instead of 'call_twitter':

```
return $this->get($api_call);
}
```

Let's save and close our files and give this a go. Load `index.php`, and you should have something like we saw in Figure 9.1 but with different content. We have not added new functionality to our application, but instead we have made our framework a little more flexible.

Adding Support for Mentions

The next API call we are going to make is to get our mentions (*statuses/mentions*). Remember that a mention in Twitter is any tweet that has your Twitter name in it preceded by an ampersand. For example: "Chatted with @perivision at the twitter hackup today." This tweet will then show up in the *statuses/mentions* feed. If we have a look at the parameters that the mention API call accepts, we will see that they are the same as the home API call. These are the current parameters as listed in the Twitter docs (<http://dev.twitter.com/doc/get/statuses/mentions>).

Parameters:

- ▶ **since_id**: Returns only statuses with an ID greater than (that is, more recent than) the specified ID.
Example: `http://api.twitter.com/1/statuses/mentions.xml?since_id=12345`
- ▶ **max_id**: Returns only statuses with an ID less than (that is, older than) or equal to the specified ID.
Example: `http://api.twitter.com/1/statuses/mentions.xml?max_id=54321`
- ▶ **count**: Specifies the number of statuses to retrieve. May not be greater than 200.
Example: `http://api.twitter.com/1/statuses/mentions.xml?count=200`

- ▶ **page:** Specifies the page or results to retrieve. Note: there are pagination limits. (More information on pagination limits can be found here: <http://apiwiki.twitter.com/Things-Every-Developer-Should-Know#6Therearepaginationlimits>.)

Example: `http://api.twitter.com/1/statuses/mentions.xml?page=3`

- ▶ **trim_user:** When set to either true, t or 1, each tweet returned in a timeline will include a user object including only the status authors numerical ID.

Example: `http://api.twitter.com/1/statuses/mentions.json?trim_user=true`

- ▶ **include_rts:** When set to either true, t or 1, the timeline will contain native retweets (if they exist) in addition to the standard stream of tweets.

Example: `http://api.twitter.com/1/statuses/mentions.json?include_rts=true`

- ▶ **include_entities:** When set to either true, t or 1, each tweet will include a node called "entities,". This node offers a variety of metadata about the tweet in a discreet structure, including: user_mentions, urls, and hashtags.

Example: `http://api.twitter.com/1/statuses/mentions.json?include_entities=true`

We can use this to our advantage as we expand our classes, but for now, it's enough to make note of this, but the main point is to understand the variety of options we have available to us. Many of these parameters only became available just recently.

Here is the code for our new call. Open the file 'twitteroauth/twitteroauth.php' in your editor and add the following function at the bottom of the file but before the last "}":

```
function getMentions($format, $page = 0, $since_id=0) {
    $api_call = sprintf("statuses/mentions.%s", $format);
    if ($page) {
        $api_call .= sprintf("?page=%d", $page);
    }
    if ($since_id) {
        $api_call .= sprintf("?since_id=%d", $since_id);
    }
    return $this->get($api_call);
}
```

Notice that we did not write code to support all the parameters listed. However, the code should be clear enough to add new parameters if desired.

Because the XML return for mentions is very much like the return for home_timeline, we can use the same XML parser function as well as the same display. So revisiting our includes/parseTwitter.php file, we can use the same parsing XML for both home_timeline and mentions.

return call_timeline(\$messages);

Let's save our code and close. Load your `index.php` file, select the Mentions tab, and you should see the 20 most recent messages with your Twitter name that you have received. If you have yet to receive a mention before, you may want to ask a friend to send you one or two.

Adding Support for Direct Messages

We have added two API calls, and now it's time to add a third. By now, you should see a pattern to how our framework is going to work. So, let's add the API call for getting direct messages that were sent to us.

Open the file `'twitteroauth/twitteroauth.php'` in your editor and add the following function at the bottom of the file but before the last `"}"`:

```
function getMessages($format, $page = 1, $since = NULL, $since_id = 0) {
    $api_call = sprintf("direct_messages.%s", $format);
    if ($since != NULL) {
        $api_call .= sprintf("?since=%s", urlencode($since));
    }
    if ($since_id > 0) {
        $api_call .= sprintf("%ssince_id=%d", (strpos($api_call,
"?since") === false) ? "?" : "&", $since_id);
    }
    if ($page > 1) {
        $api_call .= sprintf("%spage=%d", (strpos($api_call,
"?since") === false) ? "?" : "&", $page);
    }
    return $this->get($api_call);
}
```

It looks pretty much like the rest of the API calls, doesn't it? However, again, the XML that is returned is slightly different, and again, we need to account for this.

What is different with the mention API call is the return. We cannot use the same parser call as we used for the home API call because the XML is structured a little differently.

Open `'includes/parseTwitter.class.php'` in your editor and add the following code:

```
function call_direct($messages){

    $twitterReturn = new SimpleXMLElement($messages);
    $i=0;
```


Most of this code will look much like the function `call_timeline()` we defined in the beginning of this hour. However, you can see that because the returned XML is structured differently, we need to get our data differently:

```

        foreach($twitterReturn->direct_message as $status){
            $updateTime[$i] = parseDate($status->created_at);
            $update[$i] = $status->text;
            $profile_image_url[$i] = $status->sender->profile_image_url;
            $screen_name[$i] = $status->sender->screen_name;
            $i++;
        }

        $parsedReturn = array();
        $parsedReturn['updateTime']=$updateTime;
        $parsedReturn['update']=$update;
        $parsedReturn['profile_image_url']=$profile_image_url;
        $parsedReturn['screen_name']=$screen_name;

        return $parsedReturn;
    }

```

Let's save our code and close. Load your `index.php` file, select the Messages tab, and you should see the 20 most direct messages to you. If you have yet to receive a direct message, you may want to ask a friend to send you one or two.

Summary

In this hour, we completed Part 1 of building a simple Twitter client. We added three new API calls: `home`, `getMentions`, and `getMessages`. You should now have a pretty good understanding of our Twitter framework and how to add new API calls to the framework.

Q&A

- Q. *Is there a way to ask for only the fields I want from Twitter?***
- A.** Unfortunately, no. Even if all you need is just a few fields, Twitter will provide a feed with everything included. This may change in the future, however.
- Q. *Is there a way to use just one XML parser call instead of having different ones for each API call?***
- A.** Yes. Because the XML returns for the API calls are mostly the same, you could write exception rules for various sets of API calls. However, this will make your code messy and difficult for others to understand. In the interest of simplicity and clarity, we have created separate function calls for various sets of API calls.

Workshop

Quiz

1. True or False: A mention is any time your name shows up in the Twitter stream.
2. True or False: The SimpleXMLElement is the only PHP library call I can use to parse replies from Twitter.

Quiz Answers

1. False. Remember that a mention in Twitter is any tweet that has your Twitter name in it with an 'at' symbol (@).
2. False. There are other XML parsing functions you can use. Also, Twitter can return JSON as well.

Exercise

As you read in this hour, there are a number of parameters you can pass to Twitter to refine what Twitter returns, including `since_id`, `max_id`, `count`, and `page`. Try each parameter one at a time so you can experience how the returns are affected.

This page intentionally left blank

HOOR 10

Building a Simple Twitter Client, Part II

What You'll Learn in This Hour:

- ▶ How to send a message to Twitter
- ▶ How to send a direct versus a normal tweet
- ▶ How to sanitize messages

Updating and Adding New Files to Support Input Text Field

Now that we have a simple Twitter client to read messages, we need to work on making our client application send messages. The good news is that we can still use our current framework with a few additional files. In addition, we will introduce a few JavaScript calls to support user interaction.

First, we need to add to our `index.php` file to allow a user to type in a message.

Edit `index.php`

Open the file `index.php`, and after this line:

```
<title>twitterAPI 24</title>
```

Add the following line:

```
<script type='text/javascript' src='js/base.js'></script>
```

Now we need to add a few more files to include; we will create them in a moment.

After the following line:

```
include 'includes/header.inc';
```

Add the following two lines:

```
include 'includes/createMessage.php';
include 'sendMessage.php';
```

Notice that we did not place `sendMessage.php` in the include folder. That is because this file will be called by JavaScript and, thus, for simplicity, needs to live in the same folder as our OAuth code.

Create createMessage.php

Let's add a new file. Create a new file called `'createMessage.php'` within the includes folder and type in the following code:

```
<div class="sendMessage">
  <div class="messageTextBox">
    <form name="sendMessForm" method="get" action="" class="asholder">
      <label id="prefex"></label>
      <textarea id="testinput" name="sendMessField" class="inputbox"
        rows="2" cols="80" ></textarea>
      <br>
      <input class="inputbox" type="button" value="Send"
        onmouseup="sendMessage()" />
      </font>
    </form>
    <div id="serverMessages">
      </div>
  </div>
</div>
```

Then save this file and close it.

Edit main.css

Now let's open `css/main.css` and add two entries.

After the following lines:

```
.tweet {
  padding:0 10px;
}
```

Add the following lines:

```
.sendMessage {
  padding: 15px 10px 0px;
}
```

```
.messageTextBox {
  width: 600px;
}
```

Then save this file and close it.

Create base.js

Now that we have a text field and a Send button, we need some JavaScript code to catch the button click and pass the message to a new PHP file we will create soon. If you are not familiar with JavaScript, this might look a bit scary, but all we are really doing here is making a call to our server, getting the response, and putting that response on our page. This is known as an AJAX call:

```
<!--
```

```
function sendMessage(){
  message = document.getElementById("prefex").innerHTML;
  message += document.sendMessForm.sendMessField.value;
  url='sendMessage.php?message='+message;
  document.sendMessForm.sendMessField.value='';
  document.getElementById("prefex").innerHTML='';
  callPage(url, 'serverMessages');
}
```

```
//#####
#####
```

```
var k=0; req = Array();
```

```
function callPage(pageUrl, divElementId, loadingMessage, pageErrorMessage) {
  if(!loadingMessage || loadingMessage=='') loadingMessage = "ContentSet is
loading, Please Wait...";
  if(!pageErrorMessage || pageErrorMessage=='') pageErrorMessage="Error in
Loading page />";
  document.getElementById(divElementId).innerHTML = loadingMessage+'Calling
twitter...';
  try {
    req[k] = new XMLHttpRequest(); /* e.g. Firefox */
  } catch(e) {
    try {
      req[k] = new ActiveXObject('Msxml2.XMLHTTP'); /* some versions IE */
    } catch (e) {
      try {
        req[k] = new ActiveXObject('Microsoft.XMLHTTP'); /* some versions IE */
      } catch (E) {
        req[k] = false;
      }
    }
  }
}
```

```

    }
    req[k].onreadystatechange = function() {responsefromServer(divElementId,
➤pageErrorMessage);};
    req[k].open('GET',pageUrl,true);
    req[k].send(null);
}

function responsefromServer(divElementId, pageErrorMessage) {
    var output = req[k].responseText;
    if(req[k].readyState == 4) {
        if(req[k].status == 200) {
            output = req[k].responseText;
            document.getElementById(divElementId).innerHTML = output;
        } else {
            document.getElementById(divElementId).innerHTML =
➤pageErrorMessage+'\n'+output;
        }
    }
}
}

-->

```

Now let's save and close the file.

Create sendMessage.php

This next file are going to create will be placed in the home or root directory of our development environment. Create a new file called sendMessage.php and add the following code:

```

<?php

include 'oauth_index.php';

$message=$_GET['message'];

$output=$twitter->updateStatus($message);
$twitterReturn = new SimpleXMLElement($output);
echo 'Twitter said ';
print_r ($twitterReturn);

?>

```

Save and close this file.

Sending a Message to Twitter

Finally, we need to have a file to catch the newly created message and make the API call. Open twitteroauth/twitteroauth.php.

At the end of the file, but before the “} ?>”, add the following lines:

```
function updateStatus($status) {
    $status = urlencode(stripslashes(urldecode($status)));
    $api_call = sprintf("statuses/update.xml?status=%s",
    ↪$status);
```

We are going to add an echo statement here so that you can see the structure of your API call, as we have done on other functions:

```
        echo "<h4>$api_call</h4><br>";
    return $this->post($api_call);
}
```

Save and close this file.

Open index.php, and let's try sending our first message. Type something into the text box and hit Send.

If you see something like Figure 10.1, you have successfully sent your first tweet from an API call!

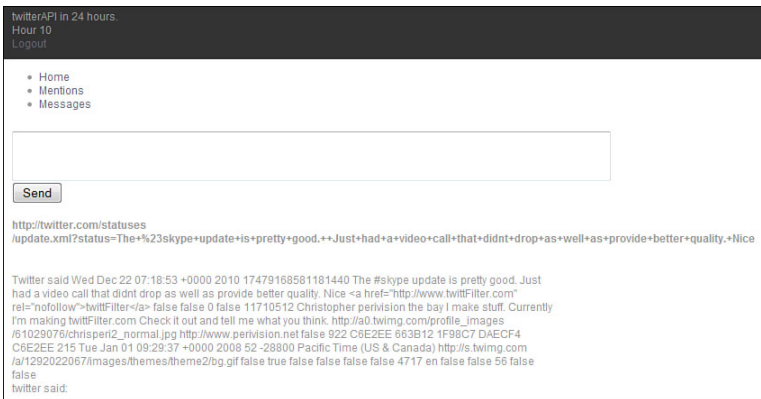


FIGURE 10.1
Raw return from sending a message.

API Call for Direct Messages

Sending a direct message is different from sending a regular message and thus requires a different API call. As such, we need to know when the user wants to send a direct message and when the user wants to send a regular message. There are a number of ways to approach this: One way is to simply look for a message that starts with the letter “d” followed by a space as an indicator that the user wants to send a direct message. Why is this? Because in the early days of Twitter, when it was SMS only, the only way to send a direct message was to use the letter “d” and a space followed by the Twitter name the message is to be delivered to, a space, and then your message. Even now, we can type d <twittername> and a message, and

Twitter will automatically treat this as a direct message. There is a downside to this, however. There are only 140 characters within a tweet, so we lose those characters to the “d,” the space, the Twitter username, and another space. This may not seem like much, but it adds up, especially if the Twitter user’s name is long.

To get around this, the Twitter API provides us with a separate API call for direct messages. Because this is a separate process, we will explore this API in Hour 12, “Direct Messages.” However, depending on what your Twitter client is going to do, you may want to look for the “d” shortcut pattern and extend the number of characters users can send in their message if you plan to implement a character counter in your application.

Sanitizing Messages

Although our current implementation works, we need to account for a few things. Because we are sending information over the Web back to our application, we need to make sure that the characters we use are web safe. So, let’s add the following code to our JavaScript to sanitize the message before we send it to our `sendMessage.php` file.

Edit `base.js`

Open `base.js`, and after the following line:

```
message += document.sendMessForm.sendMessField.value;
```

Add this line:

```
message=encodeURIComponent(message);
```

Just to be clear, the `encodeURIComponent` function will replace certain non-web safe symbols with a URI code. This is much like the `string replace` function here, `$message = str_replace("%23", "#", $message)`, but for all non web-safe characters.

Save and close this file.

Great! We now have a working Twitter client that can read messages as well as send messages. We also implemented AJAX within our application. And it’s only Hour 10.

Summary

In this hour, we have finished up the basics for our simple Twitter application and set the foundation for further expansion through the rest of book.

We also created our first AJAX call for sending messages, and we created a PHP file for responding to the requests that come from our JavaScript AJAX call.

Q&A

Q. How do you send a direct message from the message box in Twitter?

A. Type d, then a space, and then your message. This is one of many shortcuts that you can use with Twitter.

Q. Why does Twitter have shortcuts?

A. Twitter started out as a SMS-only communication system. Using a single letter was a simple way to send Twitter a command or indicate the type of message being sent.

Workshop

Quiz

1. True or False: If, for some reason, a non-web safe character is sent to Twitter, it will return an error.
2. True or False: If you wanted to, you could send a message through the normal API and place the letter d, a space, and then a valid username, and it would work just as well as using the direct message API call.

Quiz Answers

1. False. Most likely it will simply truncate the message and execute the requested API request. **However**, ALWAYS be sure that any message you send to Twitter is web-safe.
2. True for now. Although this can be done (there is no guarantee this will continue to work), it's not a good idea. However, many users still use short cuts, so it's in your best interest to look for a message that starts with a d, a space, and a valid username, and then send it as a direct message through the correct API call.

Exercise

In this hour, we simply displayed what Twitter returns after sending a direct message. Try displaying only the message you sent to Twitter. Hint: You would parse return in a similar manner as you have parsed other returns from Twitter.

This page intentionally left blank

HOOR 11

Expanding Our Client for More API Calls

What You'll Learn in This Hour:

- ▶ How to add multiple tabs to our UI
- ▶ More statuses API calls
- ▶ How to send a retweet through the API
- ▶ The various ways to handle retweets

Types of API Method Calls

Congratulations. This is Hour 11, and we have already created a Twitter client application. But we have only scratched the surface of what can be done with Twitter through the API set. We will not go through every API call in detail, because that would make this book so large we would have to break into multiple volumes. But we will list and discuss them all so that you can get an idea of what options you have for the Twitter application you would like to build. At the time of this writing, the Twitter development documents break up the API sets into the following groups: (<http://dev.twitter.com/doc>)

- ▶ Timeline
- ▶ Status
- ▶ User
- ▶ List Members
- ▶ List Subscribers
- ▶ Direct Message

- ▶ Friendship
- ▶ Social Graph
- ▶ Account
- ▶ Favorite
- ▶ Notification
- ▶ Block
- ▶ Spam
- ▶ Saved Searches
- ▶ OAuth
- ▶ Local Trends
- ▶ Geo
- ▶ Help
- ▶ Search

Quite a lot, isn't there? A vast majority of our time will be spent around the Timeline, Status, and Search methods. As such, we are going to spend this hour updating our User Interface (UI) so that we can support more API calls. By the end of this hour, we will have all the Timeline methods covered.

Adding Tabs to Our UI

To make it a bit easier for us to add new API calls, we need more buttons in our UI. We will also have these tabs organized such to reflect how Twitter currently has their API reference documents organized as well as the order of what we will be building in this book. So, let's open `includes/header.inc` and expand the code.

Edit `header.inc`

Erase everything after the line `<!-- main navigation -->` and add the following lines:

```
<div id="nav-box">
    <div class="nav-box">
    <ul id="globalnav">

    <li><a href="#" class="here">Timeline</a>
    <ul>
    <li><a href="#">Public</a></li>
    <li><a href="?page=timeline" >Home</a></li>
```

```

        <li><a href="?page=mentions" >User</a></li>
        <li><a href="?page=direct" >Mentions</a></li>
    <li><a href="?page=rt_by_me">Retweeted By Me</a></li>
    <li><a href="?page=rt_to_me">Retweeted To Me</a></li>
    <li><a href="?page=rt_of_me">Retweeted Of Me</a></li>
</ul>
</li>
<li><a href="#">List</a>
</li>
<li><a href="#">User</a>
</li>
<li><a href="#">Direct Message</a>
</li>
<li><a href="#">Favorites</a>
</li>
<li><a href="#">Search</a>
</li>
<li><a href="#">Saved Search</a>
</li>
</ul>
        </div>
    </div>
</div>

```

Save and close. Now let's create a new CSS file to support these tabs:

Create nav.css

Create a new file in your css folder called nav.css and add the following code:

```

#globalnav {
    position:relative;
    float:left;
    width:100%;
    padding:0 0 1.75em 1em;
    margin:0;
    list-style:none;
    line-height:1em;
}

#globalnav LI {
    float:left;
    margin:0;
    padding:0;
}

#globalnav A {
    display:block;
    color:#444;
    text-decoration:none;
    font-weight:bold;
    background:#c0deed;
}

```

```
        margin:0;
        padding:0.25em 1em;
        border-left:1px solid #fff;
        border-top:1px solid #fff;
        border-right:1px solid #aaa;
    }

    #globalnav A:hover,
    #globalnav A:active,
    #globalnav A.here:link,
    #globalnav A.here:visited {
        background:#a8d0e3;
    }

    #globalnav A.here:link,
    #globalnav A.here:visited {
        position:relative;
        z-index:102;
    }

    /*subnav*/

    #globalnav UL {
        position:absolute;
        left:0;
        top:1.5em;
        float:left;
        background:#a8d0e3;
        width:100%;
        margin:0;
        padding:0.25em 0.25em 0.25em 1em;
        list-style:none;
        border-top:1px solid #fff;
    }

    #globalnav UL LI {
        float:left;
        display:block;
        margin-top:1px;
    }

    #globalnav UL A {
        background:#a8d0e3;
        color:#fff;
        display:inline;
        margin:0;
        padding:0 1em;
        border:0
    }

    #globalnav UL A:hover,
    #globalnav UL A:active,
    #globalnav UL A.here:link,
```

```
#globalnav UL A.here:visited {
    color:#444;
}
```

Save and close. We have one more file to go. We need to update index.php to see the new css file.

Edit index.php

After this line:

```
<link href="css/main.css" rel="stylesheet" type="text/css" media="screen" />
```

Add the following line:

```
<link href="css/nav.css" rel="stylesheet" type="text/css"
media="screen" />
```

Save and close. Now open index.html in your browser, and you should see something like Figure 11.1.

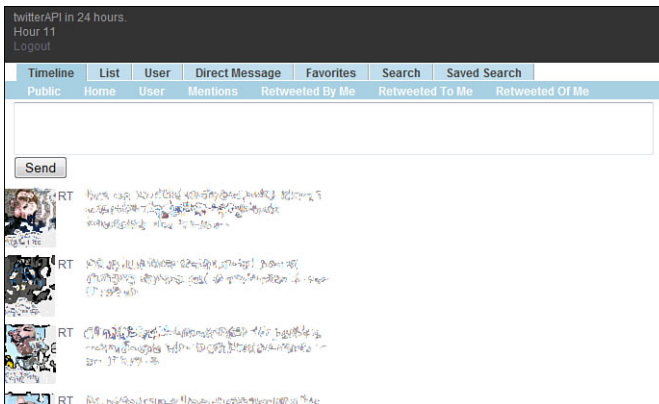


FIGURE 11.1
New tabs are now added to the top of the web page.

New Timeline API Calls: Retweeted

We are going to create three new API calls (retweeted by me, to me, and of me), and because of how we created our code structure, you will find that it will be very easy because we are cutting and pasting code from functions we have already created. However, let's explore these three calls first.

Retweeted by Me

Depending on the Twitter account you are using and the type of client application you have used, you may or may not get returns from this API call. That is because the `retweeted_by_me` API call relies on the application to send a retweet using a certain format. We will cover this later. If the application you have been using to retweet does not use the retweet API call, you may not see any returns, or they may be older returns. If this is the case, the best thing to do is log in to [Twitter.com](https://twitter.com) and retweet a message; then return to your code. You should see your retweeted message.

Retweeted to Me and of Me

The `retweeted_to_me` and `retweeted_of_me` API calls are very similar. So much so, you may find the same display when you click between the two buttons. That is because a message that is `retweeted_to_me` are retweets from people in your friends list. Messages that are `retweeted_of_me` are from anyone—a slight but important difference.

Now let's add a few new API calls.

Edit `parseTwitter.php`

We have added new tabs to our UI. Now we need to account for the new tabs in our parsing statement. Open `parseTwitter.php`, and after the following lines:

```

    case 'direct':
        { $messages=$twitter->getMessages('xml'); return
➤call_direct($messages); }
        break;

```

Add these lines:

```

    case 'rt_by_me':
        { $messages=$twitter->getRTByMe('xml'); return
➤call_timeline($messages); }
        break;
    case 'rt_to_me':
        { $messages=$twitter-> getRTtoMe ('xml'); return
➤call_timeline($messages); }
        break;
    case 'rt_of_me':
        { $messages=$twitter-> getRTofMe ('xml'); return
➤call_timeline($messages); }
        break;
    }

```

As you can see, we have extended our case statement. Also, because the XML returns of the new API calls are similar to our `home_timeline` call, we will simply reuse the same function call to display our returns.

Save and close.

Edit twitteroauth.php

Now that we have the new API calls in `parseTwitter`, we need to add functions to `twitteroauth.php` to catch those calls.

Open `twitteroauth.php` and add the following lines at the end of the file, but before `}?>:`

```
function getRTByMe() {
    $api_call = ' statuses/retweeted_by_me.xml';
    return $this->get($api_call);
}

function getRTToMe() {
    $api_call = 'statuses/retweeted_to_me.xml';
    return $this->get($api_call);
}

function getRTOfMe() {
    $api_call = 'statuses/retweets_of_me.xml';
    return $this->get($api_call);
}
```

If these functions seem a lot like our `getHomeTimeline` function call, it's because they are. Many API calls to Twitter are similar in fashion, and we get to take advantage of this. However, what is different is that we are not using the `sprint()` function anymore. Although we could have continued to use the `sprint()` functions, the extra code will only make reading more confusing as well as cause extra typing for you.

Save and close this file.

Now let's give it a try. Load `index.php` and click the three new function calls we just created. You have three new functional API calls!

New Status API Calls: Retweeted

The retweet API call is not required because it's still a convention to prepend a message with the letters 'rt' and the tweeter's name. However, this introduces extra characters, and if the original message is close to 140 characters, the addition of 'rt' and the tweeter's name could put it over. In addition, if you wanted to trace back a message that generated the retweet in the first place, you would have to search all tweets with this message or keep a log yourself. To deal with this issue, as well as make it more feasible to display related tweets inline, the Twitter API has a method called `retweet` that has one required input: the ID of the tweet to be retweeted.

Retweet

Now this is tricky; in order for our user to retweet a message, we need to offer a Retweet button, which needs to pass the ID of that message back to the server. There are two ways to do this. We can create a dumb button that makes a unique call back to the server with the message ID and thus load a new page, or we can make an AJAX call. The AJAX call is a bit more involved but provides a far superior user experience.

We need to add a few lines of code to our framework, so we are going to add a case switch here.

Edit `sendMessages.php`

After this line:

```
$message = $_GET['message'];
```

Add the following:

```
$id = $_GET['id'];
$command = $_GET['command'];
    switch ($command) {
        case '':
            { echo 'Err: No command found'; }
            break;
        case 'update':
            { $output=$twitter->updateStatus($message); }
            break;
        case 'retweet':
            { $output=$twitter->retweet('xml', $id); }
            break;
    }
```

Save and close.

Edit `twitteroauth.php`

Next we need to pass the retweet request on to Twitter, so let's edit `twitteroauth.php`. Open `twitteroauth.php` and add the following lines at the end of the file, but before `}?>':`

```
function retweet($format, $id = NULL) {
    if(!$id) { return 'Err; cannot retweet. no id found'; }
    $api_call = sprintf('statuses/retweet/%s.%s', $id,
➔$format);
    return $this->post($api_call);
}
```

Edit parseTwitter.php

Once we get our return from Twitter, we need to parse. Open parseTwitter.php:

After this:

```
foreach($twitterReturn->status as $status){
    $updateTime[$i] = parseDate($status->created_at);
    $update[$i] = $status->text;
```

Add this:

```
$id[$i] = $status->id;
```

After this:

```
$parsedReturn = array();
$parsedReturn['updateTime']=$updateTime;
$parsedReturn['update']=$update;
```

Add this:

```
$parsedReturn['id']=$id;
```

Edit render.php

And of course, we need to modify how we render so that we can provide the user with a retweet button. We are just using letters, here, but you could replace this with an icon. Open render.php:

After this:

```
for($i=0; $i<$num; $i++){
    $updateTime = $parsedReturn['updateTime'][$i];
    $update = $parsedReturn['update'][$i];
```

Add the following line:

```
$id = $parsedReturn['id'][$i];
```

After this:

```
$output.= "<tr><td><div id='$screen_name' class='mess-
pic' >
    <img src='$profile_image_url' / width='48px'
height='48px'>
    <br>$screen_name_abv</div>
```

Add the following lines:

```
<div class='mess-actions'>
<a href='#' onmouseup="\retweet('$id')\">RT</a></div>
```

Edit main.css

We need to make some space for our retweet button, so lets make some via our CSS file. Open main.css:

Add this at the end of the file:

```
.mess-actions{
float:left; padding-left:3px; width:10px;
}
```

Edit base.js

We will add a function in our javascript file to catch the users click. Add this at the end of the file of base.js:

```
function retweet(mess_id){
    alert('retweet '+mess_id);
    url='sendMessage.php?command=retweet&id='+mess_id;
    callPage(url, mess_id);
}
```

If you did not already know, the JavaScript function 'alert()', which is built in to JavaScript, will pop up a window and stop any further execution of JavaScript code. This serves as a very useful debugging tool. If everything went well with this call, you can comment out or remove this line.

retweets/id, id/retweeted_by, id/retweeted by/ids

We are not going to write code for the next three API calls, but I do want to cover them briefly. You can find this and more information in the Twitter docs. (<http://dev.twitter.com/doc/get/statuses/retweets/:id>)

retweets/id—<http://api.twitter.com/1/statuses/retweets/id.format>

Retweets give up to 100 retweet status (messages) based on a given status ID. In other words, if we pass a status id, as we did to create a retweet to the status/retweets API call, we would get back up to 100 statuses that are retweets of the original status. The value of this is not the actual status, because they would all be the same, RT @<screen_name> <original message>, but instead the list of people who retweeted the original tweet.

id/retweeted_by—http://api.twitter.com/1/statuses/id/retweeted_by.format

Show user objects of up to 100 members who retweeted the status represented by ID.

id/retweeted_by/ids—http://api.twitter.com/1/statuses/id/retweeted_by/ids.
format

Note `ids` here refers to the support of multiple ids that can be passed

Show user IDs of up to 100 users who retweeted the status represented by ID.

Keep in mind this will only return users' IDs.

Summary

In this hour, we expanded the UI of our application to support tabs, which we will need as we include more and more API calls. We also introduced the concept of retweeting. The various ways to read a retweet include messages retweeted by the user, messages retweeted to the user, and retweets about the user.

We also dealt with passing a message ID back to our script from the client using JavaScript, which we will see again in coming hours.

Q&A

Q. Is 'RT' a shortcut like 'd'?

A. No. RT was never officially supported by Twitter. It was a convention that many people and later Twitter programmers, like you, implemented.

Q. Because the difference between *retweeted_to_me* and *retweeted_of_me* is so slight, do I really need to support both in my application?

A. It depends on what you are trying to accomplish, but for most basic applications, you could get away with just `retweeted_of_me`.

Workshop

Quiz

1. What are the three different types of retweeted messages?
2. Because 'RT' is only a convention, can I ignore it in my application?

Quiz Answers

1. The three types are `retweeted_by_me`, `retweeted_to_me`, and `retweeted_of_me`.
2. No, it's only a convention; however, many people and older programs still follow this convention, so you may want to look for a message starting with or containing a 'RT' and then decide what you should do based on what your application is trying to accomplish. Caution, many users will us RT but change the message somewhat.

Exercise

Although we did not code an example for any of the Retweet API methods, pick one and implement it into our sample application.

HOUR 12

Direct Messages

What You'll Learn in This Hour:

- ▶ Send direct messages through the API
- ▶ Test for the existence of friendships between users
- ▶ How to delete direct messages using destroy method

Sending a Direct Message

Although we have briefly discussed direct messages in Hour 10, in this hour we add support for sending direct messages, as well as a few other methods. We will also look at another set of API calls, called friends, in order to know if we can send a direct message in the first place.

Adding UI Elements for Direct Messages

However, the first thing we will do is revisit reading our direct messages. Because we already have this in our API set, we just need to populate the button on our UI.

Edit index.php

After this line:

```
<meta charset="utf-8">
```

Add this line:

```
$nav = $_GET['nav'];
```

Edit header.inc

After these lines:


```

        <li><a href="/?page=rt_of_me">Retweeted Of Me</a></li>
    </ul>
<? } ?>
</li>

```

Add the following lines:

```

        <li><a href="/?nav=direct" <? if($nav=='direct') echo
➤ 'class="here"'; ?> >Direct Message</a>
        <? if($nav=='direct') { ?>
            <ul>
                <li><a href="/?page=direct" >Direct Message</a></li>
            </ul>
        <? } ?>
    </li>

```

Testing if a Direct Message Can Be Sent

Now that we have that in place, let's create an API call that deals with sending direct messages.

In Twitter, you can send a direct message only to someone who follows you. If the Twitter user does not follow you, you will get back an error. So, before we send our direct message, we should check to see whether the Twitter user is following us. That means a new API call: `friendships/exists`.

This will be a new type of API call that we have not seen before. This call returns a simple true or false. We pass two parameters: the Twitter name of user A and the Twitter name of user B. If Twitter user A follows user B (A->B), Twitter will return a value of true. If not, it returns a value of false.

There are a few ways to determine if a direct message can be sent or not:

- ▶ Get a list of IDs of every follower a user has. Then check to see if the Twitter ID displayed in the message stream is contained within this list. The only problem with this option is what do you do if the user has a very large number of followers? The current paging limit is 5,000 IDs per page. This could become cumbersome if the user has a lot of followers: 20,000 followers is not that rare.
- ▶ Get a list and store it in a database. This would be the more typical solution for a high-level functioning Twitter application. You would go through the list of pages of follower IDs, store them in a database, and then make calls as each new Twitter ID is found in the incoming stream.
- ▶ Check one at a time at the moment the user tries to create a message. That is the approach we are going to use here. This keeps the API calls to a

minimum and is the easiest to code as a book example. However, this is not practical if we need to determine a large number of friendship relationships at once. It also has the drawback that you cannot indicate to the user ahead of time if someone can receive a direct message or not.

- ▶ Send the message and alert the user if an error has occurred. This is similar to the previous point but instead of testing if a friendship exists, you look for an error from Twitter. This is not recommended since it reflects poor coding techniques as well as depends on the Twitter error code not changing.

Because we are going to check the friendship status only after the direct message button or the letters DM in our case were clicked, we will have two events to manage. First, we are going to populate the message box with the “d” shorthand and the Twitter name we want to send the direct message to. Second, within the `sendMessage.php` file, we will detect for that shorthand and use the proper API method. The reason we want to do this is to support the older shorthand of using the letter d and the username that some Twitter users still employ. We are using this approach to help illustrate how to deal with short hand. In a typical application, you would use other means to indicate that a message is a direct message.

Here is what an example call would look like:

```
http://api.twitter.com/1/friendships/exists.xml?user_a=tweetapi24&user_b=
↳perivision
```

We need two Twitter names to make this work: the name of the user and the name of the person who we want to send the direct message to. We will pass those names to the API, and if we get a true value back, we will allow the user to send a message. If we get back a false value, we will pop up a warning that the direct message will not go through. Of course, we could accomplish the same thing by simply trying to send the message and look for the error response back from Twitter, but this is a book on Twitter API programming, so we are going to do it the more formal way.

Adding Direct Message API Support

In this section, we add direct message API support.

Edit `Render.php`

We are going to add another mouse rollover action to our UI. We will add the letters “DM” under the “RT” we created in Hour 11:

```
global $twitterName;
```

After this line:

```
function renderTweets ($parsedReturn){
```

Add this line:

```
global $twitterName;
```

Edit this line:

```
<a href='#' onmouseover="\retweet('$id')\">RT</a></div>
```

To read:

```
<a href='#' onmouseover="\retweet('$id')\">RT</a>
```

This will remove “</div>”.

Now add the following under the line we just edited:

```
<a href='#' onmouseover="\direct('$screen_name' , '$twitterName')\">DM</a>
</div>
```

Open index.php, and we should now see a ‘DM’ under our ‘RT’ in the Twitter stream.

Edit base.js

The next step is to populate the message box with the letter “d” and the screen name of the person we want to send the direct message to. We also want to check to see whether the user is following us. We will initiate both actions in this JavaScript function call.

Add the following function after the ‘retweet()’ function:

```
function direct(nameA, nameB){
    document.getElementById("serverMessages").innerHTML =
    ↳'Checking... ';
    document.sendMessForm.sendMessField.value='d '+nameA+' ';
```

Here we are defining our URL to call sendMessage.php and pass the two names we want to compare:

```
url = 'sendMessage.php?command=testFriendship&parameterA=
↳'+nameA+'&parameterB='+nameB;
    callPage(url, 'serverMessages');
}
```

Edit sendMessage.php

Now we need to catch our new DM request in the sendMessage.php file. So, let's add a few more lines of code.

After this line:

```
$id = $_GET['id'];
```

Add the following lines:

```
$parameterA = $_GET['parameterA'];
$parameterB = $_GET['parameterB'];
```

Notice that we have made our variables more generic; that will allow us to use these over again for other calls. We have chosen to make these variables generic for illustrative reason for this book.

Next, we need to add a new case to our switch.

After these lines:

```
case 'retweet':
    { $output=$twitter->retweet('xml', $id); }
    break;
```

Add the following lines:

```
case 'direct':
    { $output=$twitter->sendDirectMessage('xml', $id,
↳$message); }
    break;
case 'testFriendship':
    { $output=$twitter->friendshipExists($parameterA,
↳$parameterB);
        if(stristr($output, 'false')) echo '<h3>This
↳person does not follow you. They may not receive the message</h3>';
        return; }
    break;
```

Edit twitteroauth.php

The friendships/exists API method we are going to call will be a bit different from the other API calls we have been making, mostly because all we need is a true or false based on two parameters.

At the end of the file, but before the last brace }, add the following lines:

```
function friendshipExists($a, $b){
    $api_call = " friendships/exists.xml?user_a=$a&user_b=$b";
    return $this->get($api_call);
}
```

The friendships/exists is not the only API call we could have made here. We could have called friendships/show. Here is an example:

```
http://api.twitter.com/1/friendships/show.xml?source_id=3191321&target_scre
en_name=noradio
```

The reply would be the following:

```
<relationship>
-
<target>
<screen_name>ev</screen_name>
<id_str>20</id_str>
<followed_by type="boolean">true</followed_by>
<following type="boolean">true</following>
<id type="integer">20</id>
</target>
-
<source>
<want_retweets nil="true" />
<marked_spam nil="true" />
<all_replies nil="true" />
<screen_name>noradio</screen_name>
<id_str>3191321</id_str>
<blocking nil="true" />
<followed_by type="boolean">true</followed_by>
<notifications_enabled nil="true" />
<can_dm type="boolean">true</can_dm>
<following type="boolean">true</following>
<id type="integer">3191321</id>
</source>
</relationship>
```

So, although we get more information this way, we do not get anything that we would act upon for our requirement. By using friendships/exists, we can use an 'if(stristr)' statement and look for true or false.

Okay, let's give it a try. Open index.php, and click DM from any Twitter name that you know does not follow you. You should get something like the screenshot in Figure 12.1.

Now let's try sending a direct message. Click the DM of a Twitter user who you know is following you and send them a message.

You should get something like the screenshot in Figure 12.2.

There are few other direct message API calls available to us, so let's add them to our application.

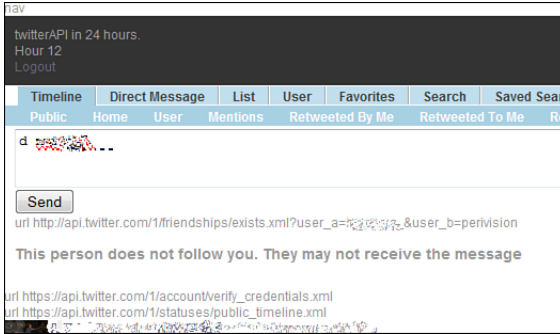


FIGURE 12.1 Screenshot showing the Not Following warning.

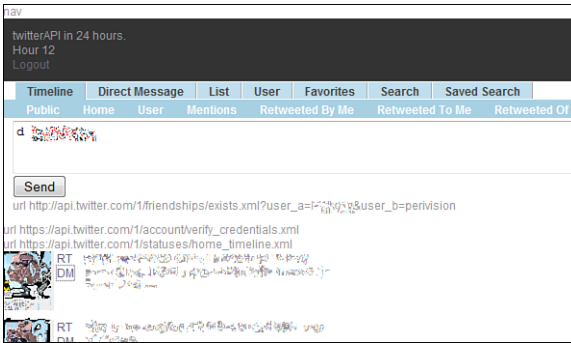


FIGURE 12.2 Screenshot showing the Following confirmed message.

Adding More Direct Message API Support

Now let's add more direct message API support.

Edit twitteroauth.php

After this code:

```
function getMessages($format, $id = NULL, $count = 60, $since =
↳NULL) {
    $api_call = sprintf("direct_messages.%s", $format);
    if($since != NULL){
        $api_call .= sprintf("?since_id=%s",
↳urlencode($since));
        $count=0;
    }
    if ($count != 60 AND $count!='') {
        $api_call .= sprintf("?count=%d", $count);
    }
    return $this->get($api_call);
}
}
```

Add the following lines:

```

function getMessagesSent($format, $id = NULL, $count = 20, $since
↳= NULL) {
    $api_call = sprintf("direct_messages/sent.%s", $format);
    if($since != NULL){
        $api_call .= sprintf("?since_id=%s",
↳urlencode($since));
        $count=0;
    }
    if ($count != 20 AND $count!='') {
        $api_call .= sprintf("?count=%d", $count);
    }
    return $this->get($api_call);
}

```

Edit parseTwitter.php

After this code:

```

    case 'direct':
        { $messages=$twitter->getMessages('xml'); return
↳call_direct($messages); }
        break;

```

Add the following lines:

```

    case 'directSent':
        { $messages=$twitter->getMessagesSent('xml'); return
↳call_direct($messages); }
        break;

```

Edit header.inc

After this code:

```
<li><a href="/?page=direct" >Direct Message</a></li>
```

Add the following line:

```
<li><a href="/?page=directSent" >Direct Messages Sent</a></li>
```

The Destroy API Method

There is one more direct message API method we did not cover: `direct_messages/destroy`. This is for direct messages that have been sent to you. Yes, it would be great if you could delete a direct message that you sent, but alas, that is not the case. So, never tweet in anger. In addition, your message is not removed off the server, it's

simply marked as deleted. It's up to the client application to respect this tag. This may change in the future.

Here is the format of the destroy call:

```
http://api.twitter.com/version/direct_messages/destroy/:id.format
```

Summary

In this hour, we explored the direct message methods. To ensure that we can send a direct message, we also learned how to test whether two users are friends with each other—a requirement for a direct message to be successfully sent. We also learned that we cannot take back a direct message once it's sent. So, think before you tweet.

Q&A

Q. Can I send a direct message without using the API?

A. Yes. You can send a normal message by placing a `d` and the name of a valid user who is following you. This is not recommended, however.

Q. Can I use Destroy to delete a direct message I sent?

A. No. Destroy will delete messages sent to you.

Workshop

Quiz

1. True or False: For a friendship to exist, both you and your friend must follow each other.
2. True or False: Using the destroy method on a message you have sent will not remove the message.
3. True or False: If you send a direct message to someone who does not follow you, they will get a request to follow you so that your message can go through.

Quiz Answers

1. False. All that is required for someone to receive a direct message is that they follow the sender. The sender does not need to follow the recipient of the direct message.
2. True. You can only destroy messages that have been sent to you. Hopefully this is changed in the future.
3. False. The target of the direct message will not receive any alerts. As such, it's up to the sender to let the recipient know they cannot receive a direct message.

Exercises

1. We did not implement the Destroy method in this hour. As an exercise, let's implement it now.

Because you can only destroy a direct message that was sent to you, you will need to create a second twitter account or get a friend to send you a few direct messages. Remember, you need the message ID to destroy this message. Look back to Hour 11 to remind yourself how to do this.

2. Because we can simply look for an error from Twitter when we try to send a direct message instead of checking for a friendship, try changing your code to skip the friendship checking procedures and instead simply display the error message Twitter provides you.

HOUR 13

Lists

What You'll Learn in This Hour:

- ▶ What is a Twitter list?
- ▶ The List Timeline resources
- ▶ The List Members resources
- ▶ The List Subscribers resources
- ▶ A new approach to restful calls not seen so far
- ▶ How to use the DELETE cURL method and what to do if your host environment does not support it

What Is a List?

Lists in Twitter are interesting—as you would guess, they are lists of Twitter accounts. For something so simple, there are lots of API methods to support it. Most of them around the identification of who is on a list or who is following a list. But first, let's get to know what a list is. On the @perivision account is a list called news-feeds. The members of this list are a few major news outlets. When you view a feed based on a list, it's the same as viewing any other feed.

Based on such a simple concept, you might be surprised at the number of API methods Twitters offers at <http://dev.twitter.com/doc>. Click on lists on the right. Here is the current list:

- ▶ <http://dev.twitter.com/doc/post/:user/lists>
- ▶ <http://dev.twitter.com/doc/post/:user/lists/:id>
- ▶ <http://dev.twitter.com/doc/get/:user/lists>
- ▶ <http://dev.twitter.com/doc/get/:user/lists/:id>

- ▶ <http://dev.twitter.com/doc/delete/:user/lists/:id>
- ▶ <http://dev.twitter.com/doc/get/:user/lists/:id/statuses>
- ▶ <http://dev.twitter.com/doc/get/:user/lists/memberships>
- ▶ <http://dev.twitter.com/doc/get/:user/lists/subscriptions>
- ▶ http://dev.twitter.com/doc/get/:user/:list_id/members
- ▶ http://dev.twitter.com/doc/post/:user/:list_id/members
- ▶ http://dev.twitter.com/doc/post/:user/:list_id/create_all
- ▶ http://dev.twitter.com/doc/delete/:user/:list_id/members
- ▶ http://dev.twitter.com/doc/get/:user/:list_id/members/:id
- ▶ http://dev.twitter.com/doc/get/:user/:list_id/subscribers
- ▶ http://dev.twitter.com/doc/post/:user/:list_id/subscribers
- ▶ http://dev.twitter.com/doc/delete/:user/:list_id/subscribers
- ▶ http://dev.twitter.com/doc/get/:user/:list_id/subscribers/:id

It would not be practical to try to add each method to our sample application, so we are going to add only the ones needed to create and display a list. But we will go through these in more detail at the end of the hour. However, we need to point out a few odd things about these methods and deal with them within our code.

Formal Use of GET, POST, and DELETE

You may notice that many of the methods defined here are duplicates. So, how do we tell them apart and what do they do differently? Let's take `user/lists`, for example.

The API method `user/lists` can do two things. It can create a list, or it can list a user's list—a list of a list if you like. Private lists are not included unless it's your own list you are accessing.

If this seems confusing, that's because it is. The difference between these two API calls is in the way you make the call. If you make a GET call, Twitter interprets this as a request to get a list of the user's lists. If you make a POST call, Twitter interprets this as a request to create a list.

If you look again at the list, you may notice that `user/:id/subscribers` is listed three times! We already know that there is a GET and a POST option. What would be the third? DELETE. DELETE unsubscribes the authenticated user from the specified list. But what if your cURL implementation does not support DELETE?

There is a workaround for clients that cannot issue DELETE requests; instead, use POST with the added parameter `_method=DELETE`.

Fortunately for us, taking the extra step of separating the function call from the actual API method request means we can deal with these new approaches fairly easily. So, let's add a few List methods to our application. We'll take a slightly different approach from the methods we have placed so far. This time, we are going to make a call to Twitter to get a list of our list. Yes, a list of a list. We will display the list and create a hyperlink on each list to make a call to that list and display the list's Twitter stream. That may sound a bit confusing, but it will make perfect sense after we put the code in place and try it out.

If you have not created a list before, it's easy to do so from the Twitter website, as you can see in Figure 13.1.

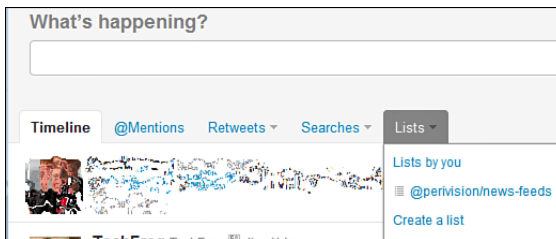


FIGURE 13.1
Creating a list
on the Twitter
website.

Implementing the List API into Our Application

Because we are here to learn about the Twitter API, let's create a list using an API call. We are going to take a slightly different approach. This time, we are going to create a new page when we click the List button. So, let's create that first. Open the `index.php` page and save that code as `list.php`. Basically we are making a copy of `index.php` and calling it `list.php`. Keep the file open; we need to edit a few things.

Edit `list.php`

Remove this line:

```
include 'includes/create_message.php';
```

Then remove this line:

```
<? include 'main.php'; ?>
```

After this line:

```
<div class="container">
```

Add the following lines:

```

    <p />
    <form name='createlists'>
      <input type="text" name="inputbox" value="" />
      <input type="button" onClick="createListItem(this.form);" value="Create
➡ a new list">
    </form>
    <?php getTwitterData(showList); ?>
```

Edit base.js

Now we need to add a few lines in our JavaScript file to support the button we just put in.

Let's add this function after the direct function:

```

function createListItem(form) {

url='commandLine.php?command=createList&name='+form.inputbox.value;
    callPage(url, 'serverMessages');
}

```

Because we are going to make an AJAX call, we need a file to call on the remote server just like we did with `sendMessage.php`. We will call this file `commandLine.php` because we will use this to respond to single requests that does not refresh the page.

Create commandLine.php

Let's create a new file at the root directory and call it 'commandLine.php'.

Add the following lines:

```

<?php

$command=$_GET['command'];
$name = $_GET['name'];
include 'oauth_index.php';

switch ($command) {
    case '';
    case 'createList':
        { $messages=$twitter->createList($name); echo ($messages); }
        break;
    case 'deleteList':

```

```

    { $messages=$twitter->deleteList($name, $twitterName); echo
↳($messages); }
        break;
    }
?>

```

Edit twitteroauth.php

To support our new call, let's add a new function at the end of the file twitteroauth.php but before the last }:

```

function createList($name , $twitterName ){
    $api_call = $twitterName.'/lists/'.$name.'.xml';
    return $this->post($api_call);
}

```

Now that we have created a few lists, let's edit our application to display that list and the feeds for each list item.

Edit header.inc

First, we need to add the correct UI elements. We are going to add only one button to our list tab, and that button will call our Twitter account and display the lists we can create.

After these lines:

```

    <li><a href="/?nav=direct" <? if($nav=='direct') echo
↳'class="here"'; ?> >Direct Message</a>
    <? if($nav=='direct') { ?>
        <ul>
            <li><a href="/?nav=direct&page=direct" >Direct Messages
↳received</a></li>
            <li><a href="/?nav=direct&page=directSent" >Direct Messages
↳Sent</a></li>
        </ul>
    <? } ?>
</li>

```

Add the following lines:

```

    <li><a href="/?nav=list" <? if($nav=='list') echo 'class="here"';
↳?> >List</a>
    <? if($nav=='list') { ?>
        <ul>
            <li><a href="/?nav=list&page=showList" >Show a list of
↳lists</a></li>
        </ul>
    <? } ?>

```

Edit parseTwitter.php

Now that we have our UI in place, let's account for our new commands in our case switch. Open the file parseTwitter.php.

After the following lines:

```

        case 'rt_of_me':
            { $messages=$twitter->getMentions('xml'); return
➔call_timeline($messages); }
            break;

```

Add these lines:

```

        case 'showList':
            { $messages=$twitter->showLists(); return call_showList($messages); }
            break;
        case 'renderList':
            { $messages=$twitter->renderLists($name, $twitterName); return
➔call_timeline($messages); }
            break;

```

After this function:

```
function call_direct($messages){ ...
```

Add this function:

```
function call_showList($messages){
```

Notice that we have done something different here. Normally, we use the SimpleXMLElement() PHP call to parse our return from Twitter, but in this case we cannot. Here is the return the 'list.xml' <get> method returns us:

```

<lists_list>
<lists_type="array">
<list>
  <id>20003603</id>
  <name>testlistno6</name>
  <full_name>@perivision/testlistno6</full_name>
  <slug>testlistno6</slug>
  <description></description>
  <subscriber_count>0</subscriber_count>
  <member_count>1</member_count>
  <uri>/perivision/testlistno6</uri>
  <following>>false</following>
  <mode>public</mode>
  <user>
    <id>11710512</id>
    <name>Christopher</name>
    <screen_name>perivision</screen_name>
    <location>the bay</location>

```

The word 'list' is a reserved word with SimpleXMLElement(), so we are going to go with another XML parsing technique:

```
...
    $parser = xml_parser_create();
    xml_parser_set_option($parser,XML_OPTION_SKIP_WHITE,1);
    xml_parser_set_option($parser,XML_OPTION_CASE_FOLDING,0);
    xml_parse_into_struct($parser,$messages,$d_ar,$i_ar) or
print_error();
    xml_parser_free($parser);
```

Here we make another departure from how we have been doing things up to this point. Because all we are going to do is display a simple list of our Twitter list, we do not need to return back to main and render:

```
$i=0;
while($i_ar['name'][$i]){
    $user_name = $i_ar['slug'][$i];
    $user_name = $d_ar[$user_name][value];
    echo '<li><a href="/hour13/?nav=list&page=renderList&name='.$
$user_name.'" >'. $user_name.'</a></li>';
    $i++;
}
}
```

Edit twitteroauth.php

Now let's add a new function to twitteroauth.php to support our request.

After this function:

```
function friendshipExists($a, $b){
    $api_call = " friendships/exists.xml?user_a=$a&user_b=$b";
    return $this->get($api_call);
}
```

Add these functions:

```
function showLists(){
    $api_call = 'lists.xml';
    return $this->get($api_call);
}
function renderLists($name, $twitterName){

    $api_call = $twitterName.'/lists/'.$name.'/statuses.xml';
    return $this->get($api_call);
}
```


Three Types of List Methods

Twitter groups the List methods into three categories:

- ▶ List resources
- ▶ List Members resources
- ▶ List Subscribers resources

We have already gone through some of the methods as we added new functions to our application; however, we need to cover many more.

List Resources

List resources are methods that focus on managing the list; that is, creating, editing, and deleting the list.

Following is a summary of the methods from the Twitter docs:

- ▶ **GET user/:lists**—Lists the lists of the specified user. Private lists will be included if the authenticated user is the same as the user whose lists are being returned.
- ▶ **POST user/:lists**—Creates a new list for the authenticated user. Accounts are limited to 20 lists.
- ▶ **GET user/lists/:id**—Shows the specified list. Private lists will be shown only if the authenticated user owns the specified list.
- ▶ **POST user/lists/:id**—Updates the specified list.
- ▶ **DELETE user/lists/:id**—Deletes the specified list. Must be owned by the authenticated user.
- ▶ **GET user/lists/:id/statuses**—Shows tweet timeline for members of the specified list.
- ▶ **GET :user/lists/memberships**—Lists the lists the specified user has been added to.
- ▶ **GET :user/lists/subscriptions**—Lists the lists the specified user follows.

List Members Resources

Members resources are the List methods that provide access to who is following a list as well as manipulating the members of a list. Although the use case is limited, the most common use would be for automatically updating a list based on other criteria. Perhaps you want to automatically create a list of the people you interact with most often. In this case, you could add a name to this list anytime you do a reply or send a direct message, and you can remove people from the list if they do not show up in any of your sent messages for a set of time or the last 100 messages.

Here is a brief summary of the list from the Twitter docs:

- ▶ **GET** `:user/:list_id/members`—Returns the members of the specified list.
- ▶ **POST** `:user/:list_id/members`—Adds a member to a list. The authenticated user must own the list to be able to add members to it. Lists are limited to having 500 members.
- ▶ **DELETE** `:user/:list_id/members`—Removes the specified member from the list. The authenticated user must be the list's owner to remove members from the list.
- ▶ **POST** `:user/:list_id/create_all`—Adds multiple members to a list by specifying a comma-separated list of member IDs or screen names. The authenticated user must own the list to be able to add members to it. Lists are limited to having 500 members, and you are limited to adding up to 100 members to a list at a time with this method.

List Subscribers Resources

The Subscribers method deals with the subscribers of a list. This is different from the List Members methods in that we are dealing with those who are following 'subscribed' to a list versus managing lists that a member is subscribed to. Again, this is a confusing but important distinction. Read through these carefully. You may need to read over this section more than once:

- ▶ **Member**—Adds a userID to a list.
- ▶ **Subscriber**—Adds a list to a userID's list of lists.

For example, the GET `:user/:list_id/members` method returns the members of a list, whereas the GET `:user/:list_id/members/:id` method checks to see if the user is a member of a list. Some methods seem to perform the same actions; however, they are different. POST `:user/:list_id/members` adds a member to a list. POST `:user/:list_id/subscribers` : makes the authenticated user follow the specified list. For

example, if we have a list called football, we could pass the following to the Twitter servers:

```
http://api.twitter.com/1/twitterapi24/football/members.xml?id=12345
```

This would add the Twitter user 12345 to the list football. However, if we pass this request to Twitter:

```
http://api.twitter.com/1/twitterapi24/football/subscribers.xml?id=12345
```

It would add the list 'football' to the user '12345' list of lists.

Here is a brief summary of the list:

- ▶ **GET :user/:list_id/ subscribers /:id:** Checks if a user is a member of the specified list.
- ▶ **POST :user/:list_id/subscribers:** Makes the authenticated user follow the specified list.
- ▶ **DELETE :user/:list_id/subscribers:** Unsubscribes the authenticated user from the specified list.
- ▶ **GET :user/:list_id/subscribers/:id:** Checks if a user is a subscriber of the specified list.

Summary

In this hour, we learned about the three types of List methods: timeline, members, and subscribers. We also learned the numerous ways we can manage a list. This hour also saw us again using more advanced techniques of the REST protocol.

We also created a new file called `commandLine.php`, which we do not call directly, but instead make a call from a JavaScript action. We will use this file a few more times in the book.

Q&A

- Q. Why did the good folks at Twitter start a new API method where API calls are the same name?**
- A.** The API is always evolving, and the list API calls are some of the most recent. Thus, it could be an indication of how future API calls will be designed.

Workshop

Quiz

1. Name the three list types.
2. List the three types of cURL request.
3. If your service does not support DELETE, what can you do?

Quiz Answers

1. List resources, List Members resources, and List Subscribers resources.
2. GET, POST, and DELETE.
3. A workaround for clients that cannot issue DELETE requests can use POST with the added parameter `_method=DELETE` instead.

Exercise

As suggested in an earlier part of this hour, add users to a list if you send them a message or retweet one of their messages. Remember to check to see if they are a member of the list first before adding.

Did you notice the function we called in the following lines?

```
. $parser = xml_parser_create();  
xml_parser_set_option($parser,XML_OPTION_SKIP_WHITE,1);  
xml_parser_set_option($parser,XML_OPTION_CASE_FOLDING,0);  
xml_parse_into_struct($parser,$messages,$d_ar,  
$i_ar) or print_error();  
xml_parser_free($parser);
```

We did not define `'print_error()'`. Let's do that now.

This page intentionally left blank

HOUR 14

Favorites and User Methods

What You'll Learn in This Hour:

- ▶ What is a favorite?
- ▶ How to create, read, and destroy favorites
- ▶ Learn about user functions
- ▶ How to create an image-based address book of Twitter friends and followers

Favorites API Methods

One of the most underrated features of Twitter but yet has the most promise is favorites. Favorites are only just now beginning to surface as a powerful feature for both the user and those performing analytics on Twitter data. A favorite is a tweet that a user saves. It's somewhat like creating a bookmark of a web page, except the user saves a message ID instead of a URL. The only thing missing is the ability to see other peoples' favorites, but who knows? Perhaps that will change in the future.

The API method calls for favorites are pretty simple; there are only three:

- ▶ GET <http://api.twitter.com/1/favorites.format>
- ▶ POST <http://api.twitter.com/1/favorites/create/id.format>
- ▶ POST/DELETE <http://api.twitter.com/1/favorites/destroy/id.format>

These methods work much like the rest of the methods we have been working with, so let's get to coding and add these methods to our application.

Adding Favorites Methods to Our Application

Now that we understand what a favorite is, let's update our application to support these API calls. First, place the code to read your list of favorites.

Edit header.inc

Keeping with how we have been adding other methods, we are going to add a few lines to our header so that we can make the call to favorites. After the HTML tags ` lists `, add the following lines:

```

        <li><a href="/?nav=showFavorites&page=showFavorites" <?
if($nav=='showFavorites') echo 'class="here"'; ?> >Favorites</a>
        <? if($nav=='showFavorites') { ?>
            <ul>
                <li><a href="/?nav=favorites&page=showFavorites" >Show
➤Favorites</a></li>
                </ul>
            <? } ?>
        </li>

```

Edit parseTwitter.php

Now that we have our text button, let's edit `parseTwitter.php` to catch the user action. Because the return from Twitter will be a standard feed of tweets, we can use the `'call_timeline()'` function to display our return.

After the case statement:

```

        case 'renderList':
            { $messages=$twitter->renderLists($name, $twitterName); return
call_timeline($messages); }
            break;

```

Add the following lines:

```

case 'showFavorites':
{ $messages=$twitter->showFavorites(); return call_timeline($messages); }
break;

```

Edit twitteroauth.php

Next, we need to support the `showFavorites()` we just created. Open `twitteroauth.php` and add the following function at the end of the file but before the last `'}'`:

```

function showFavorites(){
    $api_call = 'favorites.xml';
        return $this->get($api_call);
}

```

Edit base.js

Next, let's add a function in our JavaScript file to accept the clicks we are setting up. Open `base.js` and add these two functions after the `createListItem()` function:

```
function favorite(mess_id){
    url='commandLine.php?command=createFavorite&id='+mess_id;
    callPage(url, mess_id);
}
```

Save and close your files and give it a test. This will work only if you have selected a few messages as favorites. If you have not, you will get nothing back. If you do not know how to create a favorite, go to Twitter and click the star of any tweet. The star will show up when you mouse over the tweet, as shown in Figure 14.1. Go ahead and favor a few tweets; then go back to our application and make sure they show up when you click on the tab titled "favorites."

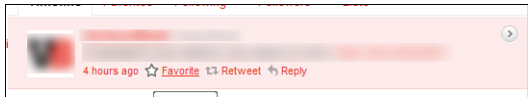


FIGURE 14.1
Create a favorite on Twitter.

Create a Favorite Tweet Using the Twitter API Method

Now that you can get your favorites, we need to be able to create favorites via Twitter API method calls. To create a favorite, we need the message's ID. The easiest way to do that is similar to how we supported creating a retweet message and populating our message box to send a direct message. We are going to place a link within the message itself.

Edit render.php

Open the `render.php` file and remove the trailing `</div>` by editing the following line from:

```
<a href='#' onmouseup="\direct('$screen_name', '$twitterName')\">DM</a></div>
```

To:

```
<a href='#' onmouseup="\direct('$screen_name', '$twitterName')\">DM</a>
```

Now add the following code:

```
<a href='#' onmouseup="\favorite('$id')\">F</a></div>
```


Save your file and refresh or open `index.php` and then select Favorites. You should see your list of favorite Twitter messages with an “F” below the “RT” and the “DM.” Clicking this F will not do anything yet because we need to capture this click in JavaScript, so let’s add some code to both `commandLine.php` and `twitteroauth.php` to execute the request.

Edit `commandLine.php`

After the case switch:

```
case 'deleteList':
    { $messages=$twitter->deleteList($name); echo ($messages); }
    break;
```

Add the following case switch:

```
case 'createFavorite':
    { $messages=$twitter->createFavorite($id); echo ($messages); }
    break;
```

Save and close this file.

Edit `twitteroauth.php`

Now let’s add the method call to our `twitteroauth` class. Open `twitteroauth.php` and add the following code after the last function but before the last ‘}’:

```
function createFavorite($id){
    $api_call = ' favorites/create/' . $id . '.xml';
    return $this->post($api_call);
}
```

Save and close this file.

We should now be able to create a favorite tweet from our application. Open your application and click the F from any tweet. You should get back an unstructured XML reply from Twitter, similar to Figure 14.2.

Pretty simple, right? To delete a favorite, we are going to destroy it. I’m guessing the good folks at Twitter decided that it was not good enough to delete a favorite—you have to go all out and destroy it. For this method, we can use either POST or DELETE; however, there is one tricky bit around the UI. To destroy a favorite, we need to have that message’s ID, and the best way to capture that is when we are rendering the tweet stream. So, we are going to be a bit creative here and put an if statement in the `render.php` call so that we can offer to delete (destroy) a tweet.



FIGURE 14.2
XML return from creating a favorite on Twitter.

Edit render.php

We need to add one extra line of code to support this if statement. To know whether we are showing the favorite list, we are going to look at the URL and see if `nav=showFavorites`. If it does, instead of rendering the 'F' link, we will display a 'Del' (delete) link.

Open `render.php` and at the top of the file, find the following line:

```
global $twitterName;
```

Add the following code:

```
$nav = $_GET['nav'];
```

Now that we know what navigation page we are on, we can create our if statement.

Find the following line of code:

```
<a href='#' onmouseup="favorite('$id')">F</a></div>
```

And add a trailing quote and semi colon as shown:

```
<a href='#' onmouseup="favorite('$id')">F</a></div> ";
```

And add the following:

```

    if($nav=='showFavorites') { $output.="<a href='#'
➔onmouseup=\"destroyFavorite('$id')\">Del</a></div> ";
    } else { $output.="<a href='#'
➔onmouseup=\"favorite('$id')\">F</a></div> "; }
$output.="

```

Save and close the file.

Edit base.js

Now that we have created a new link, let's catch it in JavaScript. Open base.js and add the following function after the 'favorite' function:

```

function destroyFavorite(mess_id){
    url='commandLine.php?command=destroyFavorite&id='+mess_id;
    callPage(url, mess_id);
}

```

Save and close.

Edit twitteroauth.php

Just as we did before, we will add a function call in twitteroauth to catch and process the destroy request.

Open twitteroauth.php and add the following function at the end of the file but before the '}':

```

function destroyFavorite($id){
    $api_call = 'favorites/destroy/'.$id.'.xml';
    return $this->post($api_call);
}

```

If you remember our reference to the destroy method in the beginning of this hour, we could use either POST or DELETE for this method call. We are going to use DELETE because it's more informative when we read the code. Either way will work; however, using DELETE could prove to be more future proofed as Twitter continues to evolve the API set.

Save and close the file.

Now let's give it a shot. Open index.php and click Favorites. Now click 'Del' on one of the messages. You should get an XML return similar to Figure 14.3 that indicates you have destroyed that favorite message.



The screenshot shows a web browser window with a 'Send' button at the top left. Below it, the URL is `https://api.twitter.com/1/account/verify_credentials.xml`. The main content is an XML response from the Twitter API. The XML starts with `RT url http://api.twitter.com/1/favorites/destroy/47514069883817984.xml`. It then contains a tweet from user '24k' (Chris Rauschnot) with a profile picture and a bio. The XML structure is as follows:

```

url https://api.twitter.com/1/account/verify_credentials.xml
url http://api.twitter.com/1/favorites.xml
RT url http://api.twitter.com/1/favorites/destroy/47514069883817984.xml
DM Tue Mar 15 04:26:51 +0000 2011 47514069883817984 Create a chatroom in 12 lines of code
Delwith NowJS http://trw.to/17ZlQ on @TheNextWeb #socialnetworking <a
href="http://www.echofon.com/" rel="nofollow">Echofon</a> false false 0 false 17145442
Chris Rauschnot 24k Las Vegas, NV Web 2.0 & Social Media Guy, Speaker, Internet
Entrepreneur for over 15 years. Certified Apple Tech. @24kMedia http://bit.ly/24kMediaFB
#vegas #tech #celebrity http://a3.twimg.com/profile_images/1193040615/24k_normal.jpg
http://TheMacWizard.com/false 50595 9AE4E8 333333 0000# DDFCC BDDCAD 42760 Tue
Nov 04 01:09:10 +0000 2008 3200 -28800 Pacific Time (US & Canada) http://a0.twimg.com
/profile_background_images/183661076/CRK002_TwitterBGround2.jpg false true false true
false true 38447 en false false 1206 false false SimpleXMLElement Object ( [created_at] =>
Tue Mar 15 04:26:51 +0000 2011 [id] => 47514069883817984 [text] => Create a chatroom in
12 lines of code with NowJS http://trw.to/17ZlQ on @TheNextWeb #socialnetworking [source]
=> Echofon [truncated] => false [favorited] => false [in_reply_to_status_id] =>
SimpleXMLElement Object ( [in_reply_to_user_id] => SimpleXMLElement Object (
[in_reply_to_screen_name] => SimpleXMLElement Object ( ) [retweet_count] => 0 [retweeted]
=> false [user] => SimpleXMLElement Object ( [id] => 17145442 [name] => Chris Rauschnot
[screen_name] => 24k [location] => Las Vegas, NV [description] => Web 2.0 & Social Media
Guy, Speaker, Internet Entrepreneur for over 15 years. Certified Apple Tech. @24kMedia
http://bit.ly/24kMediaFB #vegas #tech #celebrity [profile_image_url] => http://a3.twimg.com
/profile_images/1193040615/24k_normal.jpg [url] => http://TheMacWizard.com/ [protected] =>
false [followers_count] => 50595 [profile_background_color] => 9AE4E8 [profile_text_color] =>
333333 [profile_link_color] => 0000ff [profile_sidebar_fill_color] => DDFCC
[profile_sidebar_border_color] => BDDCAD [friends_count] => 42760 [created_at] => Tue Nov
04 01:09:10 +0000 2008 [favourites_count] => 3200 [utc_offset] => -28800 [time_zone] =>
Pacific Time (US & Canada) [profile_background_image_url] => http://a0.twimg.com
/profile_background_images/183661076/CRK002_TwitterBGround2.jpg
[profile_background_tile] => false [profile_use_background_image] => true [notifications] =>
false [geo_enabled] => true [verified] => false [following] => true [statuses_count] => 38447
[lang] => en [contributors_enabled] => false [follow_request_sent] => false [listed_count] =>
1206 [show_all_inline_media] => false [is_translator] => false ) [geo] => SimpleXMLElement
Object ( [coordinates] => SimpleXMLElement Object ( ) [place] => SimpleXMLElement
Object ( ) [contributors] => SimpleXMLElement Object ( ) [annotations] => SimpleXMLElement
Object ( ) )

```

FIGURE 14.3
XML return
from destroying
a favorite on
Twitter.

User API Methods

The user methods in Twitter are straightforward in that they are methods for getting information about a Twitter user. The user methods are as follows, all using the GET method:

- ▶ users/show
- ▶ users/lookup
- ▶ users/search
- ▶ users/suggestions
- ▶ users/suggestions/category
- ▶ statuses/friends
- ▶ statuses/followers

Add users/show to Our Application

The users/show method is much like the other API methods we have seen so far. Placing support for it in our application will take a familiar path. We will create a tab on the UI and then support the link with new code within `parseTwitter.php` and `twitteroauth.php`.

Edit header.inc

First, we'll add a new `` tab. After the following code:

```

    <li><a href="/?nav=showFavorites&page=showFavorites" <?
if($nav=='showFavorites') echo 'class="here"'; ?> >Favorites</a>
    <? if($nav=='showFavorites') { ?>
        <ul>
            <li><a href="/?nav=favorites&page=showFavorites" >Show
➔Favorites</a></li>
            </ul>
        <? } ?>
    </li>

```

Add the following code:

```

    <li><a href="/?nav=user&page=showUser" <? if($nav=='user') echo
➔'class="here"'; ?> >User</a>
    <? if($nav=='user') { ?>
        <ul>
            <li><a href="/?nav=user&page=showUser" >Show Users</a></li>
        </ul>
    <? } ?>
    </li>

```

Save and close the file.

Edit parseTwitter.php

Open `parseTwitter.php` and add a new case switch using the following code:

```

    case 'showUser':
        { $messages=$twitter->showUser($twitterName); echo($messages); }
        break;

```

Save and close the file.

Edit twitteroauth.php

Finally, we will add support for the method call. Open `twitteroauth.php` and add the following function at the end of the file but before the last `'`:

```

function showUser($name){
    $api_call = 'users/show.xml?screen_name='.$name;
    return $this->get($api_call);
}

```

Save and close the file.

Now open `index.php` and click the User tab. You should get an unstructured XML stream from Twitter with details on your Twitter account, similar to Figure 14.4.

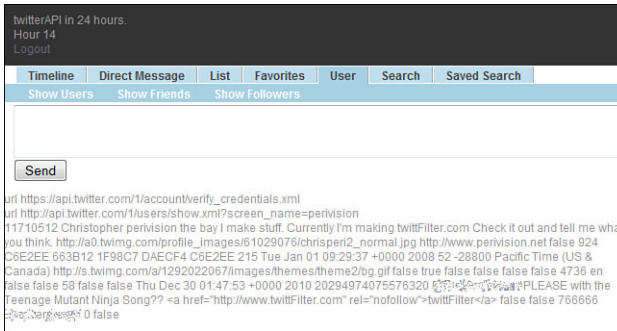


FIGURE 14.4
XML details on
your Twitter
account.

Accessing Other User Information

What if you wanted to get information on someone else? Easy enough—simply replace your Twitter name with another Twitter name or ID. You can also get more than one detailed report on a Twitter account at a time. With the API method call `GET users/lookup`, you can pass more than one `screen_name` or ID at a time. It's the same structure that we used for `'users/show'`, except we use `'users/lookup'` and provide a list of names or IDs separated by commas. For example, we just used `'http://api.twitter.com/1/users/show.xml?screen_name=perivision'`, where `'perivision'` is my account name. To construct a `'lookup'` API call, it would look something like the following:

```
http://api.twitter.com/1/users/lookup.xml?screen_name=perivision,cbsnews,
abc,breakingnews, ...
```

You would then get a list of `<users>` in your XML return, each with details about the user. Notice that I did not include a space after the comma. This is an HTTP request, and there should never be a space—but you know that, didn't you?

Understanding More of the Users APIs

`GET users/search` is an interesting API method in that it will return the first 100 users that best match a search. This functions much like the Find People button on Twitter.com.

The next three methods are interesting, but useful only for targeted applications:

- ▶ **GET users/suggestions**—Access to Twitter's suggested user list. This returns the list of suggested user categories. The category can be used in the `users/suggestions/category` endpoint to get the users in that category. And where would we use this category? In the following method call:
- ▶ **GET users/suggestions/:slug**—Access the users in a given category of the Twitter suggested user list.

For example, we can get a list of Twitter recommended users based on the category 'books'. The API call would look like the following:

```
http://api.twitter.com/1/users/suggestions/books.xml
```

This API call would then return a list of users whom Twitter thinks is a best fit for the category 'books'.

- ▶ **GET users/profile_image/:screen_name**—Access the profile image in various sizes for the user with the indicated screen_name. If no size is provided, the normal image is returned. Note this warning from the Twitter docs:
This resource does not return JSON or XML, but instead returns a 302 redirect to the actual image resource.

This method should be used only by application developers to look up or check the profile image URL for a user. This method must not be used as the image source URL presented to users of your application.

Basically, what the Twitter folks are saying here is DO NOT USE THIS! Really, there is very little reason to use this call.

Create a Simple Thumbnail Viewer in Our Application

We have two more calls to look at:

- ▶ **GET statuses/friends**—Returns a user's friends, each with current status inline. They are ordered by the order in which the user followed them; the most recently followed are first, 100 at a time.
- ▶ **GET statuses/followers**—Returns the authenticating user's followers, each with current status inline. They are ordered by the order in which they followed the user, 100 at a time.

These last two calls can be quite useful for creating an image thumbnail viewer which could be the foundation of an address book of sorts for our application. So, let's add these to our application. We'll create a page of Twitter images and screen names so that we can skim through the list to find the person we are interested in.

Edit header.inc

First, we need a button, so open the file header.inc, and after this line:

```
<li><a href="/?nav=user&page=showUser" >Show Users</a></li>
```

Add the following line:

```
<li><a href="/?nav=user&page=showFriends" >Show Friends</a></li>
<li><a href="/?nav=user&page=showFollowers" >Show Followers</a></li>
```

Save and close this file.

Edit parseTwitter.php

We will now add two new case switches for our new requests. Open parseTwitter.php, and after this case switch:

```
    case 'showUser':
        { $messages=$twitter->showUser($twitterName); echo $messages;
↳return; }
        break;
```

Add the following code:

```
    case 'showFriends':
        { $messages=$twitter->showFriends($twitterName); return
↳call_users($messages); }
        break;
    case 'showFollowers':
        { $messages=$twitter->showFollowers($twitterName); return
↳call_users($messages); }
        break;
```

Don't close just yet. We need to create a function to render our list of images and screen names. Within this same file, add the following function after the function 'call_showList(\$messages)':

```
function call_users($messages){
    $twitterReturn = new SimpleXMLElement($messages);
    foreach($twitterReturn->user as $status){
        $profile_image_url = $status->profile_image_url;
        $screen_name = $status->screen_name;
        echo "
            <div id='$screen_name' class='mess-pic' \" >
                <a href='http://twitter.com/$screen_name'>
                    <img src='$profile_image_url'/ width='48px'
↳height='48px'>
                <br>$screen_name</a>
            </div>";
    }
}
```

Because each return is a 'user', we stepped through the XML return \$twitterReturn setting \$status to each 'user' node instead of 'status' node, as we have done in the past.

Save and close this file.

Edit twitteroauth.php

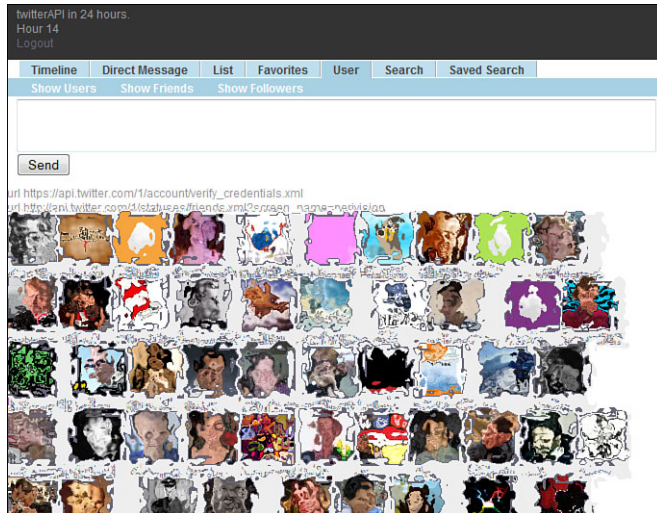
Finally, we need to implement the actual API method calls. Open twitteroauth.php and add the following functions at the end of the file but before the last '}':

```
function showFriends($name){
    $api_call = ' statuses/friends.xml?screen_name=' . $name;
    return $this->get($api_call);
}
function showFollowers($name){
    $api_call = ' statuses/followers.xml?screen_name=' . $name;
    return $this->get($api_call);
}
```

Save and close this file.

Now open index.php and click users, friends, or followers. Assuming you have at least one or more friends or followers, you should see something like the screenshot in Figure 14.5.

FIGURE 14.5
List of friends using images and screen names.



Summary

In this hour, we learned about favorites and how to create, edit, and destroy through the API. We also used the JavaScript to commandLine.php technique again because of the need to track message IDs. You should be fairly comfortable with this technique by now. If not, you should reread the past two hours because future Twitter API methods will rely more and more on IDs to perform actions as the API evolves.

We also explored users' methods in this hour. Unlike the methods we have been working with in the past two hours, the users' methods are based on GET only.

Q&A

Q. *Can I send an update using 'F' like I can with 'D'?*

A. Yes. Any currently supported shortcut can be used via the update method.

Q. *Because there is a way to use 'POST' to delete a favorite, why should I bother with DESTROY?*

A. Although either cURL call will work, we do not know what the future of the API will be, but indications are that more formal uses of the cURL protocol will be used, and that includes DESTROY.

Workshop

Quiz

1. Does 'GET users/profile_image/:screen_name' return JSON or XML?
2. How many returns do I get with the call 'GET statuses/friends'?
3. What does GET statuses/search do?

Quiz Answers

1. Neither. It returns the URL of an image only.
2. 100 at a time.
3. Returns the first 100 Twitter users that best match the search term provided.

Exercise

Now that know how to show friends and we can show followers, write a script that will check to see what user we both friend and follower. Create a new tab for this.

This page intentionally left blank

HOOR 15

Search

What You'll Learn in This Hour:

- ▶ The history of the Search API and why it's different from the other Twitter API methods
- ▶ About the single Search API call
- ▶ How to construct a search query using attributes
- ▶ How to convert characters not supported though REST requests
- ▶ How to make and parse a JSON request

History of Twitter Search API

Users who remember using Twitter from the very beginning know that Search was not a feature of Twitter. Many things were not a feature of Twitter in fact, and thus many other companies started writing software to fill this void. Summize was one of these companies. Like Twitter, Summize offered an API so other programmers could access the Summize Search service. Search is, as you would imagine, a means to search through the Twitter history of tweets. However, there are billions of tweets, and searching through all of them is simply not practical. But in the beginning, when the number was only in the six-figures range, not only was it practical, it was downright useful—so useful that Twitter made Summize its first major acquisition. Here is a bit more on Twitter Search from the site documents:

Twitter's Stance on Search

“The Twitter API consists of three parts: two REST APIs and a Streaming API. The two distinct REST APIs are entirely due to history. Summize, Inc. was originally an independent company that provided search capability for Twitter data. Summize was

later acquired and rebranded as Twitter Search. Rebranding the site was easy; fully integrating Twitter Search and its API into the Twitter codebase is more difficult. It is in our pipeline to unify the APIs, but until resources allow, the REST API and Search API will remain as separate entities. The Streaming API is distinct from the two REST APIs as Streaming supports long-lived connections on a different architecture.

The Twitter REST API methods allow developers to access core Twitter data. This includes update timelines, status data, and user information. The Search API methods give developers methods to interact with Twitter Search and trends data. The concern for developers given this separation is the effects on rate limiting and output format.”

The Lone Search API

The Search API has only one API method: search. That’s it. However, the power of Search is within the options you pass to Twitter. First, let’s have a look at the API definition and then the option list:

GET search—Returns tweets that match a specified query.

Note that as of April 1, 2010, the Search API provides an option to retrieve “popular tweets” in addition to real-time search results. In an upcoming release, this will become the default, and clients that don’t want to receive popular tweets in their search results will have to explicitly opt out. See the `result_type` parameter for more information.

Be aware that the user IDs in the Search API are different from those we have seen so far. This means that the `to_user_id` and `from_user_id` field vary from the actual user ID on Twitter.com. Applications will have to perform a screen name-based lookup with the `users/show` method to get the correct user ID if necessary.

The most typical search attribute you will use is `?q=`. `q` stands for query. However, there are other options. Let’s have a look at the current list from the Twitter docs (<http://dev.twitter.com/doc/get/search>):

- ▶ **callback**—Available only for JSON format. If supplied, the response will use the JSON format with a callback of the given name.
- ▶ **lang**—Restricts tweets to the given language, given by an ISO 639-1 code.
- ▶ **locale**—Specifies the language of the query you are sending (only `ja` is currently effective). This is intended for language-specific clients, and the default should work in the majority of cases:

<http://search.twitter.com/search.json?locale=ja>

- ▶ **rpp**—The number of tweets to return per page, up to a max of 100:

<http://search.twitter.com/search.json?rpp=100>

- ▶ **page**—The page number (starting at 1) to return, up to a max of roughly 1,500 results (based on $rpp * page$):

<http://search.twitter.com/search.json?page=10>

- ▶ **since_id**—Returns results with an ID greater than (that is, more recent than) the specified ID. There are limits to the number of tweets that can be accessed through the API. If the limit of tweets has occurred since the `since_id`, the `since_id` will be forced to the oldest ID available. If the `since_id` used is too old, a HTTP 404 error will be returned:

http://search.twitter.com/search.json?since_id=12345

- ▶ **until**—Optional. Returns tweets generated before the given date. Date should be formatted as YYYY-MM-DD:

<http://search.twitter.com/search.json?until=2010-03-28>

- ▶ **geocode**—Returns tweets by users located within a given radius of the given latitude/longitude. The location is preferentially taking from the Geotagging API, but will fall back to the Twitter profile. The parameter value is specified by `latitude,longitude,radius`, where radius units must be specified as either `mi` (miles) or `km` (kilometers). Note that you cannot use the `near` operator via the API to geocode arbitrary locations; however, you can use this `geocode` parameter to search near geocodes directly:

<http://search.twitter.com/search.json?geocode=37.781157,-122.398720,1mi>

- ▶ **show_user**—When true, prepends “:” to the beginning of the tweet. This is useful for readers that do not display Atom’s author field. The default is false.

- ▶ **result_type**—Optional. Specifies what type of search results you would prefer to receive. The current default is “mixed.” Valid values include the following:

- ▶ **mixed**—Includes both popular and real-time results in the response.
- ▶ **recent**—Returns only the most recent results in the response.
- ▶ **popular**—Returns only the most popular results in the response.

http://search.twitter.com/search.json?result_type=mixed
http://search.twitter.com/search.json?result_type=recent
http://search.twitter.com/search.json?result_type=popular

Quite a bit, right? We will go through this step by step and cover the major options you would most commonly use.

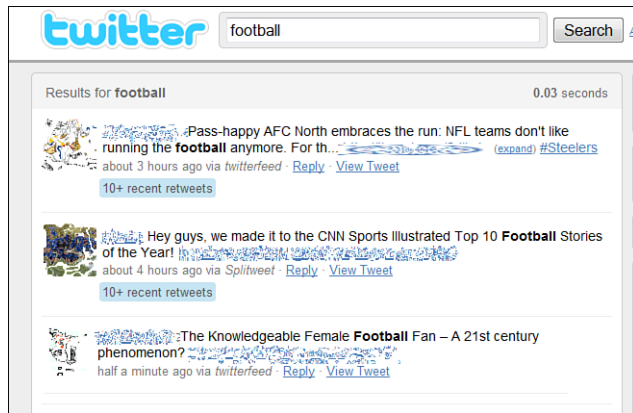
Search Request Parameters

First, let's have a look at a simple search request. Open up your web browser and type this into the URL address text field:

```
http://search.twitter.com/search?q=football
```

You should have a reply from the Twitter website that looks similar to Figure 15.1.

FIGURE 15.1
Search on
Twitter.com.



Here we set the value of *q* to *football* and got a return of the latest searches with the keyword *football* in the latest tweet. This is not the only option for Search that is available to us. Consider the following search request:

```
http://search.twitter.com/search?q=football+bears
```

By using the '+' sign, we now get tweets that have both words. However, if we want to have two words in a specific order, we would use quotes, as in the following example:

```
http://search.twitter.com/search?q="cal+bears"
```

**Watch
Out!**

Remember, you can only use web-safe symbols when making http calls.

You can see how this works. We control our Twitter search completely through the options we pass to Twitter. Let's have a look at the various options as described in the Twitter docs (<http://search.twitter.com/api/>):

- ▶ Find tweets **containing a word**:
`http://search.twitter.com/search.atom?q=twitter`
- ▶ Find tweets **from a user**:
`http://search.twitter.com/search.atom?q=from%3Aalexiskold`
- ▶ Find tweets **to a user**:
`http://search.twitter.com/search.atom?q=to%3Atechcrunch`
- ▶ Find tweets **referencing a user**:
`http://search.twitter.com/search.atom?q=%40mashable`
- ▶ Find tweets **containing a hashtag**:
`http://search.twitter.com/search.atom?q=%23haiku`
- ▶ Combine any of the operators together:
`http://search.twitter.com/search.atom?q=movie+%3A%29`

The API also supports the following optional URL parameters:

- ▶ **lang**—Restricts tweets to the given language, given by an ISO 639-1 code. For example: `http://search.twitter.com/search.atom?lang=en&q=devo`
- ▶ **rpp**—The number of tweets to return per page, up to a max of 100. For example: `http://search.twitter.com/search.atom?lang=en&q=devo&rpp=15`
- ▶ **page**—The page number to return, up to a max of roughly 1,500 results (based on `rpp * page`).
- ▶ **since_id**—Returns tweets with status IDs greater than the given ID.
- ▶ **geocode**—Returns tweets by users located within a given radius of the given latitude/longitude, where the user's location is taken from the Twitter profile. The parameter value is specified by "latitude,longitude,radius", where radius units must be specified as either "mi" (miles) or "km" (kilometers). For example: `http://search.twitter.com/search.atom?geocode=40.757929%2C-73.985506%2C25km`. Note that you cannot use the near operator via the API to geocode arbitrary locations; however, you can use this geocode parameter to search near geocodes directly.
- ▶ **show_user**—When "true", adds "<user>:" to the beginning of the tweet. This is useful for readers that do not display Atom's author field. The default is "false".

Integrating Search into Our Application

Now that we have a good idea about how Search works, we can start to get Search integrated into our application. We are going to take a slightly different approach from what we have been doing in this book so far, but we are still going to work within our normal framework.

Edit header.inc

Let's add a new button. After the last tag we created in Hour 14, `<a href="/?nav=user&page=showUser..."`, we'll add the following code:

```

    <li><a href="/?nav=search&page=search" <? if($nav=='search') echo
'class="here"' ; ?> >Search</a>
    <? if($nav=='search') { ?>
        <ul>
            <li><a href="/?nav=search&page=search" >Search</a></li>
        </ul>
    <? } ?>
</li>

```

Save and close.

Edit parseTwitter.php

Now let's add a switch to our case statement to catch the new link we just created. Open `parseTwitter.php` and add the following case switch at the end of the switch statement:

```

    case 'search':
        { $messages=$twitter->search($options); return
↳call_search($messages); }
        break;

```

Don't save and close just yet. Remember that we are not dealing with the normal method calls we have seen throughout the book. Search does not have an option to return XML; instead, we can get only JSON, ATOM, or RSS. In our case, we are going to request a JSON feed.

Did You Know?

JSON is one of the more popular formats for sending and receiving data over the web. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. You can learn more about JSON by going to this website:

www.json.org/

To handle parsing JSON instead of XML as we have so far in the book, we need to make a few changes to our parse function. We are going to start with our `call_timeline()` function and then make some changes.

We'll create a new function below the function `call_users()` in this file. Add the following code:

```
function call_search($messages){
    $twitterReturn = json_decode($messages);
```

Here we have introduced a new PHP library call called `json_decode()`. This function will convert the json return into a PHP variable:

```
    $i=0;
    foreach($twitterReturn->results as $status){
```

Now that we have the return converted to a PHP object, we can step through each of the results and get our values, just as we did with our other parse functions:

```
        $updateTime[$i] = parseDate($status->created_at);
        $update[$i] = $status->text;
        $id[$i] = $status->id;
        $profile_image_url[$i] = $status->profile_image_url;
        $screen_name[$i] = $status->from_user;

        $i++;
    }

    $parsedReturn = array();
    $parsedReturn['updateTime']=$updateTime;
    $parsedReturn['update']=$update;
    $parsedReturn['id']=$id;
    $parsedReturn['profile_image_url']=$profile_image_url;
    $parsedReturn['screen_name']=$screen_name;

    return $parsedReturn;
}
```

Save and close.

Edit twitteroauth.php

Now we can put our Search API request function code in. Open `twitteroauth.php` and add the following function at the end of the file but before the last `'}'`:

```
function search($options){
    $api_call = 'http://search.twitter.com/search.json?'.$options;
    return $this->get($api_call);
}
```

Did you notice that we are passing the variable `$options` to our API statement? We are going to populate that using JavaScript. To do so, we need to edit a few more files. Also did you notice how we defined the `$api_call` variable? We used a full path instead of the short hand we have been using. Remember that the OAuth library we are using will supply the default `https://api.twitter.com/1/` for us if we do not have http in our call. In this case we need to call `search.twitter.com` instead of `api.twitter.com`, thus the override.

Save and close.

Edit create_message.php

Open `create_message.php`. It's been a while since we have opened this file. We are going to make only a small change. We want to know when we are sending a normal message or requesting a search so that we can call the appropriate JavaScript function. We also want to keep the request in the text field after the page has reloaded. So, let's take care of that now.

Replace this line:

```
<textarea id="testinput" name='sendMessField' class='inputbox'
↳rows="2" cols="80" ></textarea>
```

With the following line:

```
<textarea id="testinput" name='sendMessField' class='inputbox'
↳rows="2" cols="80" ><? echo $_GET['options']; ?></textarea>
```

Remember that we are reloading this page, so this small bit of PHP will populate the text field box with whatever we typed in before. Now on to the next edit.

Find this line:

```
<input class='inputbox' type='button' value='Send'
↳onmouseup='sendMessage()' />
```

And replace it with the following code:

```
<? if($nav==search){ ?>
<input class='inputbox' type='button' value='Send'
↳onmouseup='sendSearch()' />
<? } else { ?>
<input class='inputbox' type='button' value='Send'
↳onmouseup='sendMessage()' />
<? } ?>
```

Here we are checking to see if we navigated to the Search tab, thus setting `$nav='search'`. If we want to submit the contents to search, we will call the JavaScript function `sendSearch` instead of `sendMessage` as we do for sending a tweet. Speaking of which, we should put some JavaScript code in place to catch this new `onmouseup()` request we just created.

Save and close this file.

Edit base.js

Open `base.js` and create a new function call after the `destroyFavorite()` call we created in the previous hour.

Put in the following code:

```
function sendSearch(){
    message = document.getElementById("prefex").innerHTML;
    message += document.sendMessForm.sendMessField.value;

    message=encodeURIComponent(message);

    trace(message);
    url = location.href.split("?");
    url = url[0]+'?nav=search&page=search&options='+message
    window.open(url, '_self');
}
```

This looks quite a bit like the JavaScript function `sendMessage()`, except we are not going to make an AJAX call here. Instead we are going to make a PHP request from our server passing the contents of our text box as an argument. We defined `url` as our current location, `location.href`; added `nav` and `page` identifiers, `?nav=search&page=search`; and finally, the contents of the text box, `&options='+message;`. This will allow us to experiment with Search options. Let's give it a try. Save and close this file.

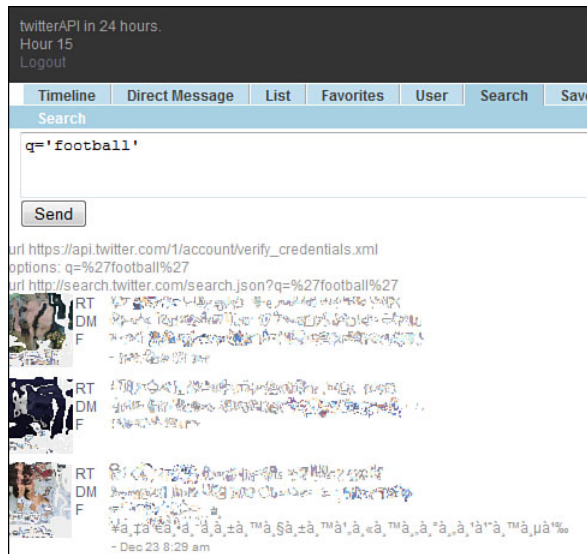
Open `index.php` and click the Search tab. Do not worry if you do not see anything. We haven't populated a search request yet, but we'll do that now. In the text field, type in the following:

```
q='football'
```

Then click the Send button. Do note that the quotes (either single or double) are optional here. You could have typed `q=football` and would get the same result.

You should get something like Figure 15.2.

FIGURE 15.2
Search result
from our appli-
cation.



Now let's try some of the options we used earlier in this hour. Try typing the following content into the Search text box and see what kind of results you get:

- ▶ q=so+cool
- ▶ q="cal+bears"
- ▶ q=+junk+#food
- ▶ q=devo&rpp=3
- ▶ q=+good+#cal+since:<today's date>

Note that when using the since operator, there is little reason to set the date more than two days older than today's date. A typical search will go back only a few days, depending on how the Twitter Search servers are feeling that day. As of this writing, you cannot use time with the since operator.

A Quick Guide to More Information on Search from the Twitter Docs

Now that we have a basic understanding of how Search works, let's look at some of the more detailed notes from the Twitter documents. The following section is a pretty good reference, but keep in mind that docs change and update all the time, so be sure to check the following link for the most updated information. Here we have

included only the core elements you should be aware of when thinking about your Search options. Let's again refer to the Twitter online docs (<http://dev.twitter.com/doc/get/search>)

GET Search

Required:

- ▶ **q**—Search query. Should be URL encoded. Queries will be limited by complexity:

<http://search.twitter.com/search.json?q=@noradio>

Optional:

- ▶ **callback**—Available only for JSON format. If supplied, the response will use the JSON format with a callback of the given name.
- ▶ **lang**—Restricts tweets to the given language, given by an ISO 639-1 code.
- ▶ **locale**—Specifies the language of the query you are sending (only ja is currently effective). This is intended for language-specific clients, and the default should work in the majority of cases:

<http://search.twitter.com/search.json?locale=ja>

- ▶ **rpp**—The number of tweets to return per page, up to a max of 100.

<http://search.twitter.com/search.json?rpp=100>

- ▶ **page**—The page number (starting at 1) to return, up to a max of roughly 1,500 results (based on $rpp * page$):

<http://search.twitter.com/search.json?page=10>

- ▶ **since_id**—Returns results with an ID greater than (that is, more recent than) the specified ID. There are limits to the number of tweets that can be accessed through the API. If the limit of tweets has occurred since the `since_id`, the `since_id` will be forced to the oldest ID available:

http://search.twitter.com/search.json?since_id=12345

- ▶ **until**—Optional. Returns tweets generated before the given date. Date should be formatted as YYYY-MM-DD:

<http://search.twitter.com/search.json?until=2010-03-28>

- ▶ **geocode**—Returns tweets by users located within a given radius of the given latitude/longitude. The location is preferentially taken from the Geotagging API, but will fall back to the Twitter profile. The parameter value is specified by “latitude,longitude,radius”, where radius units must be specified as either “mi” (miles) or “km” (kilometers). Note that you cannot use the near operator via the API to geocode arbitrary locations; however, you can use this geocode parameter to search near geocodes directly:

`http://search.twitter.com/search.json?geocode=37.781157,-122.398720,1mi`

- ▶ **show_user**—When true, prepends “:” to the beginning of the tweet. This is useful for readers that do not display Atom’s author field. The default is false.
- ▶ **result_type**—Optional. Specifies what type of search results you would prefer to receive. The current default is “mixed.” Valid values include the following:
 - ▶ **mixed**—Includes both popular and real-time results in the response.
 - ▶ **recent**—Returns only the most recent results in the response.
 - ▶ **popular**—Returns only the most popular results in the response.

`http://search.twitter.com/search.json?result_type=mixed`

`http://search.twitter.com/search.json?result_type=recent`

`http://search.twitter.com/search.json?result_type=popular`

Usage Notes

Search does have a few issues and rules you need to be aware of. Most are common sense, such as knowing that your query string must be URL encoded. Some are trial and error, like the second item saying queries are limited by complexity. Sometimes you do not know what “complex” means until you try it. So, do not assume you can string some long query combination. Always test every possible combination you want to support first. Returning back to the Twitter docs:

- ▶ Query strings should be URL encoded.
- ▶ Queries may be limited by complexity.
- ▶ Some users may be absent from search results.
- ▶ The `since_id` parameter will be removed from the `next_page` element because it is not supported for pagination. If `since_id` is removed, a warning will be added to alert you.

- ▶ This method will return an HTTP 404 error if `since_id` is used and is too old to be in the Search index.
- ▶ If you are having trouble constructing your query, use the advanced search form to construct your search, which is located here: <http://dev.twitter.com/console>. Then add the format. For example, <http://search.twitter.com/search?q=twitter> would become <http://search.twitter.com/search.json?q=twitter>.
- ▶ Applications must have a meaningful and unique User Agent when using this method. An HTTP Referrer is expected but not required. Search traffic that does not include a User Agent will be rate limited to fewer API calls per hour than applications including a User Agent string. More information on rate limiting can be found here: <https://dev.twitter.com/pages/rate-limiting>.

More information on Twitter Search best practices can be found here: <http://support.twitter.com/forums/10713/entries/42646>.

Notes on Metadata in Responses

The metadata node will sometimes contain a `result_type` field with a value of either “recent” or “popular”—although other values may be possible in the future. Popular results are derived by an algorithm that Twitter computes, and up to three will appear in “mixed mode” at the top of the resultset. Popular results will also include another node to metadata called “recent_retweets” and will indicate how many retweets the tweet was bestowed recently. The metadata node will contain more fields as time goes on.

Refreshing Search Results

For those using client-side search widgets, by default the first request might include popular results. If you want to display these, you can use the `result_type` attribute to visually differentiate them. If you don’t want to display these, you can always pass the “`result_type`” parameter with a value of “recent” along with your request, and they’ll never be included.

Summary

In this hour, we learned about the Search method and why it’s different from other Twitter method calls. We discovered that all searches are performed using only one method! Instead, Search queries are constructed through attributes passed through the single Search method call.

We also learned that despite having only one method call, we can use a rich variety of attributes for our search. However, we also discovered that there are certain restrictions on how many and what attributes we can use together in one call.

We also were introduced to the JSON protocol because XML is not supported in Search. In fact, many programmers prefer JSON over XML. So, we learned how to parse a JSON reply to be compatible with our rendering code.

Q&A

- Q.** *Search is the only API method. Will there be others in the future?*
- A.** No one knows for sure, but in the short term, no—it will be only Search.

Workshop

Quiz

1. Who is Summize?
2. Twitter Search uses only one API call. Why?
3. What is the operator to search back tweets by time?

Quiz Answers

1. It is the name of a company that created a Twitter Search technology that was purchased by Twitter. Hopefully, you got the Jeopardy reference.
2. The Summize product was originally created to search only Twitter; thus, it needs to perform only one function: Search.
3. Trick question. You cannot. You can only search by date, not time.

Exercises

Try these various operators listed below. Remember, these are examples. For example, using the exact date as listed from some of these examples may not work. Be aware of context.

Operator	Finds Tweets...
twitter search	containing both “twitter” and “search.” This is the default operator.
”happy hour”	containing the exact phrase “happy hour.”
love OR hate	containing either “love” or “hate” (or both).
beer -root	containing “beer” but not “root.”
#haiku	containing the hashtag “haiku.”
from: alexiskold	sent from person “alexiskold.”
to: techcrunch	sent to person “techcrunch.”
@mashable	referencing person “mashable.”
”happy hour” near: ”san francisco”	containing the exact phrase “happy hour” and sent near “san francisco.”
near: NYC within: 15mi	sent within 15 miles of “NYC.”
superhero since: 2010-09-01	containing “superhero” and sent since date “2010-09-01” (year-month-day).
ftw until: 2010-09-01	containing “ftw” and sent up to date “2010-09-01.”
movie -scary :)	containing “movie,” but not “scary,” and with a positive attitude.
flight :(containing “flight” and with a negative attitude.
traffic ?	containing “traffic” and asking a question.
hilarious filter:links	containing “hilarious” and linking to URLs.
news source:twitterfeed	containing “news” and entered via TwitterFeed.

This page intentionally left blank

HOUR 16

Trends and GEO

What You'll Learn in This Hour:

- ▶ What is a Twitter trend?
- ▶ Not all returns from the Trend method are in order based on time.
- ▶ What is the WOEID format?
- ▶ GEO methods.

What Is a Trending Topic?

Trends are an interesting aspect of Twitter that has been originated and copied by many other services. A trend is just a type of search; a term that is trending is simply a term that appears in higher frequency than other terms over a set amount of time. For example, during the World Cup final of 2010, when Holland played Spain, the terms “holland,” “spain,” and “worldcup” would have been trending highly because Twitter was almost brought to its knees by the amount of traffic during the final. As you would expect, a trending topic term, more commonly referred to as a *trending topic*, tends to reflect the major events of the day. What is really fascinating is that trending topics on Twitter tend to beat most news outlets for fast-breaking news.

Supporting Trends in Our Application

The API for Trends is a lot like Search given that it's based on Search. So, as you would expect, we are going to use JSON and various operators to fine tune what we get back for Twitter.

To start out, let's add a new menu item to our application to display finding topics.

Edit header.inc

This should be almost routine at this point. Let's add a new button to our UI. Open header.inc and add a new ` .. ` tag set with the following code:

```
<li><a href="/?nav=trends&page=trends" <? if($nav=='trends') echo
'class="here" '; ?> >Trends</a>
<? if($nav=='trends') { ?>
  <ul>
    <li><a href="/?nav=trends&page=trends" >Trends</a></li>
  </ul>
<? } ?>
</li>
```

Save and close.

Edit parseTwitter.php

Now that we have our tab, let's add a switch to support our new request. There are no options with this method, so we will make an empty function call. Open parseTwitter.php and add the following case statement at the end of our switch:

```
case 'trends':
    { $messages=$twitter->showTrends(); return call_trends($messages); }
break;
```

Notice that we are making a call to the function `call_trends()`. A new parsing function is required for the JSON return we get back from Twitter. So, let's create a new function called `call_trends()` and add it after the function `call_search()`. Here is the code for our new function:

```
function call_trends($messages){
    $twitterReturn = json_decode($messages);
    foreach($twitterReturn->trends as $status){
        $query = explode('?', $status->url);
        echo "<p /><a href='/?nav=search&page=search&options=$query[1]'">
Term: $status->name</a>";
    }
}
```

Let's go through this function because there are some tricky things going on behind the scenes that you should be aware of. First, let's have a look at a typical raw JSON return from the trends request using `var_dump()`:

```
<urlrequest>url
https://api.twitter.com/1/account/verify_credentials.xml</urlrequest><br>
➡<urlrequest>url
http://api.twitter.com/1/trends.json?</urlrequest><br>object(stdClass)#5
➡(2) {
  ["as_of"]=>
```

```
string(31) "Sat, 04 Sep 2010 04:45:05 +0000"
["trends"]=>
array(10) {
  [0]=>
  object(stdClass)#7 (2) {
    ["url"]=>
    string(53) "http://search.twitter.com/search?q=Duke+Nukem+Forever"
    ["name"]=>
    string(18) "Duke Nukem Forever"
  }
  [1]=>
  object(stdClass)#8 (2) {
    ["url"]=>
    string(51) "http://search.twitter.com/search?q=%23lessonlearned"
    ["name"]=>
    string(14) "#lessonlearned"
  }
  [2]=>
  object(stdClass)#9 (2) {
    ["url"]=>
    string(42) "http://search.twitter.com/search?q=Mitchie"
    ["name"]=>
    string(7) "Mitchie"
  }
  [3]=>
  object(stdClass)#10 (2) {
    ["url"]=>
    string(44) "http://search.twitter.com/search?q=Birdhouse"
    ["name"]=>
    string(9) "Birdhouse"
  }
  ...
  [9]=>
  object(stdClass)#16 (2) {
    ["url"]=>
    string(45) "http://search.twitter.com/search?q=Earthquake"
    ["name"]=>
    string(10) "Earthquake"
  }
}
```

Notice that the “url” string has a full URL to `http://search.twitter.com/search?q=<term>`. This is great if we want our users to go to Twitter. However, if we want to keep the users on our application, we are going to need to parse this string to get the query value the text after the `?q=`. So, we are using the PHP library call `explode` function to split all text after the `'?'` so that we can use our `Search` function that we created in Hour 15.

Save and close this file.

Edit twitteroauth.php

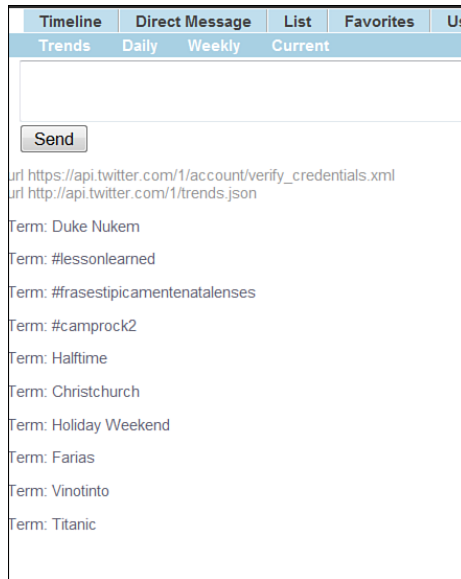
Let's add our API call. Open `twitteroauth.php` and add the following function at the end of the file but before the last `}`:

```
function showTrends(){
    $api_call = 'trends.json';
    return $this->get($api_call);
}
```

Save and close this file.

Now we'll give it a go. Open `index.php` in your browser and click Trends. You should see something like Figure 16.1.

FIGURE 16.1
Screenshot of
Trends.



Trends: Recent, Daily, Weekly

There are a few more Trend API calls we can make; however, its usefulness is most likely only within a select domain. The following three API calls return the same information in general: a collection of trending terms based on a segment of time—most recent, daily, and weekly. Let's have a look at the returns for `trend/daily.json`:

```
<urlrequest>url
https://api.twitter.com/1/account/verify_credentials.xml</urlrequest><br>
➡<urlrequest>url
http://api.twitter.com/1/trends/daily.json?</urlrequest><br>array(2) {
    ["trends"]=>
        array(24) {
```

```
[ "2010-09-04 03:40" ]=>
array(20) {
  [0]=>
  array(4) {
    [ "events" ]=>
    NULL
    [ "query" ]=>
    string(18) "Duke Nukem Forever"
    [ "promoted_content" ]=>
    NULL
    [ "name" ]=>
    string(18) "Duke Nukem Forever"
  }
  [1]=>
  array(4) {
    [ "events" ]=>
    NULL
    [ "query" ]=>
    string(14) "#lessonlearned"
    [ "promoted_content" ]=>
    NULL
    [ "name" ]=>
    string(14) "#lessonlearned"
  }
  [2]=>
  array(4) {
    [ "events" ]=>
    NULL
    [ "query" ]=>
    string(10) "#voltaxuxa"
    [ "promoted_content" ]=>
    NULL
    [ "name" ]=>
    string(10) "#voltaxuxa"
  }
  [3]=>
  array(4) {
    [ "events" ]=>
    NULL
    [ "query" ]=>
    string(10) "#camprock2"
    [ "promoted_content" ]=>
    NULL
    [ "name" ]=>
    string(10) "#camprock2"
  }
  ...
  [19]=>
  array(4) {
    [ "events" ]=>
    NULL
    [ "query" ]=>
```



```

        string(9) "Hurricane"
        ["promoted_content"]=>
        NULL
        ["name"]=>
        string(9) "Hurricane"
    }
}
["2010-09-04 05:40"]=>
array(20) {
    [0]=>
    array(4) {
        ["events"]=>
        NULL
        ["query"]=>
        string(10) "Duke Nukem"
        ["promoted_content"]=>
        NULL
        ["name"]=>
        string(10) "Duke Nukem"
    }
    [1]=>
    array(4) {
        ["events"]=>
        NULL
        ["query"]=>
        string(14) "#lessonlearned"
        ["promoted_content"]=>
        NULL
        ["name"]=>
        string(14) "#lessonlearned"
    }
}
...

```

The rest of the output is truncated.

There are 20 items per date. As you can imagine, this is quite a large dataset being returned. You may have also noticed that the key for the first array attribute is the date and time when those terms were trending. Let's have a look at the first five:

```

["2010-09-04 03:40"]=>
["2010-09-04 05:40"]=>
["2010-09-04 07:40"]=>
["2010-09-03 18:40"]=>
["2010-09-04 04:40"]=>

```

As you can see, the first three seem to respect a two-hour interval, but after that, it's fairly random which time slice is going to show up when. As such, you need to parse and convert these times into a date/time object so that you can sort them properly. In addition, the dates are not value pairs but the key of the nested array within the object. As such, we need to do a bit more work to parse these returns.

The value of looking at trending term data in time slices is to compare which terms are rising and which terms are declining and at what rate. Creating that capability in our application is a bit out of scope, but we still want to display the data, so let's instead display the returns separated by date but still linking to our search function.

Edit header.inc

Let's add a few more tabs to our Trends section. Under the following line:

```
<li><a href="/?nav=trends&page=trends" >Trends</a></li>
```

Add the following lines:

```
<li><a href="/?nav=trends&page=trends_daily" >Daily</a></li>
<li><a href="/?nav=trends&page=trends_weekly" >Weekly</a></li>
<li><a href="/?nav=trends&page=trends_current" >Current</a></li>
```

Save and close.

Edit parseTwitter.php

Now let's add three new case statements to our switch. Open parseTwitter.php and add these case statements at the end of the switch function:

```
    case 'trends_daily':
        { $messages=$twitter->showTrends_daily($options); return
call_trends_time($messages); }
        break;
    case 'trends_weekly':
        { $messages=$twitter->showTrends_weekly($options); return
call_trends_time($messages); }
        break;
    case 'trends_current':
        { $messages=$twitter->showTrends_current($options); return
call_trends_time($messages); }
        break;
```

Notice that all three are sending the returns to the same function. Let's create that function now.

In this same file, create a new function under the previous function we created called 'call_trends()':

```
function call_trends_daily($messages){
    $twitterReturn = json_decode($messages, true);
```

Notice that I have used the option 'true' while using json_decode. I'm doing this because it's sometimes easier to work with an array than with an object. Because we

want to display the date, we need the key as we loop through the nested arrays. The `array_keys()` function within PHP makes that very easy for us:

```
$keys=array_keys($twitterReturn['trends']);
```

Here we set `$keys` to the array of dates from our JSON return:

```
$i=0;
foreach($twitterReturn['trends'] as $foo){
    echo '<br><b>'. $keys[$i]. '<br></b> ';
    foreach($twitterReturn['trends'][$keys[$i]] as $trend){
```

Now that we know the time, we can access the nested array for each of the 20 returns per time segment:

```
        echo "<a
href='/?nav=search&page=search&options=q=".urlencode($trend['name']). "'>
Term: ".$trend['name']. "</a>";
        echo '<p />';
    }
    $i++;
}
}
```

Save and close the file.

Edit twitteroauth.php

Now that we have our parsing in place, let's insert the code to make the API requests to Twitter. Open `twitteroauth.php` and add the following three function calls at the end of the file but before the last `}`:

```
function showTrends_daily($options){
    $api_call = 'trends/daily.json?'.$options;
    return $this->get($api_call);
}
function showTrends_weekly($options){
    $api_call = 'trends/weekly.json?'.$options;
    return $this->get($api_call);
}
function showTrends_current($options){
    $api_call = 'trends/current.json?'.$options;
    return $this->get($api_call);
}
```

Save and close this file.

Now let's see what we get when we run our application. Open `index.php` in your web browser, and you should see something similar to Figure 16.2. Remember to scroll down the page to see more returns and times.

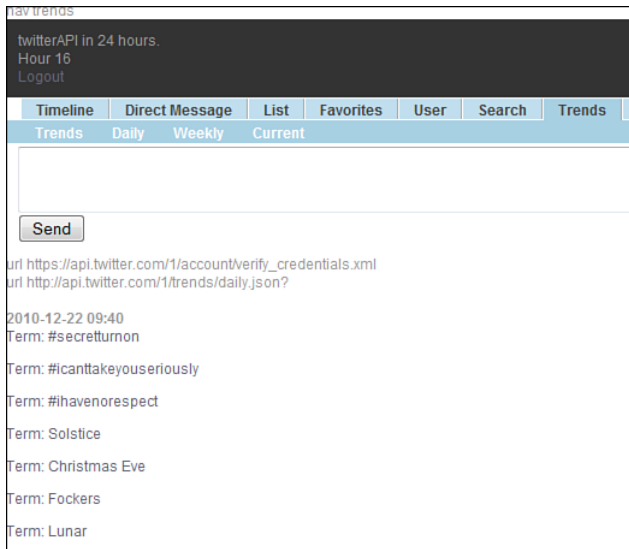


FIGURE 16.2
Screenshot of
Trends/daily.

Click Trends/weekly, and you will see something very much like Trends/daily but again with some of the dates out of sequence.

Clicking Trends/current returns something very much like the generic ‘trends’ API call, but based on a time slice in the recent past. Note that you will only get the top ten with this API. I expect this to change in the future.

Trends/Available and WOED

Let’s have a look at the API call Trends/available from Twitter documents: <http://dev.twitter.com/doc/get/trends/:woeid>.

The API call trends/available returns the locations that Twitter has trending topic information for. The response is an array of “locations” that encode the location’s WOED and some other human-readable information, such as a canonical name and country the location belongs in. A WOED is a Yahoo! Where On Earth ID.

Clearly there is something different here. The trends/available call relies on another API service from Yahoo! called “Yahoo! Where On Earth ID,” or WOED.

What is WOED?—WOED stands for Where On Earth ID by Yahoo!. It’s a stand-alone integer that represents a place. A more detailed explanation relevant to Twitter was posted at <http://engineering.twitter.com/2010/02/woeids-in-twitters-trends.html>, by Raffi Krikorian @raffi.

Basically, Twitter will try to determine a location for a tweet based on its content, not just the location of the tweet’s owner. This could be quite powerful and

interesting when studying large numbers of tweets. Keep in mind that this applies only to trending topics, not ALL tweets.

The actual API method call, `trends/:woeid` is pretty typical, as most of the API calls we have seen so far. It returns the top 10 trending topics for a specific WOEID, if trending information is available for it. The response is an array of “trend” objects that encode the name of the trending topic, the query parameter that can be used to search for the topic on Twitter Search, and the Twitter Search URL. Keep in mind that return for this call is cached for 5 minutes on the Twitter servers so calling it more frequently will not return any more data, and will count against your rate limit usage.

Example: <http://api.twitter.com/1/trends/23424975.xml>

Output:

```
<matching_trends type="array">
-
<trends as_of="2010-09-04T01:12:21Z" created_at="2010-09-04T00:54:16Z">
-
<locations>
-
<location>
<woeid>23424975</woeid>
<name>United Kingdom</name>
</location>
</locations>
<trend url="http://search.twitter.com/search?q=Duke+Nukem+Forever"
query="Duke+Nukem+Forever">Duke Nukem Forever</trend>
<trend url="http://search.twitter.com/search?q=Dries+Roelvink"
query="Dries+Roelvink">Dries Roelvink</trend>
<trend url="http://search.twitter.com/search?q=Vanessa+Feltz"
query="Vanessa+Feltz">Vanessa Feltz</trend>
<trend url="http://search.twitter.com/search?q=Blues+Brothers"
query="Blues+Brothers">Blues Brothers</trend>
<trend url="http://search.twitter.com/search?q=Heart+Vacancy"
query="Heart+Vacancy">Heart Vacancy</trend>
<trend url="http://search.twitter.com/search?q=%23lessonlearned"
query="%23lessonlearned">#lessonlearned</trend>
<trend url="http://search.twitter.com/search?q=Cloverfield"
query="Cloverfield">Cloverfield</trend>
<trend url="http://search.twitter.com/search?q=Glyn"
query="Glyn">Glyn</trend>
<trend url="http://search.twitter.com/search?q=Cyril"
query="Cyril">Cyril</trend>
<trend url="http://search.twitter.com/search?q=ITV4"
query="ITV4">ITV4</trend>
</trends>
</matching_trends>
```

There are other API calls with trends that can be found here: <http://dev.twitter.com/doc/get/trends>.

Many of the Trends API functions are still in development, so I would expect that there will be some evolution of these methods as time goes on. Similar to WOEID, another API method we can look at is the GEO tag.

Understanding the GEO Tag

The GEO methods are very interesting in that you can filter tweets based on location. The most obvious example is to map tweets over a map. A typical example is Trendsmap.com. This site will take your location, if enabled, and provide all trending topics in this area.

So, let's have a look at GEO API call from the Twitter docs.

GET geo/search

The geo/search API call will search for places that can be attached to a status/update. Given a latitude and a longitude pair, an IP address, or a name, this request will return a list of all the valid places that can be used as the `place_id` when updating a status.

This is the recommended method to use to find places that can be attached to statuses/update. Unlike `geo/reverse_geocode`, which provides raw data access, this endpoint can potentially reorder places in regard to the user who is authenticated. This approach is also preferred for interactive place matching with the user.

As we have seen before with other switch operator, there are GEO-centric search operators we can look at. Here's the current list from the Twitter docs (<http://dev.twitter.com/doc/get/geo/search>):

- ▶ `lat`
- ▶ `long`
- ▶ `query`
- ▶ `ip`
- ▶ `granularity`
- ▶ `accuracy`
- ▶ `max_results`
- ▶ `contained_within`
- ▶ `attribute:street_address`
- ▶ `callback`

A rich list of operators, right? Let's take one and see what it looks like. Here is an example search using the `attribute:street_address` operator:

```
http://api.twitter.com/1/geo/search.json?attribute:street_address=795%
↳20Folsom%20St
```

If we look at the sample JSON return from the Twitter docs, we can see that we get a rich amount of information. First, here is the sample return:

```
{
  "result": {
    "places": [
      {
        "name": "Twitter HQ",
        "country": "The United States of America",
        "country_code": "US",
        "attributes": {
          "street_address": "795 Folsom St"
        },
        "url": "http://api.twitter.com/1/geo/id/247f43d441defc03.JSON",
        "id": "247f43d441defc03",
        "bounding_box": {
          "coordinates": [
            [
              [
                -122.400612831116,
                37.7821120598956
              ],
              [
                -122.400612831116,
                37.7821120598956
              ],
              [
                -122.400612831116,
                37.7821120598956
              ],
              [
                -122.400612831116,
                37.7821120598956
              ]
            ]
          ],
          "type": "Polygon"
        },
        "contained_within": [
          {
            "name": "San Francisco",
            "country": "The United States of America",
            "country_code": "US",
            "attributes": {
```

```

    },
    "url": "http://api.twitter.com/1/geo/id/5a110d312052166f.JSON",
    "id": "5a110d312052166f",
    "bounding_box": {
      "coordinates": [
        [
          [
            -122.51368188,
            37.70813196
          ],
          [
            -122.35845384,
            37.70813196
          ],
          [
            -122.35845384,
            37.83245301
          ],
          [
            -122.51368188,
            37.83245301
          ]
        ]
      ],
      "type": "Polygon"
    },
    "full_name": "San Francisco, CA",
    "place_type": "city"
  }
],
"full_name": "Twitter HQ, San Francisco",
"place_type": "poi"
}
]
},
"query": {
  "url":
"http://api.twitter.com/1/geo/search.JSON?query=Twitter+HQ&accuracy=0&
autocomplete=false&granularity=neighborhood",
  "type":
"search",
  "params": {
    "granularity":
"neighborhood",
    "accuracy": 0,
    "autocomplete": false,
    "query":
"Twitter HQ"
  }
}
}
}

```


What is interesting here is getting not only an area bounding box for the address, but also for the city in general where the address is located. The number of options for searching is quite useful. As we can see from the preceding options list, we can use lat, long, ip, and—what I would think would be the most useful—query.

Summary

In this hour, we learned about trending topic and some of the limitations of how returns are presented to us. We also discovered that trends do not provide a timeline themselves, but instead a set of terms that are trending that we can thus use with the search method to get a timeline of tweets using that term.

We also learned how to search Twitter using GEO codes, including using the WOEID protocol created by Yahoo!. You will have discovered that the returns from the GEO methods are rich with information that lends itself nicely to a mapping application.

Q&A

Q. *Because a trending topic is much like a search, can I employ the same search parameters?*

A. No, although there are parameters you can use to refine the returns.

Q. *Will we see other GEO tools that we can use with Twitter other than WOEID?*

A. Yes, in fact, as of the writing of this book, a new set of reference tools is being Alpha tested.

Workshop

Quiz

1. What does WOEID mean?
2. What is a trending topic?
3. Why do hash keywords seem to appear in trending topics so often?

Quiz Answers

1. Yahoo! Where On Earth ID.
2. A trending topic is a keyword that has appeared repeatedly in a given bracket of time.
3. Remember, a hash (#) is an unofficial convention to designate the subject of a Tweet. As such, a subject will appear more often than standalone words.

Exercises

1. When we get a return from a GEO method call, we get a return type 'Polygon'. Take the information from this return and draw a polygon using Google Maps API.
2. Create a simple script to display the top 10 trending topic keywords. For each topic, place code that will allow the user to perform a search of that keyword.

This page intentionally left blank

HOUR 17

Friendships, Notification, Block, and Account Methods

What You'll Learn in This Hour:

- ▶ How to follow and unfollow a user with the Friendships methods
- ▶ Notification methods
- ▶ Block methods
- ▶ Account methods

Friendships Methods

The Friendships methods are what you would expect: the ability to create or destroy a friendship, which is also known as following and unfollowing a user. This is a core functionality and something that is expected to be supported by most Twitter applications. There are two ways to approach supporting the friendships. First, we can try to find out if we are already following the user, and based on that condition, offer a follow or unfollow button. Or, we can provide one button that assumes to follow, and if we get an HTTP reply of 403, meaning we are already following the Twitter user, we can offer to unfollow. The second procedure is more useful for automated systems, so we are going to go with the first option—checking to see if we are following someone and then offering the appropriate button.

Implement Friendships Methods in Our Application

Although the notification API seems straightforward, not all API methods return the status of whether you are following someone. As such, we will have to write a nested `if()` statement within our function call of `render.php`.

Edit `render.php`

Because we are using letters for our action buttons, we are going to use 'FOL' for a follow request and 'LEV' for a leave request.

After the following line:

```
$textBody = formatUpdates($update, $updateTime);
```

Add this line:

```
$following = $parsedReturn['following'][$i];
```

Now we need to add the code to render the 'LEV' link. In the same file, find the following code:

```
if($nav=='showFavorites') { $output.="<a href='#' on-
mouseup=\"destroyFavorite('$id')\">Del</a></div> ";
} else { $output.="<a href='#'
onmouseup=\"favorite('$id')\">Fav</a></div> "; }
```

And replace it with the following, removing the trailing '`</div>`'s:

```
if($nav=='showFavorites') { $output.="<a href='#'
onmouseup=\"destroyFavorite('$id')\">Del</a> ";
} else { $output.="<a href='#' onmouseup=\"favorite('$id')\">Fav</a> "; }
```

Now we can add our new code. Add the following under the code you just changed:

```
if($following==null) {$output.= '</div>'; } else {
    if($following=='true') { $output.="<a
➔href='#' onmouseup=\"leave('$screen_name')\">LEV</a></div> ";
    } else { $output.="<a href='#'
➔onmouseup=\"follow('$screen_name')\">Fol</a></div> "; }
}
```

Save and close.

Edit base.js

Now we need to catch these links in JavaScript. Add the following function after the function `sendGeoSearch()` in `base.js`:

```
function follow(id){
    url='commandLine.php?command=follow&id='+id;

    callPage(url, 'serverMessages');
}
function leave(id){
    url='commandLine.php?command=leave&id='+id;

    callPage(url, 'serverMessages');
}
```

Save and close.

Edit commandLine.php

Now we need to catch the AJAX request we made in `base.js`. We will use the same approach as we have for the other AJAX calls.

At the end of the file, but before the last `'`, add the following code:

```
case 'follow':
    { $messages=$twitter->follow($id); echo ($messages); }
    break;
case 'leave':
    { $messages=$twitter->leave($id); echo ($messages); }
    break;
```

Save and close.

Edit twitteroauth.php

Now let's put our notification methods into `twitteroauth.php`. Open `twitteroauth.php` and add these two functions at the end of the file but before the last `'`:

```
function follow($options){
    $sapi_call = "friendships/create.XML?screen_name=$options";
return $this->post($sapi_call);
}
function leave($options){
    $sapi_call = "friendships/destroy.XML?screen_name=$options";
return $this->post($sapi_call);
}
```

Save and close.

Edit parseTwitter.php

Finally, we need to parse our return to find out whether the owner of the tweet in the returned stream is following us. Not all APIs return this value, so we are only going to put code into our function `call_timeline()`. Fortunately, this is a simple matter of adding two more lines.

After the following line:

```
$screen_name[$i] = $status->user->screen_name;
```

Add this line:

```
$following[$i] = $status->user->following;
```

And after this line:

```
$parsedReturn[ 'screen_name' ]=$screen_name;
```

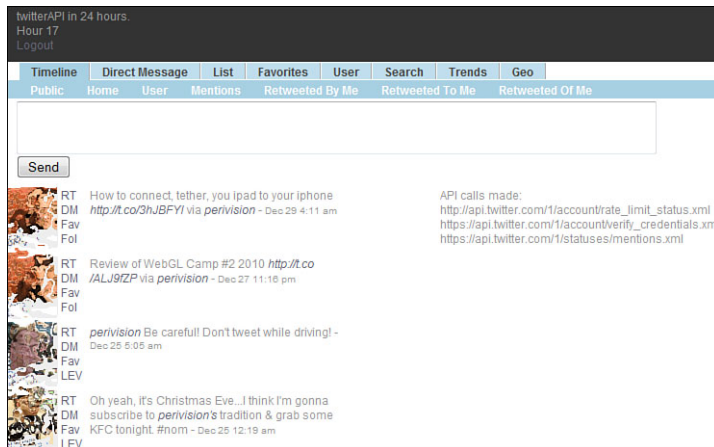
And this line:

```
$parsedReturn[ 'following' ]=$following;
```

Save and close.

Now let's give it a try. You should see something like Figure 17.1.

FIGURE 17.1
Screenshot of
Following
method but-
tons.



Try clicking on a FOL, and you should see a raw dump of the tweeter's information, as shown in Figure 17.2.

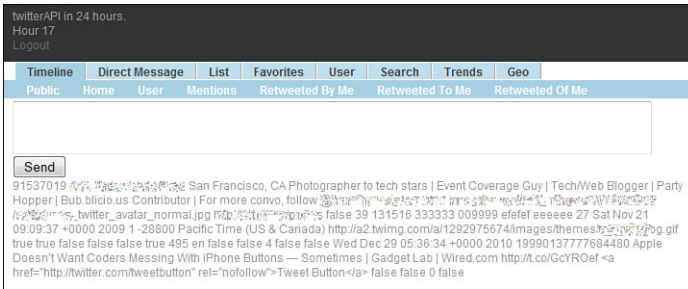


FIGURE 17.2
Screenshot showing user info from a notification call.

Notification Methods

The notification methods are somewhat like the account methods in that they are used to get information about or change the operation of a user's account. There are only two methods to review: notifications/follow and notifications/leave. They also work much the same as many of the API methods. Simply POST the proper http: request with either the user_id or screen_name, and you will get back a code 200 with profile information about the person you followed or left. Let's have a look at these methods and then we will add this feature to our application. First notifications/follow as defined in the Twitter docs:

- ▶ **POST notifications/follow:** Enables device notifications for updates from the specified user. Returns the specified user when successful.

We have a choice of operators for letting Twitter know who we want to follow. We can use either user_id or screen_name. In our example, let's use screen_name.

Did You Know?

Example:

OK, now let's have a look at the API from the Twitter docs about notifications/leave:

- ▶ **POST notifications/leave:** Disables notifications for updates from the specified user to the authenticating user. Returns the specified user when successful.

Pretty simple, right? Sometimes, it's just that easy.

Block Methods

Blocking occurs when a user blocks another user from following you, sending you a @reply or @mention, or putting your account on any of their lists. When users are blocked, they are not notified that they have been blocked. Keep in mind, though, if you have a public Twitter account, which most people do, your tweets will still show up in the public stream and thus a blocked user will still be able to see them.

The methods for blocking are very similar to friends and follows. So, let's look at these methods from the Twitter docs:

- ▶ **POST blocks/create:** Blocks the user specified in the ID parameter as the authenticating user. Destroys a friendship to the blocked user if it exists. Returns the blocked user in the requested format when successful.

Now, the next one may seem confusing and, if so, don't feel bad. To remove a block, you do not unblock or delete the block—you destroy it! Let's look at blocks/destroy from Twitter docs:

- ▶ **POST (DELETE) blocks/destroy:** Unblocks the user specified in the ID parameter for the authenticating user. Returns the unblocked user in the requested format when successful.

We can also see if a block exists. Again, from the Twitter docs:

- ▶ **GET blocks/exists:** Returns if the authenticating user is blocking a target user. Returns the blocked user's object if a block exists, and returns an error with an HTTP 404 response code otherwise.

This next API is a bit different; blocks/blocking returns a list of blocked accounts:

- ▶ **GET blocks/blocking:** Returns an array of user objects that the authenticating user is blocking.

The blocks/blocking/ids is just like the blocks/blocking API call except the return is user IDs:

- ▶ **GET blocks/blocking/ids:** Returns an array of numeric user IDs the authenticating user is blocking.

There are quite a number of methods for blocking, right? And you may not even know right off what you may use blocking for, but once you start to build more complete Twitter applications, or integrations into a larger application, you will find these to be more useful than you think.

Account Methods

As we have mentioned before, Twitter exposed almost all the elements of its functionality through its API system. This includes managing a user's account. We have already used one of these methods: `account/verify_credentials`. There are a few other useful credentials we can access. For example, `'account/rate_limit_status'` can be a useful warning to users that they are approaching the limit of how many API calls they can make for the hour or day.

Here is a list currently supported:

- ▶ `account/verify_credentials`
- ▶ `account/rate_limit_status`
- ▶ `account/end_session`
- ▶ `account/update_delivery_device`
- ▶ `account/update_profile_colors`
- ▶ `account/update_profile_image`
- ▶ `account/update_profile_background_image`
- ▶ `account/update_profile`

Just for fun, let's add a display in our application for displaying the rate limit of our user. We will make this call when we make the `account/verify_credentials` call.

Adding the Rate Status Method to Our Application

These next series of edits will be pretty simple because we are going to make a single API call and display the results, but no new menu items are needed. Also, it's good to note that making this API method call does not count against the user's API limit counts.

Edit header.inc

We'll do something a little different this time from a presentation point of view. We are going to use the space on the right of our dark header bar to display this information. Let's open `header.inc`.

Near the top of the file, find this line:

```
TwitterAPI in 24 hours. <br>
```

Then add the following lines:

```
<div style='float: right'>
  <? echo getTwitterData('getUserRate'); ?>
</div>
```

Save and close.

Edit parseTwitter.php

Because we make a request of `getTwitterData()`, we need to create a switch to catch it. Open `parseTwitter.php`, and add this case to the switch function:

```
case 'getUserName':
    { $messages=$twitter->getVerifyCredentials('XML'); return
    call_credentials($messages); }
    break;
```

Save and close.

Edit twitteroauth.php

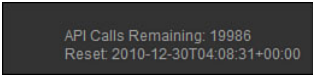
Now we need to create the API method call to support our new request. Open `twitteroauth.php` and at the end of the file, but before the last `'}`, add the following function:

```
function getUserRate() {
    $api_call = 'account/rate_limit_status.XML';
return $this->get($api_call);
}
```

Save and close.

Now let's give it a try. Open `index.php` in your browser, and you should see something like Figure 17.3. Do not worry if you do not have as many calls as shown in the figure. This is from a whitelisted account.

FIGURE 17.3
Screenshot
showing API
calls remaining.



API Calls Remaining: 19986
Reset: 2010-12-30T04:08:31+00:00

Additional Account Methods

For reference, let's have a look at some of the other methods we have access to via 'account' method calls from the Twitter docs. It is not likely you will use these methods in more typical Twitter client applications, but it's good to be aware of them. Again referencing the Twitter docs, let's have a quick look at some additional account methods that you have access to via the APIs:

- ▶ **POST account/end_session:** Ends the session of the authenticating user, returning a null cookie. Use this method to sign users out of client-facing applications like widgets.

URL: http://api.twitter.com/version/account/end_session.format

This does not always work because of aggressive OAuth caching. Always provide an option to log out from Twitter.com.

**Watch
Out!**

- ▶ **POST account/update_profile_colors:** Sets one or more hex values that control the color scheme of the authenticating user's profile page on Twitter.com. Each parameter's value must be a valid hexadecimal value and may be either three or six characters (ex: #fff or #ffffff).
- URL: http://api.twitter.com/version/account/update_profile_colors.format
- ▶ **POST account/update_profile_image:** Updates the authenticating user's profile image. Note that this method expects raw multipart data, not a URL to an image. This method asynchronously processes the uploaded file before updating the user's profile image URL. You can either update your local cache the next time you request the user's information, or, at least 5 seconds after uploading the image, ask for the updated URL using `users/profile_image/:screen_name`.

URL: http://api.twitter.com/version/account/update_profile_image.format

- ▶ **POST account/update_profile_background_image:** Updates the authenticating user's profile background image. Note that this method expects raw multipart data, not a URL to an image.

URL: http://api.twitter.com/version/account/update_profile_background_image.format

- ▶ **POST account/update_profile:** Sets values that users are able to set under the Account tab of their settings page. Only the parameters specified will be updated.

URL: http://api.twitter.com/version/account/update_profile.format

Summary

In this hour, we explored the Friendships and Notification methods, which are used to get information about or change the operation of a user's account. We looked at blocking and the methods associated with that account.

We also looked at the Account methods and how we can use some of those methods to track how many API calls a user has per hour. We also used a new portion of our UI to display API rate information in the upper right.

Q&A

- Q. *What API method would you use to change an account holder's email address?***
- A.** Trick question. You are not allowed to change a user's email address due to security reasons because a password reminder is sent to a user via email.
- Q. *What are the API method calls to follow and unfollow another twitter user?***
- A.** To follow a user: .. /friendships/create. To unfollow a user: .. /friendships/destroy.

Workshop

Quiz

1. What are the five profile colors methods we have access to?
2. When you block a user, will the user you block be notified?
3. Does the `rate_limit_status` API call count against your API limits?

Quiz Answers

1. They are as follows:
Background_color
Text_color
Link_color
Sidebar_fill_color
Sidebar_border_color

2. No, the user is not notified.
3. No, but if you create a loop making this call too often, you can run into limits of any calls coming from the same IP too often, as well as other traffic monitoring tools that are not publicly expressed or have yet to be deployed.

Exercises

1. Check to see how many tweets a user has made in the past 24 hours and change the background color based on that number. For example, blue could mean 1 or less in 24; red could be 40 tweets or more. Use color gradation based on the number of tweets.
2. In the `commandline.php` file, we used `echo` to see the return after creating a friendship. Parse this return and provide the user with something more useful.
3. When we reviewed notification methods, we did not implement them into our application. Try to do that now.

This page intentionally left blank

HOOR 18

Twitter Documentation

What You'll Learn in This Hour:

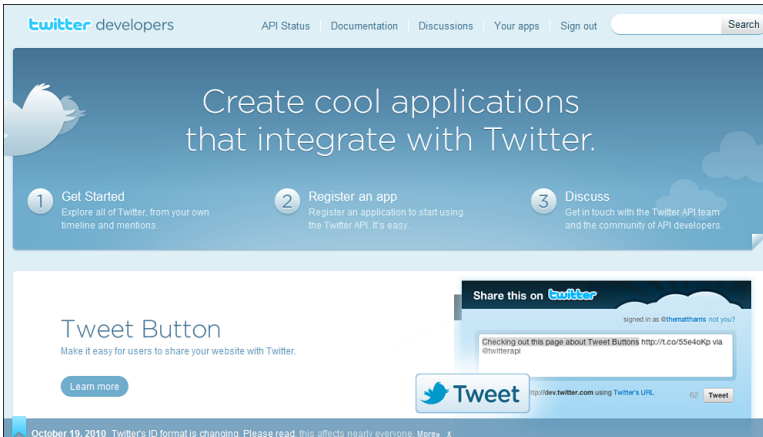
- ▶ How to navigate the Twitter Development website
- ▶ How to use `dev.twitter.com/console`
- ▶ About the Tweet Button widget
- ▶ How to navigate Twitter documents
- ▶ Some recommended best practices from Twitter

The Twitter Dev Website

Twitter has been taking great pains to make working with the Twitter API as clear as possible. However, as the API evolves, so does the documentation; as of this writing, the documentation is evolving. Much of the information in this book is a combination of references from the old docs and the new docs. Both versions of the document will still be online by the time this book is published, so be sure to pay attention to the URL when you look things up. It should be `dev.twitter.com`.

Currently, Figure 18.1 is the front page when you go to `dev.twitter.com`.

FIGURE 18.1
dev.twitter.com.



Getting Started

From dev.twitter.com, click Getting Started or navigate to dev.twitter.com/start. You might see something like Figure 18.2. Click Explore the API with the Twurl Web Console or navigate to dev.twitter.com/console. You should see something like Figure 18.3.

FIGURE 18.2
The
dev.twitter.com/
start page.



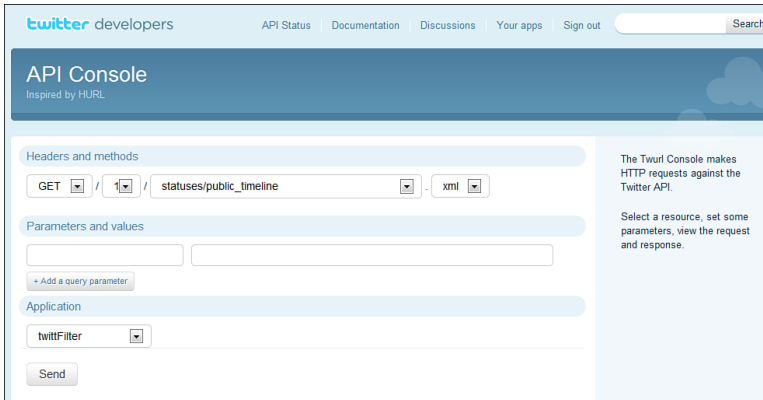


FIGURE 18.3
The dev.twitter.com/console page.

This is a useful tool for exploring exactly how the Twitter API is returning data at the moment. You need to have a registered application to use this tool. As you can see in this example, I'm using my own application twittFilter.com. If you have not done this already, you may want to refer back to Hour 2 on registering your application.

Something really useful with this tool is the display of the response headers. For example, here are the response headers I get when making the call `'..statuses/user_timeline'`, as you can see in Figure 18.4.

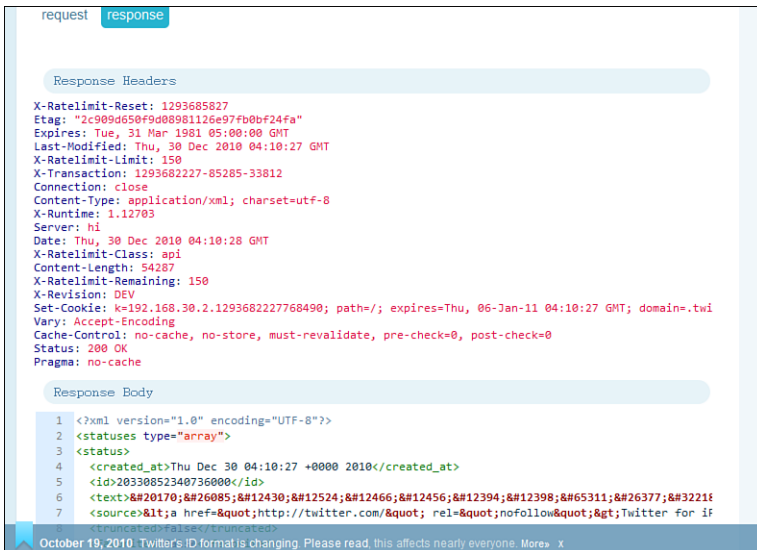


FIGURE 18.4
statuses/user_
timeline con-
sole response.

Response headers:

```
X-Ratelimit-Reset: 1284408284
Etag: "f739af2f2888af9..."
Expires: Tue, 31 Mar 1981 05:00:00 GMT
Last-Modified: Mon, 13 Sep 2010 19:04:44 GMT
X-Ratelimit-Limit: 150
X-Transaction: 1284404684-62667-59215
Connection: close
Content-Type: application/xml; charset=utf-8
X-Runtime: 0.71764
Server: hi
Date: Mon, 13 Sep 2010 19:04:45 GMT
X-Ratelimit-Class: api
Content-Length: 47666
X-Ratelimit-Remaining: 150
X-Revision: DEV
Set-Cookie: k=192.168.30.2.1284404684672016; path=/; expires=Mon, 20-Sep-10
19:04:44
    GMT; domain=.twitter.com, guest_id=1284404684xxxxx; path=/; expires=Wed,
13 Oct 2010
19:04:44 GMT, lang=en; path=/,
_twitter_sess=BAh7CjoPY3JlYXRlZF9hdGwrCI8nfAwrAToTcGFzc3dxxxxxx...; do-
main=.twitter.com; path=/
Vary: Accept-Encoding
Cache-Control: no-cache, no-store, must-revalidate, pre-check=0, post-
check=0
Status: 200 OK
Pragma: no-cache
```

Notice that we use the ‘status’ element of the response header to determine whether we received a valid response. In this case, we got a 200, which is ‘ok’.

Now let’s click the Request tab. You should see something like Figure 18.5. You can see the actual HTTP GET request that we normally see in our application:

```
opening connection to api.local.twitter.com...
opened
<- "GET /1/statuses/user_timeline.xml HTTP/1.1
Accept: */*
Connection: close
User-Agent: OAuth gem v0.3.4.1
Authorization: OAuth oauth_nonce="\moGd323sds...",
oauth_signature_method="HMAC-SHA1",
oauth_timestamp="1284404684", oauth_consumer_key="wIRkjfiul...",
oauth_token="1171057892...", oauth_signature="sAnXn1L6XY7s8s...",
oauth_version="1.0"
Host: api.local.twitter.com:9000
```

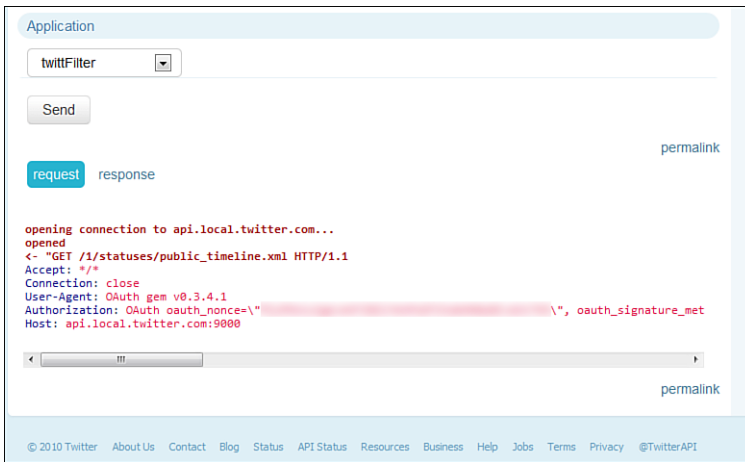


FIGURE 18.5
statuses/user_
timeline con-
sole request.

The Tweet Button

Let's refer back to the main page dev.twitter.com. As you saw in Figure 18.1, there is more on this page than navigation options. Currently, Twitter's Tweet button is featured. Because it's here, let's have a look at it.

The Tweet button is a simple JavaScript call that you can place on your web page to allow a user to retweet web content. For example, on my personal blog, "As seen through PeriVisioN," I have a retweet button for each article, as shown in Figure 18.6.

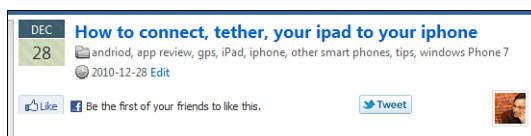


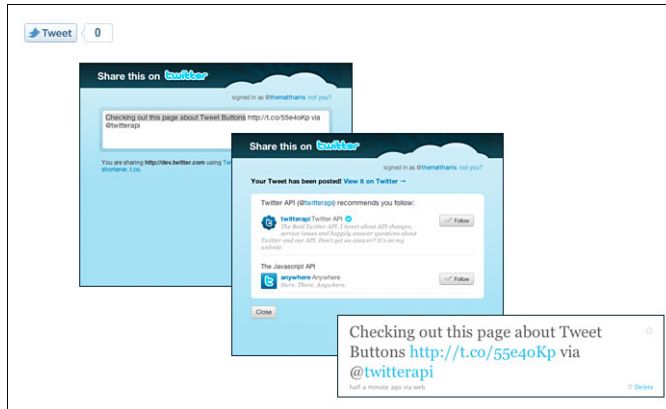
FIGURE 18.6
Retweet button
on PeriVisioN.

The user experience for using the Tweet button is standard, as you can see from the following steps and Figure 18.7:

1. The user clicks the Tweet button.
2. The user is asked to log in to Twitter if the user isn't already logged in. The user who is new to Twitter can also create an account.
3. The Share box appears already completed with the information provided in the properties of the Tweet button. Users can change the content if they wish.

4. Posting of the Tweet is confirmed, and the user is suggested a maximum of two accounts the user may want to follow as provided in the properties of the Tweet button.
5. The Share box remains open until the user clicks Close.

FIGURE 18.7
Retweet button
experience.



Implementation is also easy. Here are three ways you can do it, as posted on http://dev.twitter.com/pages/tweet_button.

Using JavaScript

The easiest way to add the Tweet button to your website is to use JavaScript. This method requires adding a line of JavaScript and an HTML anchor to your web page. With this method, you can customize the Tweet button using data attributes and query string parameters.

Notice how the anchor element has a class of `twitter-share-button`. This is required for the Tweet button JavaScript to know which anchor elements to convert to buttons:

```
<script src="http://platform.twitter.com/widgets.js"
type="text/javascript"></script>
<a href="http://twitter.com/share" class="twitter-share-button">Tweet</a>
```

Using an iframe

If you prefer, you can add a Tweet button using an `iframe`. When using this method, you have to use query string parameters to customize the Tweet button's behavior:

```
<iframe allowtransparency="true" frameborder="0" scrolling="no"
src="http://platform.twitter.com/widgets/tweet_button.html"
style="width:130px; height:50px;"></iframe>
```

Building Your Own

If you want to be able to customize the way the Tweet button looks, you will want to use this basic format. When using this method, you have to use query string parameters to customize the Tweet button's behavior as well as handle the pop-up of the Share box.

The dimensions of the Share box are listed here in the FAQ: http://dev.twitter.com/pages/tweet_button_faq#dimensions.

Dev.twitter.com/doc

The current set of Twitter API docs that you would find on dev.twitter.com is not the same documents that Twitter started with. In fact, this collection is fairly recent; the original site was created using a wiki: <http://apiwiki.twitter.com/>. A screenshot of the original documentation resource can be seen in Figure 18.8, in contrast to the new system shown in Figure 18.9.

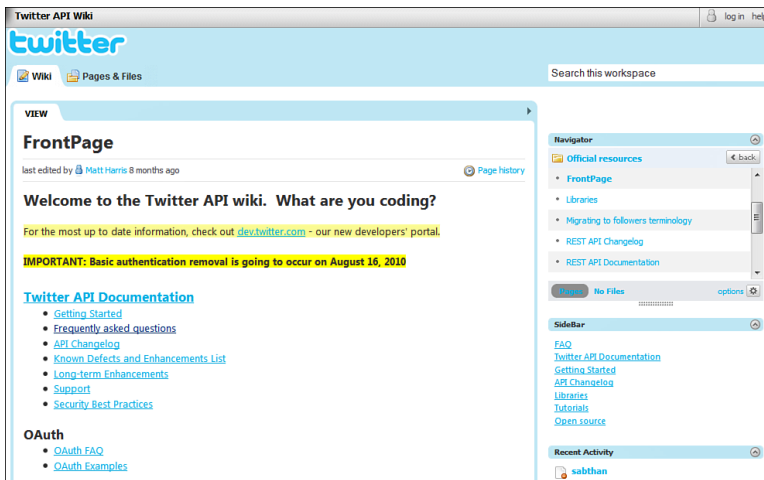
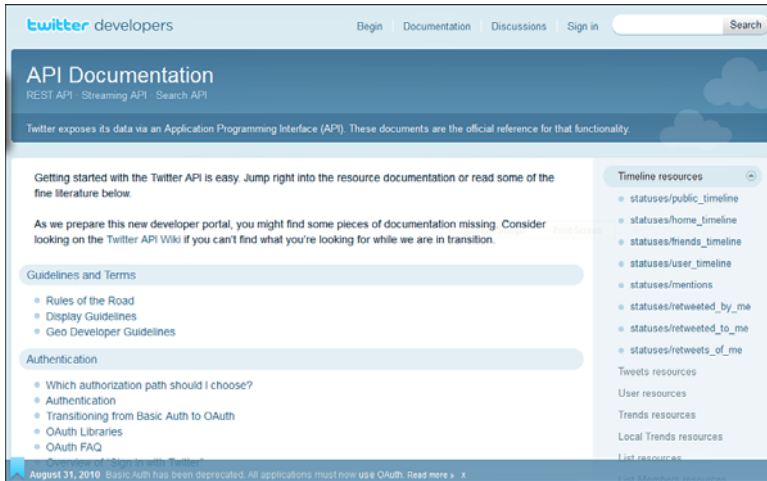


FIGURE 18.8
Twitter API wiki.

FIGURE 18.9
New twitter API
page.



At the time of this writing, all the API methods have been moved, but not everything has been fully documented, and there are a few errors, omissions, and even grammatical errors, of which we are sure we have plenty ourselves, in the new document. It is anticipated that these errors will go away as the new documents mature. It's not in the scope to go through all these documents, but instead we'll highlight a few interesting and generally useful resources.

Twitter Resource Page Overview

Although the API Documentation (<http://dev.twitter.com/doc>) is still being created and refined, these documents are still a great resource for any programmer or product developer. The Twitter docs are broken up into seven sections:

- ▶ Guidelines and Terms
- ▶ Authentication
- ▶ REST API and General
- ▶ Streaming API and User Streams
- ▶ Search API
- ▶ @Anywhere and Tweet Button
- ▶ Ecosystem

Guidelines and Terms

There are three sections to Guidelines and Terms: Rules of the Road, Display Guidelines, and Geo Developer Guidelines.

Rules of the Road (http://dev.twitter.com/pages/api_terms) are “a set of (“Rules”) that describe the policies and philosophy around what type of innovation is permitted with the content and information shared on Twitter.” This section is the meatiest of the Guidelines. Here we can read about how and where you can use Twitter data. You will also discover what you CANNOT do with Twitter data, including trying to sell access to the Twitter API, and if you provide an API that returns Twitter data, you can only return IDs. Also from the docs, “Exporting Twitter Content to a data-store as a service or other cloud-based service, however, is not permitted.” So, be sure you read ALL of the Twitter Content section before you even start to develop your application. You will also find information on commercial use as well as legal terms.

The Display Guidelines (http://dev.twitter.com/pages/display_guidelines) section is pretty straightforward and reflects the recommended way to display a tweet. You will notice that our application already follows most of these guidelines. At this point in the book, you should feel confident to make the modifications to satisfy this display requirement.

The Geo Developer Guidelines (http://dev.twitter.com/pages/geo_dev_guidelines) are more around privacy about a user’s location as well as certain functions you should allow the user.

Authentication

We discussed OAuth in Hour 8. However, it is still a good idea to check the OAuth FAQ every now and again to see what has changed:

- ▶ Which authorization path should I choose?
- ▶ Authentication
- ▶ Transitioning from Basic Auth to OAuth
- ▶ OAuth Libraries
- ▶ OAuth FAQ
- ▶ Overview of “Sign in with Twitter”

It is strongly recommended that you look at the “OAuth Libraries” section to look for a more recent and feature-rich OAuth library than we used here now that you have a good idea of how OAuth works.

REST API and General

This area is where most programmers spend their time, and not just within the API section. Because these are new documents, a few corrections and annotations are expected:

- ▶ Recently Updated Documentation

Whenever you find that something is not working the way you expect, it's a good idea to check the Recently Updated section in case something changed, updating your previous understanding on how the API functions.

- ▶ Introduction to @twitterapi

A nice slideshow from @raffi on Twitter API development.

Watch Out!

There are 99 slides in the deck at the time of this writing. Although they go fast, pace yourself.

- ▶ Things Every Developer Should Know
- ▶ Twitter API FAQ
- ▶ API Overview
- ▶ API Support
- ▶ Security Best Practices

Whether you are a first-time coder or have been writing for years, it's a good idea to always have security in mind. Twitter provides a good checklist of things to keep in mind when building and testing your application. Even the outline from the site on its own serves as a good check-off list.

- ▶ Rate Limiting
- ▶ Rate Limiting FAQ
- ▶ HTTP Responses and Errors
- ▶ Counting Characters
- ▶ Tweet Entities

We did not spend much time on Tweet Entities in our previous hours because the feature is still in development and not fully live yet. Tweet Entities are additional structured data elements that automatically identify and reformat various known

Twitter conventions for you. For example, if a hyperlink is discovered within a tweet, that hyperlink will be provided via an attribute within JSON or XML. If the link is shorted, the full link will be provided as well. This is becoming necessary as well as useful to guard against malicious links being hidden in a URL-shortened link.

From Wikipedia

URL shortening is a technique on the World Wide Web in which a URL may be made substantially shorter in length. This involves using an HTTP Redirect on a domain name that is short to link to a website that has a long URL. Domain names are linked to IP addresses; however, in order to be human friendly, the Latin alphabet can be used. Sometimes this can lead to extremely long URLs. For example, the URL <http://en.wikipedia.org/w/index.php?title=TinyURL&diff=283621022&oldid=283308287> can be shortened to <http://tinyurl.com/mmw6lb>.

**By the
Way**

Streaming API Documentation and Search API

Streaming on Twitter is normally used only in very specific applications, and as such was not included in this book in detail. However, we will discuss it in Hour 19. As for search, we covered much of these sections in previous hours; however, the search API section is a very handy reference section for building a search query, so that is worth bookmarking.

@Anywhere and Tweet Button

@Anywhere and the Tweet Buttons are Twitter's early attempts at creating widgets for making the adding of Twitter functionality to a website easier. @Anywhere is an interesting effort by Twitter that will allow you to place Twitter widgets within other applications. For example, one of the @Anywhere widgets is a 'hovercard'. This widget pops up a simple Twitter card upon rollover, as shown in Figure 18.10.

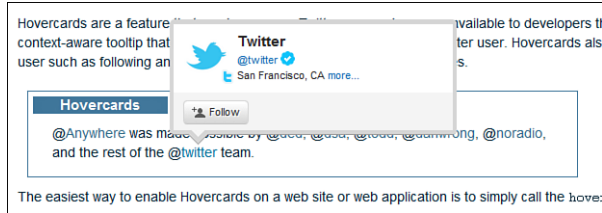
The code to implement this is pretty easy:

```
<script type="text/javascript">

    twttr.anywhere(function (T) {
        T.hovercards();
    });

</script>
```

FIGURE 18.10
@Anywhere
hovercard



Watch Out!

Twitter recommends that you set and register a separate application for use with @Anywhere

Ecosystem

This section contains information about non-programmatic issues, like libraries, where to find Developers, mention of @twitterapi - a twitter account about the Twitter API, and a mailing list.

As a developer, I'm on one of the mailing lists, but I caution you, there is a lot of traffic on these lists. I would strongly recommend that you select the option to receive only one email a day.

Summary

In this hour, we looked at many of the API resources that are made available on dev.twitter.com. Many of these documents we have referenced before. We explored the console page and looked at the Twitter widget Reweet button.

Also in this hour, we looked at the online Twitter docs and explored how they are structured. We also touched on recommended best practices and security recommendations.

Q&A

- Q.** *Because the old API docs on apiwiki.twitter.com/ are being deprecated in favor of dev.twitter.com/docs, can we ignore the apiwiki completely?*
- A.** At some point, yes, but at the time of this writing, a few small details in the old docs have not made it to the new ones. So, if you do not find what you looking for, it may be worth the time to try the older docs.

Q. *If the user does not have Tweeting With Location enabled, can I enable via API calls?*

A. At this time, you cannot, nor do I think you will able to in the future, but that could change.

Workshop

Quiz

1. If you wanted to look at response headers, what tool does Twitter offer to help with this?
2. What is a URL shortener?
3. What are the “Rules of the Road?”

Quiz Answers

1. [Dev.twitter.com/console](https://dev.twitter.com/console).
2. URL shortening is a technique on the Web in which a URL may be made substantially shorter in length. This involves using an HTTP Redirect on a domain name that is short to link to a website that has a long URL.
3. A set of (“Rules”) that describes the policies and philosophy around what type of innovation is permitted with the content and information shared on Twitter. Make sure you read them.

Exercise

We did not do this in our sample program, so as an exercise, put the Retweet button in your application. Use either the JavaScript or iframe version.

This page intentionally left blank

HOUR 19

Streaming API

What You'll Learn in This Hour:

- ▶ The three types of streaming APIs
- ▶ The user streams and site streams
- ▶ The Firehose, Gardenhose, Birddog, and Shadow

The Three Types of Streaming APIs

Because search was never designed to support large-scale queries or the volume of tweets Twitter has seen recently, the streaming architecture was put in place.

There are three Twitter API sets: the RESTful APIs, Search, and the streaming APIs. Streaming is as it sounds, opening up a connection to the Twitter servers and getting a stream of information back.

There are three main streaming products: the Streaming API, user streams, and sites streams. How we would call and work with each type of stream is basically the same, but each one has its own particular focus. Let's have a look at these as defined in the Twitter docs. http://dev.twitter.com/pages/streaming_api:

- ▶ **Streaming API**—Public statuses from all users, filtered in various ways: by userid, by keyword, by random sampling, by geographic location, and so on.
- ▶ **User streams**—Nearly all data required to update a user's display. This requires the user's OAuth token. It provides public and protected statuses from followings, direct messages, mentions, and other events taken on and by the user. A large number of user streams may not be created from the same host or service. For example, an application that displays a few

accounts at once may open a connection per account. The primary use case is providing updates to a Twitter client.

- ▶ **Site streams**—Allows multiplexing of multiple user streams over a site stream connection. When more than a handful of user stream connections are opened from the same host or service, site streams must be used. The primary use case is website and other service integrations.

Although there are three types of streams listed, the streaming and user streams API's would be the ones you would most likely be interested in. You can think of the Streaming API and the incoming stream from Twitter and the user streams as outgoing or what you would provide to Twitter. We will get into the particulars of how to use these API calls later in this hour.

Who Would Normally Use Streaming?

Streaming is mostly targeted to people who make the same search over and over or who need a large dataset for research or data analysis. For example, suppose you want to track the performance of a keyword over time. You would open a stream and pass the results through a set of business rules and thus update a log file(s) or a database. Trends are done somewhat like this. A stream of tweets is tracked and all found keywords are logged, so you could create your own trending application using streaming.

As stated in the Twitter docs, each of the three products has its intended uses.

The Streaming API is very useful for data analysis. To discover interesting trends in the Twitter universe, you may need quite a bit of data. For example, a very interesting experiment was done to track the movement of the swine flu virus by tracking people who tweeted the words “swine” or “flu” and displaying that information on a map.

The user streams product has a different objective. It's intended to give you everything you need or want to know about a user. This includes nonpublic information like DMs, private tweets, and so on, so you will need the user's OAuth information. Remember, we already have access to this information through our Twitter application for anyone who logs in. The user streams enable us to get much of the information that takes many API calls using just one call.

Site streams is basically user streams en masse. A vast majority of developers would not need to use site streams, but large-scale integration operations or those clients with many users would use this product.

What Makes Streaming Different?

The Streaming API methods are similar to search methods, with one major exception. This is a streaming connection, not your normal REST connection where we pass a request and we get a response. Instead, we are opening a socket connection (or a stream) to Twitter, and the Twitter servers will happily continue to feed us data for as long as we (or Twitter) keep the connection open. Keep in mind that in comparison to REST where you parse the complete response when the connection is closed, you just create a single persistent connection that has responses separated by carriage returns `\r` (http://dev.twitter.com/pages/streaming_api_concepts#parsing-responses).

It's important to stress that you only get one connection per account. From the Twitter docs: "Each account may create only one standing connection to the Streaming API. Subsequent connections from the same account may cause previously established connections to be disconnected. Excessive connection attempts, regardless of success, will result in an automatic ban of the client's IP address. Continually failing connections will result in your IP address being blacklisted from all Twitter access."

Pre-Launch Checklist

Within the Twitter docs we visited in the previous section, there is a checklist of things to consider before you start your streaming request. At the time of this writing, there are 11 items on the checklist. Why 11 and not the traditional 10? Actually, it was 10 when we first started writing this book, and now they have added another one—yet another indication of how fast things with Twitter change and get updated. That or it's an homage to Spinal Tap. Look it up.

1. Using the correct product?
2. Not purposefully attempting to circumvent access limits and levels?
3. Creating the minimal number of connections?
4. Avoiding duplicate logins?
5. Avoiding needless reconnection?
6. Backing off from failures: none for first disconnect, seconds for repeated network (TCP/IP) level issues, minutes for repeated HTTP (4XX codes)?
7. Using long-lived connections?
8. Tolerant of other objects and newlines in markup stream (non `<status>` objects...)?

9. Tolerant of duplicate and out-of-order messages?
10. Carefully monitoring the Twitter-API-Announce mailing list and using the Twitter-Dev-Talk list for questions?
11. Following @twitterapi and the @sitestreams accounts?

Item 6 is of the most important in my opinion, although I'm sure you will not be making any friends by violating item 2. Before you make *any* calls, make sure you have a timeout switch to kill the connection before trying to make a new one. Although you can use cURL to make this call, using cURL tends to have operational gotchas, so it's best to use cURL for debugging. It would also be a good idea to limit how long you keep the stream open if you are storing the information to a file. Keep in mind that Twitter will hold a connection indefinitely short of server-side errors. However, what item 6 is referring to is failing gracefully. There are plenty of reasons a connection can fail: network errors, server overload, or perhaps your client cannot keep up with the traffic flow. The best solution is to look for any HTTP response code that is over 200 and have your script pause before trying to reconnect again. It is suggested to start at 250 milliseconds and continue to up the wait time to around 16 seconds. I would recommend that after 10 consecutive errors, send a message to a human to look at the issue. Keep a log of the errors.

Parsing JSON responses from the Streaming API is simple: Every object is returned on its own line and ends with a carriage return. Newline characters (`\n`) may occur in object elements (the `text` element of a status object, for example), but carriage returns (`\r`) should not.

Not listed on the checklist, but probably should be, is planning for growth. Every time we think Twitter has gotten as big as it's going to get, it doubles again. You should make sure that whatever system you put in place to read and process the stream has enough hardware power and storage space to handle three times whatever you are experiencing now.

Streaming Methods

There are a number of API methods available to us within the `stream.twitter.com` API call. Each type of stream is geared to a particular use. Unlike any of the other API calls we have seen in Twitter, these types of streams all have names. For example, the most common and simple method to use is called Spritzer. This API stream provides a sample of tweets from the Twitter stream.

Keep in mind that sometimes a status will come with a delete status or a delete location. Twitter requests that you honor this delete status and remove statuses from your client and storage. Some of these delete and scrub requests can come significantly out of order. Try to account for this. Here are examples of the delete status and delete geo attributes.

Delete Status:

```
JSON : { "delete": { "status": { "id": 1234, "user_id": 3 } } }
```

Delete Geo (scrub):

```
JSON: {"scrub_geo":{"user_id":14090452,"up_to_status_id":23260136625}}
```

Remember that the API is changing all the time, so make sure your code can handle unexpected attributes that will be coming in the future.

**Watch
Out!**

Because of volume, the Twitter stream may not provide each status update in exactly the correct chronological order. It is not unusual to have status updates delivered 3 seconds or more out of order. If the order of the status is critical to your application, you may want to delay updates for 5 seconds to be sure that statuses are presented based on creation date, not the order it was delivered.

**Watch
Out!**

Types of Stream Methods

Given the enormous volume of status updates flowing through the Twitter servers, it's not necessary or practical to provide everyone with a stream of every status update. As such, Twitter provides different levels of samples from the stream. All accounts may access the statuses/sample and statuses/filter methods at default access levels. To apply for greater level access, refer to the support page (<http://dev.twitter.com/pages/support>) or email Twitter directly at api@twitter.com.

The Streaming methods are similar to the search methods, including supporting only JSON. Let's have a look at these methods as currently documented in Twitter docs (http://dev.twitter.com/pages/streaming_api_methods).

POST statuses/filter: Returns public statuses that match one or more filter predicates. At least one predicate parameter, follow, locations, or track must be specified. Multiple parameters may be specified, which allows most clients to use a single connection to the Streaming API. Placing long parameters in the URL may cause the request to be rejected for excessive URL length. Use a POST request header parameter to avoid long URLs.

There are various levels of access (or roles) you can apply for with the API filter:

- ▶ The default access level allows up to 400 track keywords, 5,000 follow userids, and 25 0.1-360 degree location boxes.
- ▶ Shadow role allows 100,000 follow userids.
- ▶ Birddog role allows 400,000 follow userids.
- ▶ Restricted track role allow for 10,000 track keywords.
- ▶ Partner Track role allows for 200,000 track keywords.
- ▶ LocRestricted role supports 200 0.1-360 degree location boxes.

Any increased track access levels also pass a higher proportion of statuses before limiting the stream. You can apply for greater tracking and access by filling out the online form currently found here:

<https://spreadsheets.google.com/viewform?hl=en&formkey=dFBTbHZIMVhseUtqS2NkT283RTluX3c6MQ&ndplr=1#gid=0>.

Here is an abbreviated list of parameters statuses/filter supports. Details on these parameters can be found in the referenced Twitter docs:

- ▶ **count**—Indicates the number of previous statuses to consider for delivery before transitioning to live stream delivery.
- ▶ **delimited**—Indicates that statuses should be delimited in the stream. Statuses are represented by a length, in bytes, a newline, and the status text that is exactly length bytes.
- ▶ **follow**—Returns public statuses that reference the given set of users. Users are specified by a comma-separated list.
- ▶ **locations**—Specifies a set of bounding boxes to track. Only tweets that are both created using the Geotagging API and are placed from within a tracked bounding box will be included in the stream.
- ▶ **track**—Specifies keywords to track. Keywords are specified by a comma-separated list.

GET statuses/firehose: Returns all public statuses. The Firehose is not a generally available resource. Few applications require this level of access. Creative use of a combination of other resources and various access levels can satisfy nearly every application use case.

Here is an abbreviated list of parameters statuses/firehose supports. Details on these parameters can be found in the referenced Twitter docs:

- ▶ **count**—Indicates the number of previous statuses to consider for delivery before transitioning to live stream delivery.
- ▶ **delimited**—Indicates that statuses should be delimited in the stream. Statuses are represented by a length, in bytes, a newline, and the status text that is exactly length bytes.

GET statuses/links: Returns all statuses containing http: and https:. The links stream is not a generally available resource.

Here is an abbreviated list of parameters statuses/retweet supports. Details on these parameters can be found in the referenced Twitter docs:

- ▶ **count**—Indicates the number of previous statuses to consider for delivery before transitioning to live stream delivery.
- ▶ **delimited**—Indicates that statuses should be delimited in the stream. Statuses are represented by a length, in bytes, a newline, and the status text that is exactly length bytes.

GET statuses/retweet: Returns all retweets. The retweet stream is not a generally available resource. Few applications require this level of access. Creative use of a combination of other resources and various access levels can satisfy nearly every application use case.

Here is an abbreviated list of parameters statuses/retweet supports. Details on these parameters can be found in the referenced Twitter docs:

- ▶ **delimited**—Indicates that statuses should be delimited in the stream. Statuses are represented by a length, in bytes, a newline, and the status text that is exactly length bytes.

GET statuses/sample: Returns a random sample of all public statuses. The default access level provides a small proportion of the Firehose. The “Gardenhose” access level provides a proportion more suitable for data mining and research applications that desire a larger proportion to be statistically significant sample.

Here is an abbreviated list of parameters statuses/sample supports. Details on these parameters can be found in the referenced Twitter docs:

- ▶ **count**—Indicates the number of previous statuses to consider for delivery before transitioning to live stream delivery.

- ▶ **delimited**—Indicates that statuses should be delimited in the stream. Statuses are represented by a length, in bytes, a newline, and the status text that is exactly length bytes.

Summary

In this hour, we learned about the Streaming API and the three main products of the Streaming API: the Streaming API, user streams, and site streams. We also learned about a very useful pre-launch checklist provided by Twitter.

In addition, we learned about filtering the stream and the various parameters we can employ.

Q&A

- Q.** *If I'm building a normal Twitter client, do I really need to employ streaming?*
- A.** No. Streaming is typically used for special purposes where you need lots of data for analysis.
- Q.** *What is the advantage of using streaming?*
- A.** In addition to a far greater amount of data being available to you, it also is a persistent connection, meaning that you do not have to keep making cURL calls to maintain a flow of data. If your application requires real-time data or a large amount of data, streaming is worth the extra work required to support it.

Workshop

Quiz

1. Twitter gave you a list of 11 things you need to consider before you start streaming. Can you list five of them without looking back at the text?
2. What are the sampling rates for Spritzer and Gardenhose?
3. What type of connection gives you everything?

Quiz Answers

1. Things to consider before streaming include the following:
 - ▶ Using the correct product?
 - ▶ Not purposefully attempting to circumvent access limits and levels?
 - ▶ Creating the minimal number of connections?
 - ▶ Avoiding duplicate logins?
 - ▶ Avoiding needless reconnection?
 - ▶ Backing off from failures: none for first disconnect, seconds for repeated network (TCP/IP) level issues, minutes for repeated HTTP (4XX codes)?
 - ▶ Using long-lived connections?
 - ▶ Tolerant of other objects and newlines in markup stream (non <status> objects...)?
 - ▶ Tolerant of duplicate and out-of-order messages?
 - ▶ Using JSON if at all possible?
 - ▶ Planning for growth.
2. ~1% for Spritzer, ~5% for Gardenhose.
3. Firehose, but this is only by request.

Exercises

Create a test page that opens up a stream for 3 seconds and then displays the result. Do not worry about displaying formatting for this exercise; the key is to open a connection, process the data, and then close the connection. After you have done it for 3 seconds, try 15, and then try 5 minutes. You may find it easier to open and write the output to a file.

This page intentionally left blank

HOUR 20

FailWhale and the Future of the API

What You'll Learn in This Hour:

- ▶ The History of the FailWhale
- ▶ How to get your application to retry if Twitter is not working
- ▶ What 420 really means
- ▶ The future of the Twitter API

What Is Spotting the FailWhale?

We touched this briefly in Hour 6, but now it's time to formalize how we deal with getting replies back from Twitter that are other than what we expect, and twitter servers being over capacity; sometimes referred to as spotting the FailWhale. First, what is the FailWhale?

The designer behind the FailWhale, Yiying Lu (<http://www.yiyinglu.com/sc/> illustration) had posted the image to the stock photo website, iStockPhoto (that image is now removed). The original title of this artwork was "Lifting a Dreamer." Twitter adapted this image for its 502 and 503 HTTP response pages. In a discussion thread on FriendFeed about Twitter's downtime art, Robert Scoble deemed that it should be called FailWhale, and it stuck. "Fail" was already an existing Internet meme, at the time used for anything that was not successful in one way or another. For a brief time, Twitter tried to get rid of the FailWhale with a generic overcapacity message, and later other graphics, but user backlash caused it to be reinstated. The FailWhale has become so popular that it even has a fan club, failwhale.com (<http://failwhale.com/>). Because Twitter's growth and popularity exceeded anyone's expectations, the FailWhale was seen over and over again, and it became a reference for Twitter being down.

The FailWhale has become so well known that other variations have appeared, including Homer Simpson as the whale, a beer called Fail Whale Pale Ale, cakes, pumpkins, and of course printed on t-shirts, jackets, and coffee mugs.

Now that we know the history of the FailWhale, and why it appears, let's figure out a way to deal with it so we can have our own "Fail" art pages.

Just a reminder, the following are the currently supported HTTP response codes from Twitter:

- ▶ **200 OK**—Success!
- ▶ **304 Not Modified**—There was no new data to return.
- ▶ **400 Bad Request**—The request was invalid. An accompanying error message will explain why. This is the status code that will be returned during rate limiting.
- ▶ **401 Unauthorized**—Authentication credentials were missing or incorrect.
- ▶ **403 Forbidden**—The request is understood, but it has been refused. An accompanying error message will explain why. This code is used when requests are being denied because of update limits.
- ▶ **404 Not Found**—The URI requested is invalid or the resource requested, such as a user, does not exist.
- ▶ **406 Not Acceptable**—Returned by the Search API when an invalid format is specified in the request.
- ▶ **420 Enhance Your Calm**—Returned by the Search and Trends API when you are being rate limited.
- ▶ **500 Internal Server Error**—Something is broken. Please post to the group so the Twitter team can investigate.
- ▶ **502 Bad Gateway**—Twitter is down or being upgraded.
- ▶ **503 Service Unavailable**—The Twitter servers are up, but overloaded with requests. Try again later.

Did You Know?

There is a standard for what HTTP response codes are supposed to indicate, but it's up to the one who set up the server configuration to decide what they will mean. For example, 420 used by Twitter is not a standard response code. There is even a 418 code saying, I'm a teapot: The HTCPCP server is a teapot. The responding entity may be short and stout. Defined by the April Fools specification RFC 2324. See Hypertext Coffee Pot Control Protocol for more information: http://en.wikipedia.org/wiki/Hyper_Text_Coffee_Pot_Control_Protocol.

Review of the Application We Just Built

It's Hour 20, and we have covered a lot of ground in very little time. It's a good time to step back to review what we have built, how we are using the API currently, and how we may want to use it in the future.

Application Architecture

In Hour 3, we talked about the various types of Twitter applications. So, what did we just build over the past 20 hours? Well, believe it or not, we created a functioning Twitter application. Notice I did not say full featured because there are some elements we glossed over in the interest of time.

The first Twitter.com website had fewer features than what we have created here. Only recently has Twitter.com updated its site's features and functions with a major upgrade in the fall of 2010.

Did You Know?

So, let's take a look at a diagram of our application architecture, as shown in Figure 20.1.

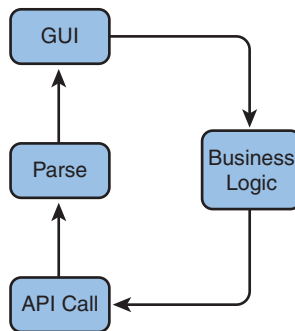
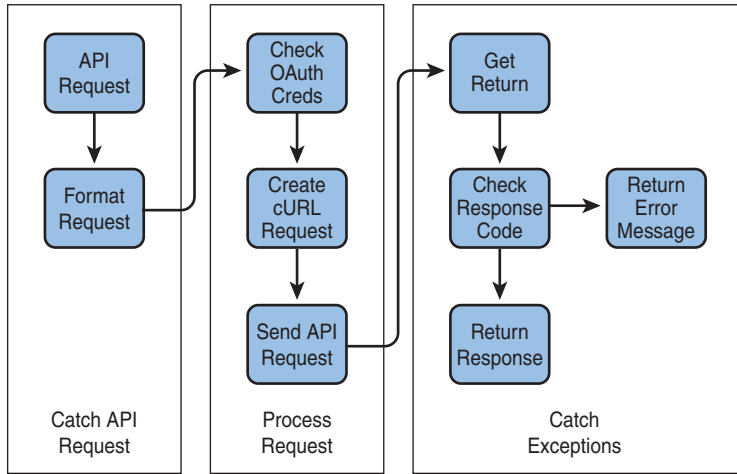


FIGURE 20.1
High-level architectural view of our Twitter application.

We have a straightforward application structure that can support almost any future API methods that Twitter will come up with. Let's take a closer look at the API Call box in our architecture diagram in Figure 20.2.

FIGURE 20.2
Architecture of
our API class.



As we can see from the diagram, we have three main parts of the class: Catch API Request, Process Request, and Catch Exceptions. In Hour 8, we explored Abraham Williams' OAuth class and talked a little bit about how it worked, but now that we have learned about the API, it's a good idea revisit this class and understand it within the larger context of developing a Twitter application.

Catch API Request

Almost all our work on the file `twitteroauth.php` has been around defining function calls to catch requests from our application. A good question to ask is, "Why didn't we just build the request in the application?" The reason is that the Twitter API is ever evolving. Twitter makes a great effort to make sure its changes will not break things, but as Twitter continues its massive growth and evolution of product offerings, we have to be prepared. Another reason to keep the API class outside the application is the ease of replacing the current class with a more updated class library. Although the class library we used in this book serves our purpose in its clarity and simplicity, you may be well served to explore more recent and sophisticated classes to use with your application. So, let's take a look at one of the functions we use to format our API call for catching API calls:

```
function showgeo_search($options){
    $api_call = '/geo/search.json?query='.$options;
    return $this->get($api_call);
}
```

This should be old hat to you by now. We have our function call, the options we want to pass, construction of the `$api_call` variable, and the actual call to the function that begins the Process Request part of our class. This is pretty typical, but do you remember this function call?

```
function getMentions($format, $id = NULL, $count = 60, $since = NULL) {
    if ($id != NULL) {
        $api_call = sprintf("statuses/mentions/%s.%s", $id,
➤$format);
    }
    else {
        $api_call = sprintf("statuses/mentions.%s", $format);
    }

    if($since != NULL){
        $api_call .= sprintf("?since_id=%s", urlencode($since));
        $count=0;
    }

    if ($count != 60 AND $count!='') {
        $api_call .= sprintf("?count=%d", $count);
    }

    return $this->get($api_call);
}
```

Not every API request catch function is a few simple lines. As Twitter starts to get more and more RESTful, we will see more attributes that will be passed to Twitter after the initial REST call. Also, by keeping these calls separate, we can customize our default setting based on our needs. For example, we may write a function knowing the default response from Twitter is 20 statuses. Then, it changes to 60, but our application is not set up to handle that many status updates. By explicitly stating how many returns we expect to get back by default, we can better manage changes in how Twitter provides returns to us.

What to Do When the Twitter Service Is Down

From time to time, the Twitter servers will not be able to respond to your API request. There will also be times when your users have used more than the current limit of Twitter API calls. To deal with this, we need to examine the HTTP response codes we talked about at the top of this hour and respond accordingly.

First, we'll add a few simple lines of code to allow us to respond to the unexpected.

Edit twitteroauth.php

Open the twitteroauth.php file and add some code. Find the following lines:

```
curl_setopt($ci, CURLOPT_URL, $url);
$response = curl_exec($ci);
$this->http_code = curl_getinfo($ci, CURLINFO_HTTP_CODE);
$this->last_api_call = $url;
```

Add the following lines, including the comment text:

```
$errCode = $this->http_code;
/*
200 OK: Success!
304 Not Modified: There was no new data to return.
400 Bad Request: The request was invalid. An accompanying error message
will explain why. This is the status code will be returned during rate
limiting.
401 Not Authorized: Authentication credentials were missing or incorrect.
403 Forbidden: The request is understood, but it has been refused. An
accompanying error message will explain why. This code is used when
requests are being denied due to update limits.
404 Not Found: The URI requested is invalid or the resource requested, such
as a user, does not exist.
406 Not Acceptable: Returned by the Search API when an invalid format is
specified in the request.
420 Enhance Your Calm—Returned by the Search and Trends API when you are
being rate limited.
500 Internal Server Error: Something is broken. Please post to the group so
the Twitter team can investigate.
502 Bad Gateway: Twitter is down or being upgraded.
503 Service Unavailable: The Twitter servers are up, but overloaded with
requests. Try again later. The search and trend methods use this to
indicate when you are being rate limited.
*/
        switch ($errCode) {
            case '400':
                { return $response; }
            break;
            case '401':
                { return $response; }
            break;
            case '403':
                { return '<err>403</err> '; }
            break;
        }
        if(stristr($errCode,'200') ) {} else {
            print '<h4>status '.$response.'</h4><br>';
            print "Calling Twitter again ...<p>";
            $i++; if ($i>2) { print 'Giving up'; return; }
        }
        sleep(2);
        curl_setopt($ci, CURLOPT_CONNECTTIMEOUT , $timeout);
        //CURLOPT_TIMEOUT
        $response = curl_exec($ci);
```

```

        if (empty($response)){
            print "<h4>Sorry, twitter just does not want to talk now. Try
➔again in a few min? <p></h4>";
            break;
        }
    }
}

```

Save your file and close.

The commented code is clear enough. We are keeping a log of the existing possible HTTP response codes that Twitter will return. This is useful for when new error codes come into being, such as the 420 code: Enhance your calm.

420 is a fairly well-known shorthand in the Internet community to represent smoking, planting, or just identifying the use of marijuana. The joke here is that you are trying to hit the Search and Trends servers too often and need to slow down. Who says the people at Twitter do not have a sense of humor?

Did You Know?

The next lines are pretty straightforward. We are using a case switch for some of the 40x conditions. We also set up the switch to make one of the exercises at the end of the hour easier for some beginning programmers.

The next line, however, is more interesting:

```
if(stristr($errCode,'200') ) {} else {
```

Here we are checking to see if we have any response code other than '200'.

Remember that we already have accounted for 400–403, so all other current and future codes will be handled here.

We want to catch the unknown exceptions here because often, and more often than Twitter would like, the service will be come unavailable. However, waiting one or two seconds can be enough that the service will come back on again. So, we want to put our script to sleep for two seconds and then try the API call again. You can see how we are doing by using \$i as a counter to keep track of how many times we retry hitting the Twitter servers, as you can see in the following lines:

```

        $i++; if ($i>2) { print 'Giving up'; return; }
sleep(2);
curl_setopt($ci, CURLOPT_CONNECTTIMEOUT , $timeout); //CURLOPT_TIMEOUT
$twitter_data = curl_exec($ci);

```

Notice that if \$i is greater than 2, we printed 'Giving up' and then return the function call. You could set this higher and keep hitting Twitter over and over, but this is frowned upon and considered bad practice. In addition, if Twitter finds that you are

constantly hitting the servers and the same non-200 response code is given, you may find yourself flagged and have your privileges revoked.

We also have a check for when 'nothing' comes back:

```
if (empty($twitter_data)){
    print "<h4>Sorry, twitter just does not want to talk now.. Try again in a
few min? <p></h4>";
    break;
}
```

Although here we print a simple message saying that Twitter does not want to talk right now, we could also put our own version of a FailWhale at this point.

Where Is the Twitter API Going?

Those of us who have been using the Twitter API since the beginning have seen some interesting changes in the approach and style of how the API is designed and implemented. Although Twitter tries not to break past applications with API changes, it has happened and most likely will happen again. Greater and greater load requirements exist for Twitter to make changes to support the growing user base and expand Twitter's functionality.

First, if you were to ask 10 people at Twitter where the API is going, you will get 11 responses back, most likely because at least one person will change his or her mind. That is not unexpected because right now the major focus of Twitter is to keep the system stable. Changes in the API, however, can give us clues about where it will be in the future.

Have a look at the List API methods. Remember from Hour 13 that we had an approach to the API we had not seen before. With the List API, we use the same URL method call for `:user/:lists/:id`, but to accomplish three different tasks. This is because we would use different cURL protocols; for example, GET: Shows the specified list; POST: Updates the specified list; and DELETE: Deletes the specified list.

Although there is no official position on this, I would expect that we will see future API calls working this way.

Remember Search? The search and trend protocols do not work like the rest of the Twitter API family because of search having been developed by an outside company. There have been intentions to revise the search API to work more like the rest of the Twitter API, but because of the amount of work required to keep Twitter running and to update features and functions, it is simply not worth the time and effort to revise something that already works. Will we see major changes in the search or

trends APIs in the future? Not likely. Instead, Twitter has invested significant time and effort in its streaming functionality, which has taken a good deal of pressure off the search API methods.

Attributes

Twitter has been working to expand the functionality of the API set to allow users to become even more creative. One idea was to allow users to define their own attributes that would be attached to each status. It was a great idea but had to be put on the shelf because of some technical difficulties and yet another massive traffic growth spike. Although on the sideline for now, I fully expect user-defined attributes at some point in the future. A user-defined attribute could be quite interesting and allow tweets to become more self-defining. Imagine a tweet having meta information about the tweet. Perhaps the tweet is a headline and the attribute contains more information like the source of the story, or type of tweet (that is, funny, business, stock, and so on). How would this work with the API? Most likely the same way we have seen all the other API calls: perhaps `/userDefined?`

Widgets and a New Web Interface

In 2010, two big changes happened with Twitter, but not around the API. Twitter introduced widgets. We discussed them briefly in Hour 18. We can expect a few more widgets to be released by Twitter, and perhaps an API that is dedicated to interacting with special functions on these widgets. We also saw a new web interface in mid-2010 that did not expose more API calls; however, as Twitter starts to put more effort into its web-based tools, as well as its recently released iPhone and iPad applications, perhaps we can look forward to new features and functionality that can be exposed via the API.

Speaking of iPhones, it's clear that mobile devices and thin tablets will become more and more prevalent. There are features and functions that lend themselves to mobile and tablet devices that are not needed or well supported with a web-based client. As Twitter matures its current offering in this sphere, new features and functions will come online and, hopefully, the API will expand to support information generated or unique to its display.

Summary

In this hour, we learned that Twitter shows a graphic when something is wrong on the server side, normally overcapacity. This image was coined the FailWhale and gained a significant level of popularity as a standalone meme.

We learned how to catch HTTP response codes that are not 200, meaning everything seems fine, and perform different actions based on the response code. We also learned how to pause and resend our request to Twitter as a workaround in case the Twitter servers are not able to fulfill our API requests.

Q&A

- Q.** *If the Twitter servers are not responding, and given the massive growth Twitter has experienced, what would be the most likely HTTP response code you would see when Twitter is overloaded?*
- A.** 502 or 503.
- Q.** *What is the advantage of keeping our API library separated from our application code?*
- A.** The advantage is making changes, upgrade, or complete library replacement easier. Given the rapid changes at Twitter, this is highly recommended.
- Q.** *What is the future of Twitter?*
- A.** Trick question. No one knows the future of Twitter, not even the people at Twitter.

Workshop

Quiz

1. Of the HTTP response codes we have explored in this hour, which one is not standard HTTP response code?
2. Why is it not a good idea to build API calls directly in the application?
3. What is the HTTP response code 304, and what type of error page should be created for it?

Quiz Answers

1. 420 Enhance Your Calm. It is returned by the Search and Trends API when you are being rate limited.

2. It's not a good idea because the API is always evolving and expanding. Also, new libraries are created and old ones are updated often, so it's to your advantage to make updating and changing the API library as easy as possible.
3. Not Modified—There was no new data to return. Unlike other HTTP Response codes, 304 as well as 200 are not errors; thus, an error page would not be appropriate for this response.

Exercise

Build your own custom pages for each error message. Try to come up with an image for each.

This page intentionally left blank

HOUR 21

Getting Started in Twitter Android Application

What You'll Learn in This Hour:

- ▶ Creating an AVD
- ▶ Using the Android Development Tools (ADT) plug-in for Eclipse
- ▶ Building your first Hello Android application
- ▶ Using Eclipse and Android Emulator
- ▶ Getting Twitter Java library and OAuth Java library

Introducing Android

We have spent the last 20 hours learning about the Twitter API and how to employ it using the LAMP stack. Now we are going spend the last 4 hours learning how to set up a Twitter library for mobile devices, in our case, Android and iOS4. Unless you decided to skip ahead directly to this hour, you should already be comfortable with the various API calls and how to employ them, so we will instead focus on simply getting you up and running on your mobile platform with OAuth.

Before we get started, a few words of warning. As we have stated before, this book is not intended for beginners, but those that have some experienced with programming. However, we did try to accommodate beginning programmers who are willing to look up terms and concepts that they do not understand. In the following 4 hours, we took the same approach; however, these hours are very dense with information and move quite quickly. Although a beginning programmer could successfully navigate the mobile section with the step by step technique used throughout this book, you may find yourself getting lost in the flurry of terminology used. Do not despair, you will get through it, just have a little faith and do LOTS of web searches for terms you do not understand. Also, be sure to check this book's website (<http://www.twitterapi24.com/>) for any changes or update since this book has gone to print. Doing so may save you some headaches down the road.

First, we are going to start with the Android SDK and Eclipse IDE, by creating a Hello Android application. A simple Hello Android application contains the basic Android application functionality and prints out the text “Hello Android” on the device screen.

You can download the Android SDK at <http://developer.android.com/sdk/>. Follow the Official Android documentation to install Android SDK at <http://developer.android.com/sdk/installing.html>. You can download the Eclipse IDE at <http://www.eclipse.org/downloads/>. The preferred Eclipse version for mobile development is Eclipse Classic.

It's important that you set up your Android Development Environment properly before we start working on Twitter API: Android SDK (Software Development Kit), AVD (Android Virtual Devices) Manager, Eclipse IDE (Integrated Development Environment), and ADT (Android Development Tools) plug-in.

Android ADT Plug-In

The ADT (Android Development Tools) plug-in provides the tools to develop, compile, package, and deploy Android applications:

- ▶ The Android Project Wizard
- ▶ The Android SDK (Software Development Kit) and AVD (Android Virtual Devices) Manager
- ▶ The Eclipse DDMS (Dalvik Debug Monitor Service) for monitoring and debugging Android applications
- ▶ The Android LogCat logging
- ▶ Automated builds and application deployment to Android emulators and devices
- ▶ Packing and code-signing tools for application deployment

Creating an AVD

Before you can launch the Android Emulator, you have to create an Android Virtual Device (AVD). This AVD defines the system image and device settings used by the emulator.

To create an AVD, follow these steps:

1. In Eclipse, select Window, Android SDK and AVD Manager.
2. In the left panel, select Virtual devices.
3. Click New.

4. Enter the name of AVD.
5. Select the target of platform (version of Android SDK) for the Android emulator.
6. Click Create AVD.

Creating the Hello Android Project

The Android Project Wizard creates all the required files for an Android application. Before you can create an Android project from Eclipse, ADT plug-in for Eclipse must be installed properly. Follow these steps to create a new project:

1. In Eclipse, select File, New, Project.
2. Select the Android folder and choose Android Project, as shown in Figure 21.1. Click Next.

In the Build Target, you can select Android X.X or Google APIs. If you decide to use Google Map in your Android Application, select Google APIs. If you would like to use Google Android Market License APIs, select Google APIs.

Did You Know?

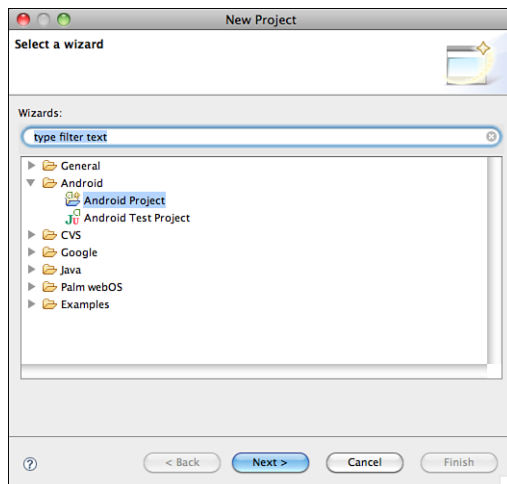
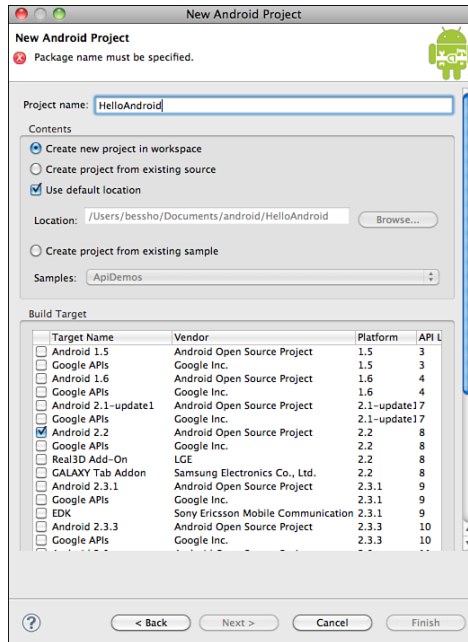


FIGURE 21.1
Selecting
Android Project
in the Android
Project Wizard.

3. Enter HelloAndroid in the project name.
4. Select Create new project in workspace.
5. Select the Use default location check box.

6. In the Build Target section, select Android 2.2 on Platform 2.1 or Google APIs on Platform 2.1, as shown in Figure 21.2.

FIGURE 21.2
Creating a New Android Project in the Android Project Wizard.



7. In the Properties section, type Hello Android in the Application name. This application name will appear on the application screen of Android Emulator or Device.
8. Type com.example.helloandroid in the Package name. This is the package namespace where all your source code will reside. The package name must be unique across all packages installed on the Android. It is suggested to use a standard domain-style package for the application such as “com.example.com” namespace.
9. Select Create Activity check box. Type helloandroid in the Create Activity. This is for your main Activity class. It’s an optional field.
10. Enter a Min SDK Version. Type 8. This is an integer to indicate the minimum API Level required to run your application. This value will automatically set the minSdkVersion attribute in the <uses-sdk> of your Android Manifest file.
11. Click Finish.

If you don't see the Android folder in the Android Project Wizard, you may not have a properly installed ADT plug-in.

**Watch
Out!**

Exploring the helloandroid.java File

By now, your Android project should be created. It should be visible in the Package Explorer in the left panel. You can find the helloandroid.java file at helloandroid, src, com.example.helloandroid.

This helloandroid.java file contains the following source code:

```
package com.example.helloandroid;

import android.app.Activity;
import android.os.Bundle;

public class helloandroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

The class is based on the Activity class. The onCreate() method will be called by the Android system after the activity starts. This is where you initialize and set up the UI.

To add import packages to your project, you can use Ctrl+Shift+O (PC) or Cmd-Shift-O (Mac). It's an Eclipse short key to identify missing packages based on your code and add them.

**Did You
Know?**

Eclipse detects the missing packages and added android.R package to the code. Eclipse adds the import statement of android.R package to the helloandroid.java file:

```
package com.example.helloandroid;

import android.R;
import android.app.Activity;
import android.os.Bundle;

public class helloandroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```


**Watch
Out!**

If you run the HelloAndroid application now, you will encounter an error: `R.layout.main` cannot be resolved in `helloandroid.java` line 12. You can view the error description in the Problem window.

Creating the Hello Android Application

After creating the HelloAndroid project, it's time to add your own codes.

The Android UI is composed of hierarchies of objects named Views. A view is a drawable object in the UI layout element, such as a button, image, or text label. The subclass that handles text is `TextView`.

You create a `TextView` with the class constructor that takes Android Context instance as its parameter. Define the text content with the `setText()` method. Display the content by passing `TextView` to the `setContentView()` method for the Activity UI.

In `helloandroid.java`, you will create a `TextView`, define the text in the `setText()` method, and display the content in the `setContentView()` method:

```
package com.example.helloandroid;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class helloandroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView viewHello = new TextView(this);
        viewHello.setText("Hello Android");
        setContentView(viewHello);
    }
}
```

Eclipse detects the missing packages and adds `android.widget.TextView`. We can now remove the unnecessary `android.R` package:

1. In Eclipse, select the HelloAndroid project on the left panel. Select Run, Run As, Android Application.
2. Select Android Application and click OK.
3. In your first time, the Android AVD Error window will appear and show "Do you wish to add a new Android Virtual Device?" Click Yes.

4. In the Android Device Choose window, select Launch a new Android Virtual Device.
5. Select New.
6. In the Create new Android Virtual Device (AVD), enter Android.
7. In the Target drop-down list, select Android 2.2 – API Level 8.
8. In the SD Card section, select Size and MiB and enter 20.
9. Leave Skin as Built-in Default (WVGA800).
10. Click Create AVD.
11. Select Android from the list of existing Android Virtual Devices.
12. Click Start.
13. Android emulator starts and launches the Android operating system, as shown in Figure 21.3.
14. Click the Menu button to unlock the home screen.

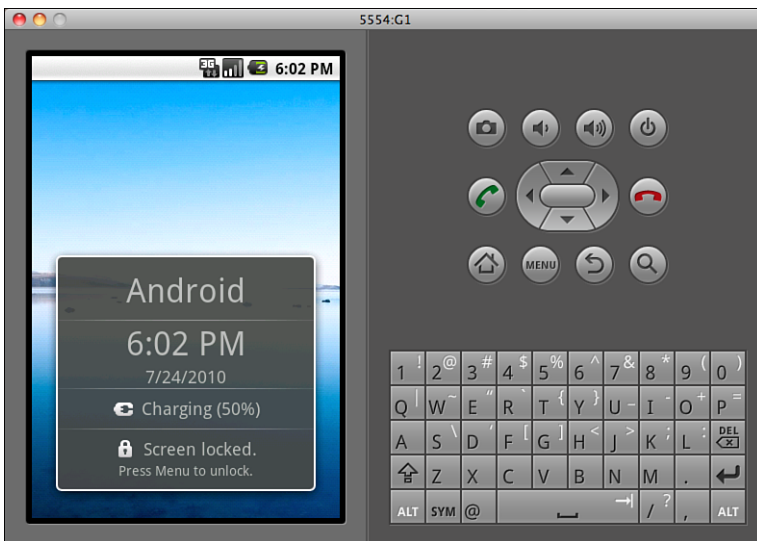
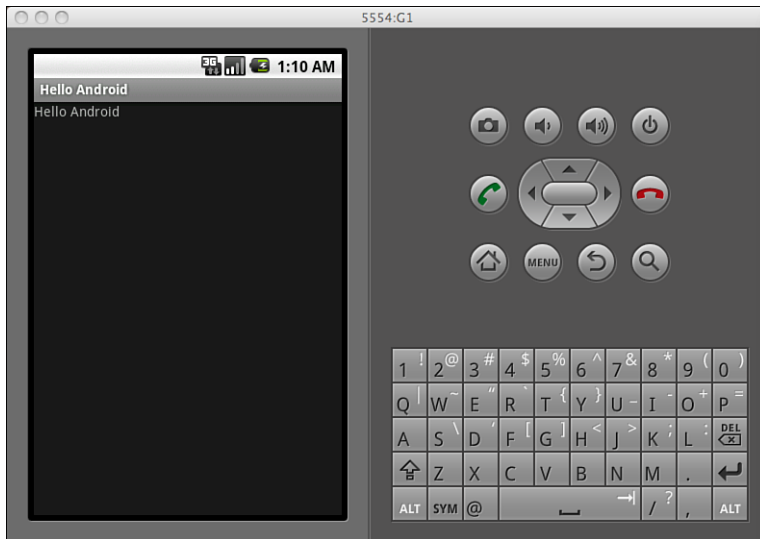


FIGURE 21.3
Unlocking the screen on the Android home screen.

15. After unlocking the home screen, the text string “Hello Android” inside the TextView appears on the screen, as shown in Figure 21.4.

FIGURE 21.4
 Launching
 HelloAndroid
 application.



The Eclipse plug-in generates the Title Bar in gray below the Status Bar. The Title Bar shows the string “Hello Android”. The Eclipse plug-in creates the Title Bar automatically based on the string defined in the `res/values/strings.xml` file and `AndroidManifest.xml` file. The text below the Title Bar is the text string you have created in the `TextView` object.

The purpose of running the Hello Android application is to verify that your Eclipse, Android SDK, and Android ADT are properly installed. If you cannot run the Hello Android application, be sure you use the Official Android documentation as a guide and repeat the entire installation.

Watch Out!

You must enter at least 9 MiB in size in your SD Card section.

If you have created an AVD, you can select an existing AVD in the Android Device Chooser window. Click OK. Be sure your selected AVD Target API Level matches your `android:minSdkVersion` in your `AndroidManifest.xml` file.

Android automatically generates this `manifest.xml` file for new Android project. To check your minimal SDK version of your Android project, open the `AndroidManifest.xml` file. Check the value in the `<uses-sdk>` tag.

1. In the Package Explorer, select and double-click to open the `AndroidManifest.xml` file in your HelloAndroid project folder.

2. Select the AndroidManifest.xml tab below the window to view the file in XML format.

In HelloAndroid Manifest:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.helloandroid"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-sdk android:minSdkVersion="8" />

    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".HelloAndroid"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>
</manifest>
```

In the AndroidManifest.xml `<uses-sdk>` tag, always check if your `android:minSdkVersion` matches your selected AVD Target API Level. If it doesn't, change the `android:minSdkVersion`.

This list specifies the API Level supported by each version of the Android platform:

- ▶ Android 3.0—API Level: 11
- ▶ Android 2.3.3—API Level: 10
- ▶ Android 2.3.1—API Level: 9
- ▶ Android 2.2—API Level: 8
- ▶ Android 2.1—API Level: 7
- ▶ Android 2.0.1—API Level: 6
- ▶ Android 2.0—API Level: 5
- ▶ Android 1.6—API Level: 4
- ▶ Android 1.5—API Level: 3
- ▶ Android 1.1—API Level: 2
- ▶ Android 1.0—API Level: 1

You can always find more updates and information about API Levels at the Android developer site: <http://developer.android.com/guide/appendix/api-levels.html>.

To understand the landscape of device distribution and prioritize the development of your Android application features, let's reveal the current distribution published by Android on May 2, 2011:

- ▶ Android 1.5 API Level 3—Distribution: 2.3%
- ▶ Android 1.6 API Level 4—Distribution: 3.0%
- ▶ Android 2.1 API Level 7—Distribution: 24.5%
- ▶ Android 2.2 API Level 8—Distribution: 65.9%
- ▶ Android 2.3 API Level 9—Distribution: 1.0%
- ▶ Android 2.3.3 API Level 10—Distribution: 3.0%
- ▶ Android 3.0 API Level 11—Distribution: 0.3%

You can always find up-to-date data at the Android developer site: <http://developer.android.com/resources/dashboard/platform-versions.html>.

Resolving Any AVD-Related Issues

Sometimes you may not set up everything properly to match API levels in your application. You need to start fresh. To do so, follow these steps:

1. Select the gen folder of your Android project in the Package Explorer, and right-click to select Delete.
2. Click OK.
3. Disable Project Build Automatically by selecting Project, Build Automatically.
4. Clean the project by selecting Project, Clean.
5. Select Project, Build All.
6. Debug by checking the Console window for any errors.

Resolving Any Android SDK Issues

Sometimes you will run into problems in Android ADT relating to upgrading your Android SDK or Android ADT or Eclipse. Sometimes the latest Android SDK will require a newer version of Eclipse. You would have to download and install the

newer Eclipse version and then install Android ADT. To resolve any Android ADT issues, follow these steps:

1. Start a terminal window.
2. Locate your directory where you save the Android SDK. You can find the path of `/android-sdk-mac_x86/platform-tools/android`.
3. At the terminal prompt, type `ls -a` to view the file list.
4. Type `rm -rf .android` to remove the entire directory.

Downloading the Twitter4J Library

The Twitter developer website suggests a list of Twitter Libraries at <http://dev.twitter.com/pages/libraries>. Among the Java libraries, Twitter4J is widely accepted by Android developers.

Twitter4J is a Java library for the Twitter API. You can download the latest version of Twitter4J file `twitter4j-core-<version>.jar` at <http://twitter4j.org/en/index.html#download>.

A list of Twitter APIs supported by Twitter4J is documented at <http://twitter4j.org/en/api-support.html>.

Twitter4J is thread-safe, and you can make method calls concurrently.

Did You Know?

Downloading the OAuth Library for Java

OAuth-signpost is a simple OAuth message signing for Java. Signpost is widely accepted by Android developers. The latest version can be downloaded at <http://code.google.com/p/oauth-signpost/downloads/list>. Download both `signpost-core-<version>.jar` and `signpost-commonshttp4-<version>.jar`. Documentation of Signpost is available at <http://kaeppler.github.com/signpost/index.html>.

Summary

Congratulations! You are now an Android developer. You have downloaded tools, created a development environment and created your first Android project, however this is just the start. You still have to attach an OAuth library to allow messages to be sent back and forth to the Twitter API servers. We will cover this in the next hour.. Finally, you have run your newly created Android application on the Android Emulator. You also have downloaded Twitter libraries for Android.

Q&A

- Q. *What kind of a development environment do I need for creating an Android application?***
- A.** You need the JDK, Android SDK, Eclipse IDE, Android Developer Tools (ADT) plug-in, and SDK components.
- Q. *What operating systems are supported for Android application development?***
- A.** It is supported by Microsoft Windows XP, Vista and Windows 7, Mac OS X, and Linux.
- Q. *What features does the Android ADT plug-in provide?***
- A.** The Android ADT plug-in adds capacity to Eclipse to create new projects, create UIs, add components based on the Android Framework API, manage installed APIs, manage AVDs, debug using Android SDK tools, and export signed or unsigned APKs to distribute application.

Workshop

Quiz

1. Which programming languages do I use to code and develop an Android application?
 - A. Objective-C
 - B. Java
 - C. HTML / JavaScript / CSS
 - D. C++
2. Which file do you use to create the layout of the Android application?
 - A. helloandroid.java
 - B. main.xml
 - C. AndroidManifest.xml

3. Which element attribute in the XML<uses-sdk> tag of the AndroidManifest.xml file specifies the minimum API Level that the application is able to run?
- A. android:minSdkVersion
 - B. android:targetSdkVersion
 - C. android:maxSdkVersion

Quiz Answers

- 1. B. Use Java to write the code for the Android application.
- 2. B. Use main.xml to create the layout.
- 3. A. Use android:minSdkVersion in the <uses-sdk> tag.

Exercises

- 1. Install JDK, Android SDK, Eclipse, ADT plug-in, and SDK components. Create a HelloAndroid application.
- 2. Build Project and run the application in the Android Emulator.

This page intentionally left blank

HOUR 22

Building Android Applications with Twitter

What You'll Learn in This Hour:

- ▶ Creating the Android OAuth application
- ▶ Adding Java library
- ▶ Creating a layout in main.xml
- ▶ Adding Intent Filters and Permission
- ▶ Using the Java OAuth library
- ▶ Posting a tweet from an Android application
- ▶ Using xAuth in an Android application

Using Twitter OAuth in Android

At the launch of the Twitter API platform, Twitter introduced Basic Authentication. Twitter allows developers to capture and save the username and password in the native app and use them to make all API calls directly. In August 2010, Twitter banned the use of Basic Auth due to security concerns. Twitter now supports both OAuth and xAuth.

Twitter accepts the use of OAuth and xAuth to authenticate the application in a device. OAuth (Open Authorization) is an open standard for authorization that was initially designed for web authentication.

After the Android application initiates a request token, it sends the user to the Twitter website to authorize the request token in exchange for returning the access token to the user.

OAuth-Signpost is a simple HTTP message signing on the Java platform in conformance with the OAuth Core 1.0a standard. OAuth-Signpost has been widely used in Android.

Selecting Twitter Java Libraries

The most popular Twitter Java libraries, Twitter4J and JTwitter, support Android implementation. OAuth Java library OAuth-Signpost works well with these Twitter Java libraries.

Creating an Android OAuth Application

It's time to start a new Android project and add the external libraries. Be sure you follow these steps:

1. In the Package Explorer panel, right-click the project and select New, Folder.
2. Enter lib in the folder name field and click Finish.
3. Open the folder where you keep the downloaded libraries.
4. Select the signpost-commonshttp4-<version>.jar, signpost-core-<version>.jar, and twitter4j-core-<version>.jar files.
5. Drag these files from the folder to the lib folder inside the Package Explorer panel.
6. In the Package Explorer panel, right-click the project and select Properties.
7. Select Java Build Path and select the Libraries tab.
8. Click the Add JARs button.
9. Expand the Android project until you see the lib folder. Select all .jar files and click OK.
10. You should see the added .jar files, as shown in Figure 22.1.
11. Click OK.

After adding the external libraries, you will find that a new folder, Referenced Libraries, is created by Eclipse.

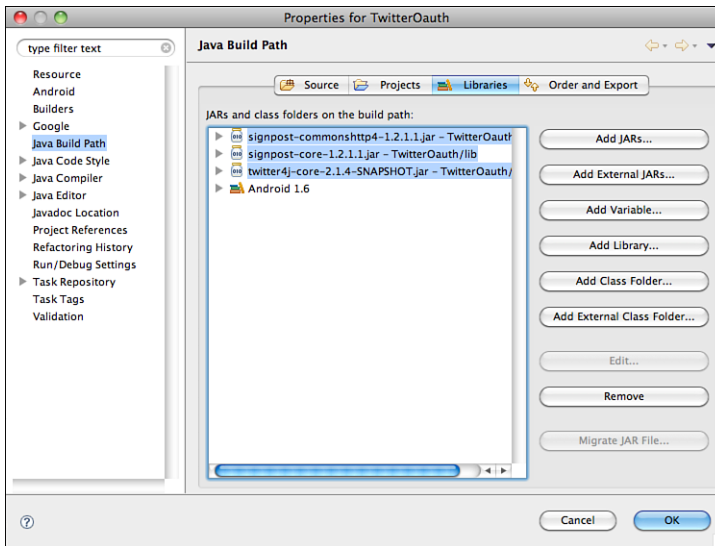


FIGURE 22.1
Adding external libraries to the Android project.

Creating the Layout

The layout is the architecture for the user interface in an Activity. The layout also defines the layout structure and holds all the visible UI elements. You can declare UI elements in XML or instantiate layout elements at runtime. You can create and manipulate the View and ViewGroup objects and its properties programmatically.

In this Android application, we declare the UI in XML. The advantage is that it separates the presentation layer from your behavior, similar to the MVC model. It allows you to create XML layouts for different screen orientations, different device screen sizes, and different languages. It is much easier to debug layout issues to visualize the UI structure in XML.

Each layout file must contain only one root element of a View or ViewGroup object. In this XML layout, we use a vertical LinearLayout to hold a TextView and a Button. The layout file is located in `res/layout/main.xml`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
> </LinearLayout>
```

In this app layout, we first create a text label “Twitter OAuth” by using a `TextView` tag with an id “TextView”:

```
<TextView android:text="Twitter OAuth" android:id="@+id/TextView" an-
droid:layout_width="wrap_content" android:layout_height="wrap_content" an-
droid:layout_marginBottom="20dp">
</TextView>
```

ID is associated with View object. It is used to uniquely identify the View within the tree. When the Android application is compiled, the ID is referenced as an integer, but the ID is assigned in the layout XML file as a string in the XML id attribute. The at symbol (@) at the beginning of the string states that the XML parser should parse and expand the entire ID string and identify it as an ID resource. The plus-symbol (+) refers to a new resource name that must be created and added to the R.java file:

```
android:id="@+id/TextView"
```

`TextView` contains XML attributes of `android:layout_width` and `android:layout_height`. `android:layout_width` specifies the basic width of the `TextView`. It is a required attribute for any view inside of a containing layout manager. Available units are px (pixels), dp (density-independent pixels), sp (scaled pixels based on preferred font size), in (inches), and mm (millimeters). `wrap_content` defines to size itself to the dimension required by its content (with padding). Another option is `fill_parent` (renamed `match_parent` in API Level 8), which resizes the view as big as its parent view group (without padding). Similar rules apply to `android:layout_height`.

`TextView` contains XML attributes of `android:layout_marginBottom`. `android:layout_marginBottom` specifies the extra space on the bottom side of the view, and this space is outside the view’s bounds. Other options are `android:layout_marginLeft`, `android:layout_marginRight`, and `android:layout_marginTop`.

Next, we create a button with the text label “Authenticate” by using a `<Button>` tag with an id “ButtonLogin”:

```
<Button android:text="Authenticate" android:id="@+id/ButtonLogin" an-
droid:layout_width="wrap_content" android:layout_height="wrap_content" an-
droid:layout_marginBottom="50dp">
</Button>
```

Last, we create an empty text label by using a `<TextView>` tag with an id “tweet”:

```
<TextView android:text="" android:id="@+id/tweet" an-
droid:layout_width="wrap_content" android:layout_height="wrap_content">
</TextView>
```

The order of the codes in the layout main.xml is shown here:

Both Data Scheme and Data Authority matching are case sensitive. It is recommended to use all lowercase letters.

Did You Know?

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
><TextView android:text="Twitter OAuth" android:id="@+id/TextView" an-
➤droid:layout_width="wrap_content" android:layout_height="wrap_content"
➤android:layout_marginBottom="20dp">
</TextView>
<Button android:text="Authenticate" android:id="@+id/ButtonLogin" an-
➤droid:layout_width="wrap_content" android:layout_height="wrap_content"
➤android:layout_marginBottom="50dp">
</Button>
<TextView android:text="" android:id="@+id/tweet" an-
➤droid:layout_width="wrap_content" android:layout_height="wrap_content">
</TextView>
</LinearLayout>
```

Adding Intent Filters and Permission

Intent is an abstract description of a performed operation. It is heavily used in launching activities and is sometimes described as the glue between activities.

IntentFilter objects are XML `<intent-filter>` tags in the `AndroidManifest.xml` file. You can filter Intent into three areas: action, data, and categories.

Action matches if any values match the Intent action or if no actions were specified in the filter. Categories match if all categories in the Intent match categories given in the filter. Data Type matches if any values match the Intent type. Data Type is classified into Data Scheme, Data Authority, and Data Path.

It is not recommended to use absolute units such as pixels in specifying a layout width and height. It is highly recommended to use relative measurements, such as density-independent pixel units in `dp`, `wrap_content`, or `fill_parent`.

Did You Know?

`android.intent.action.VIEW` is Activity Action that displays the data to the user. `android.intent.category.DEFAULT` is set if the activity should be an option for the default action to perform on data. `android.intent.category.BROWSABLE` is activities that invoke from a browser.

android:scheme is scheme part of a URI and used as a filter. android: host is the host part of a URI authority. This android: host attribute is meaningless unless android:scheme scheme attribute is also specified in the filter.

android.permission.INTERNET allows applications to open network sockets. This gives permission to access the Internet.

An Android application uses the android:minSdkVersion attribute to indicate the lowest system API Level it supported. System checks the value of android:minSdkVersion and allows the install only if the referenced integer is less than or equal to the API Level integer stored in the system. The Minimal SDK version level is set at 4 in the <uses-sdk> tag.

In this AndroidManifest.xml, it shows intent filter and permission required to integrate Twitter. The <uses-permission> tag is used to give the app permission on internet. In the <activity> tag, the single instance is added in the android:launchMode. This is to restrict the main activity to one stance:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.twitteroauth"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        ↪android:label="@string/app_name">
        <activity android:name=".TwitterOauth"
            android:label="@string/app_name"
            android:launchMode="singleInstance">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />
                <data android:scheme="myapp" android:host="twitteroauth" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="4" />
    <uses-permission android:name="android.permission.INTERNET"></uses-
    permission>
</manifest>
```

Loading the XML Resource

The Android application main Java file is located in `src/com.example.twitteroauth/TwitterOauth.java`. When the Android application is compiled, each XML layout file is compiled into a View resource. Calling `setContentView()` method will load the referenced layout resource in the form of `R.layout.<layout_filename>`, such as `R.layout.main`. `onCreate()` method in the Activity is called by the Android framework when your Activity is launched. Keep in mind that the activity will be destroyed and re-created in an orientation change or if a physical keyboard is opened (even if the orientation is locked).

This is the default `TwitterOauth.java` file that is created by Android project. This is your starting point where we would include packages and add codes to support Twitter authentication and posting tweet. Let's put this class in place:

```
package com.example.twitteroauth;

import android.app.Activity;
import android.os.Bundle;

public class TwitterOauth extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Importing Packages

Java classes are groups in packages. A package is the same as the directory name that contains the `.java` file. You declare the packages when you define the Android application. Name the packages from other libraries in an import statement. Activity class is in the `app` package, which is located in the Android package. `Android.app.Activity` is a standard screen with no specialization.

It's time to add the additional packages from other libraries we need from Twitter4J, Signpost, and additional Android packages.

In `TwitterOauth.java`:

```
package com.example.twitteroauth;

import java.sql.Date;
```



```

import oauth.signpost.OAuthConsumer;
import oauth.signpost.OAuthProvider;
import oauth.signpost.commonshttp.CommonsHttpOAuthConsumer;
import oauth.signpost.commonshttp.CommonsHttpOAuthProvider;
import twitter4j.Twitter;
import twitter4j.TwitterFactory;
import twitter4j.auth.AccessToken;
import twitter4j.conf.ConfigurationBuilder;
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

public class TwitterOAuth extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}

```

Adding OAuth

First, we declare APP as string in the private static final statement. Private indicates that it is visible only to the objects of the same class. Static gives only one instance of this member. Final allows it to be assigned once.

Next, we declare twitter as Twitter, provider as OAuthProvider, and consumer as CommonsHttpOAuthConsumer:

```

private Twitter twitter;
private OAuthProvider provider;
private OAuthConsumer consumer;

```

We also declare the CONSUMER_KEY in string, CONSUMER_SECRET in string, and CALLBACK_URL in string. This is where you insert your Twitter Application consumer key, consumer secret, and callback url from the application settings from your Twitter app:

```

private String CONSUMER_KEY = "<Consumer Key>";
private String CONSUMER_SECRET = "<Consumer Secret>";
private String CALLBACK_URL = "http://www.twitter.com";

```

Now we declare variables `tweetTextView` as `TextView`, and `buttonLogin` as `Button` used in the layout. In the app, we are adding a label and a button:

```
private TextView tweetTextView;
private Button buttonLogin;
```

`onCreate(Bundle)` is where you initialize the activity. Call `setContentView(int)` to define the UI from the layout resource. Call `findViewById(int)` to retrieve the UI widget.

This `onClick(View v)` method is called when a view has been clicked:

```
public void onClick(View v){}
```

In `TwitterOauth.java`:

```
package com.example.twitteroauth;

import java.sql.Date;

import oauth.signpost.OAuthConsumer;
import oauth.signpost.OAuthProvider;
import oauth.signpost.commonshttp.CommonsHttpOAuthConsumer;
import oauth.signpost.commonshttp.CommonsHttpOAuthProvider;
import twitter4j.Twitter;
import twitter4j.TwitterFactory;
import twitter4j.auth.AccessToken;
import twitter4j.conf.ConfigurationBuilder;
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

/**
 * Using Twitter4J Java library and Signpost OAuth library to access
 * Twitter
 */
public class TwitterOauth extends Activity {

    private static final String APP = "TWITTEROAUTH";

    private Twitter twitter;
    private OAuthProvider provider;
    private OAuthConsumer consumer;

    private String CONSUMER_KEY = "v7xwr2xFzDEsgDvg962Paw";
```

```

        private String CONSUMER_SECRET =
        ↪ "Y1bknWgtA9t5u4q6q18IwHTQ2vt0lHesVB09u5EtEy4";
        private String CALLBACK_URL = "myapp://twitteroauth";

        private TextView tweetTextView;
        private Button buttonLogin;

        @Override
        public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
            tweetTextView = (TextView) findViewById(R.id.tweet);
            buttonLogin = (Button) findViewById(R.id.ButtonLogin);
            buttonLogin.setOnClickListener(new OnClickListener() {
                public void onClick(View v) {
                    askOAuth();
                }
            });
        }
    }

```

Authenticating the Application

To focus on illustrating the Twitter authentication process, we will not go into details in explaining Java programming line by line. Instead, we will highlight a few things that are unique or key to understand for implementing this framework. We also include the section code so you can see the logic and workflow in the codes.

To authenticate the Android application, we add a private method `askOAuth()` to verify against the Twitter application consumer key and secret:

```
askOAuth();
```

This is a key section where we declare the objects for the OAuth HTTP messaging for the consumer and provider. To make it more secure, we are using “https” version of Twitter urls for request token, access token, and authorize. We define a string object `authURL` after authentication is completed:

```

// Use Apache HttpClient for HTTP messaging
consumer = new CommonsHttpOAuthConsumer(CONSUMER_KEY, CONSUMER_SECRET);
provider = new CommonsHttpOAuthProvider
("https://api.twitter.com/oauth/request_token",
 "https://api.twitter.com/oauth/access_token",
 "https://api.twitter.com/oauth/authorize");
String authUrl = provider.retrieveRequestToken(consumer, CALLBACK_URL);

```

Here we introduce the `android.widget` class `Toast`. A toast is a view containing a quick message to the user. This class creates and shows this message. It appears as a floating view, and it is unobtrusive to the user, as shown in Figure 22.2.

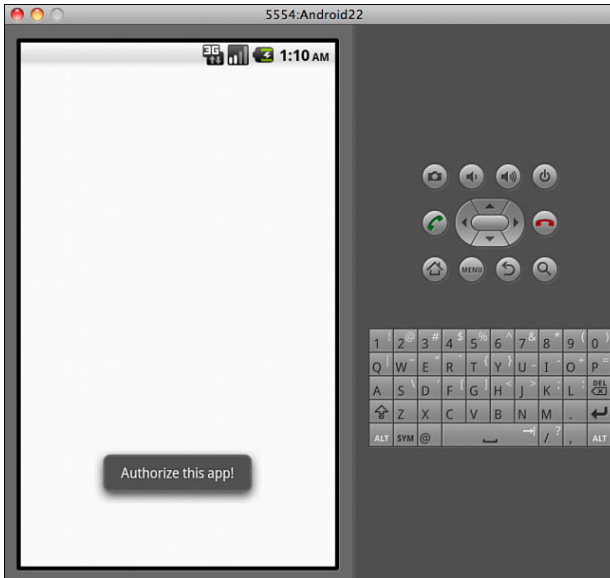


FIGURE 22.2
Showing the
Toast class on
the app.

`Toast.makeText()` is to make a standard toast that contains a text view. We want to show the message “Authorize this app!” as text:

```
Toast.makeText(this, "Authorize this app!", Toast.LENGTH_LONG).show();
```

`Toast.LENGTH_LONG` is a constant, showing the text notification for a long period of time. Another option is `LENGTH_SHORT`. `Toast.show()` is to show the view for the specified duration.

```
this.startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse(authUrl)));
```

The `startActivity (Intent intent)` method is to launch a new activity. The intent is `ACTION_VIEW` displaying the data to the user. `authUrl` is a URI string. The `Uri.parse()` method is to create a `Uri` that parses the given encoded URI string of `authUrl` variable:

```
/**
 * Direct to Twitter to authenticate the user
 */
private void askOAuth() {
    try {
        // Use Apache HttpClient for HTTP messaging
        consumer = new CommonsHttpOAuthConsumer(CONSUMER_KEY,
        ↪CONSUMER_SECRET);
        provider = new
        ↪CommonsHttpOAuthProvider("https://api.twitter.com/oauth/request_token",
        "https://api.twitter.com/oauth/access_token",
```

```

"https://api.twitter.com/oauth/authorize");
        String authUrl = provider.retrieveRequestToken(consumer,
➤CALLBACK_URL);
        Toast.makeText(this, "Authorize this app!",
➤Toast.LENGTH_LONG).show();
        this.startActivity(new Intent(Intent.ACTION_VIEW,
➤Uri.parse(authUrl)));
    } catch (Exception e) {
        Log.e(APP, e.getMessage());
        Toast.makeText(this, e.getMessage(),
➤Toast.LENGTH_LONG).show();
    }
}

```

Watch Out!

The default OAuth implementation in Android won't work with Twitter API. It is highly recommended not to use the DefaultOAuth* implementation in Android because there is a bug in Android's `java.net.HttpURLConnection` that keeps it from working with some service providers. Instead, CommonsHttpOAuth* classes are used in Android. You can find more detail information at <http://code.google.com/p/oauth-signpost>. You can find examples at <http://github.com/kaeppler/signpost-examples>.

Responding After Authentication

We add another `onNewIntent()` method to process after authenticating the Android application:

```

@Override
    protected void onNewIntent(Intent intent) {
    }

```

We retrieve the URI by using `intent.getData()` method. If the returning URL is not null and it matches the `CALLBACK_URL` string, we define the string variable `verifier` from `oauth.signpost.OAuth.OAUTH_VERIFIER` parameter:

```

Uri uri = intent.getData();
if (uri != null && uri.toString().startsWith(CALLBACK_URL))

```

This `getQueryParameter(String key)` method is to search the query string for the first value with the given key. We create a string object "verifier" to keep the returning query string from the search:

```

String verifier = uri.getQueryParameter(oauth.signpost.OAuth.OAUTH_
VERIFIER);

```

We populate the token and token_secret from the consumer object:

```
// Populate token and token_secret in consumer
provider.retrieveAccessToken(consumer, verifier);

// TODO: you might want to store token and token_secret in you app settings!
AccessToken a = new AccessToken(consumer.getToken(),
consumer.getTokenSecret());
```

Then we initialize Twitter4J:

```
// Initialize Twitter4J
ConfigurationBuilder confbuilder = new ConfigurationBuilder();
confbuilder.setOAuthAccessToken(a.getToken())
.setOAuthAccessTokenSecret(a.getTokenSecret())
.setOAuthConsumerKey(CONSUMER_KEY)
.setOAuthConsumerSecret(CONSUMER_SECRET);
twitter = new TwitterFactory(confbuilder.build()).getInstance();
```

The `System.currentTimeMillis()` method retrieves the date from the standard “wall” clock date/time in milliseconds since the epoch:

```
Date d = new Date(System.currentTimeMillis());
```

The `toLocaleString()` method (replaced by `DateFormat`) formats the date to the current Locale. Date/time is added to the end of the string as tweet.

We are creating a string object “tweet” for posting the message to a Twitter update. First, we include the Twitter message we like to post in a string “Twitter API 24 HRS: TwitterOAuth works on Android “. We add the `d.toLocaleString()` to the end. Twitter API has a detecting mechanism to prevent you from posting the same tweet message more than once in a given time period. By adding a date time string, each message we define will be different at a different date time. We will be able to pass the Twitter filtering system to show the message from our API call:

```
String tweet = "Twitter API 24 HRS: TwitterOAuth works on Android " +
d.toLocaleString();
```

The tweet is posted using `updateStatus()` method in twitter class:

```
twitter.updateStatus(tweet);
```

We display the tweet object inside the `tweetTextView`:

```
tweetTextView.setText(tweet);
```

We add the Toast to show the tweet object as an indicator of success:

```
Toast.makeText(this, tweet, Toast.LENGTH_LONG).show();
```

setVisibility(int visibility) method is to set the enabled state of the view. Options are VISIBLE, INVISIBLE, or GONE. GONE refers to invisible, but it doesn't take any space for layout. Basically, we make the button disappear from the layout in the last step:

```
buttonLogin.setVisibility(Button.GONE);
```

Now let's wrap this section. Here is the complete set of codes so you can follow and understand the order:

```

/**
 * Get the verifier from the callback URL.
 * Retrieve token and token_secret.
 * Feed them to twitter4j along with consumer key and secret
 */
@Override
protected void onNewIntent(Intent intent) {

    super.onNewIntent(intent);

    Uri uri = intent.getData();
    if (uri != null && uri.toString().startsWith(CALLBACK_URL)) {

        String verifier =
        ↪uri.getQueryParameter(oauth.signpost.OAuth.OAUTH_VERIFIER);

        try {
            // Populate token and token_secret in consumer
            provider.retrieveAccessToken(consumer,
            ↪verifier);

            // TODO: you might want to store token and
            ↪token_secret in you app settings!
            AccessToken a = new
            ↪AccessToken(consumer.getToken(), consumer.getTokenSecret());

            // Initialize Twitter4J
            ConfigurationBuilder confbuilder = new
            ↪ConfigurationBuilder();
            confbuilder.setOAuthAccessToken(a.getToken())
            ↪.setOAuthAccessTokenSecret(a.getTokenSecret())
                .setOAuthConsumerKey(CONSUMER_KEY)
                .setOAuthConsumerSecret(CONSUMER_SECRET);
            twitter = new
            ↪TwitterFactory(confbuilder.build()).getInstance();

            // Create a tweet
            Date d = new Date(System.currentTimeMillis());
            String tweet = "Twitter API 24 HRS: TwitterOAuth
            ↪works on Android " + d.toLocaleString();

            // Post the tweet

```

```

        twitter.updateStatus(tweet);

        // Show message
        tweetTextView.setText(tweet);
        Toast.makeText(this, tweet,
↳Toast.LENGTH_LONG).show();
            buttonLogin.setVisibility(Button.GONE);

        } catch (Exception e) {
            Log.e(APP, e.getMessage());
            Toast.makeText(this, e.getMessage(),
↳Toast.LENGTH_LONG).show();
        }
    }
}

```

Complete Code

This is the complete code in the `TwitterOAuth.java` file. It combines all the code we have discussed and explained in each section earlier:

```

package com.example.twitteroauth;

import java.sql.Date;

import oauth.signpost.OAuthConsumer;
import oauth.signpost.OAuthProvider;
import oauth.signpost.commonshttp.CommonsHttpOAuthConsumer;
import oauth.signpost.commonshttp.CommonsHttpOAuthProvider;
import twitter4j.Twitter;
import twitter4j.TwitterFactory;
import twitter4j.auth.AccessToken;
import twitter4j.conf.ConfigurationBuilder;
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;

/**
 * Using Twitter4J Java library and Signpost OAuth library to access
 * Twitter
 */
public class TwitterOAuth extends Activity {

    private static final String APP = "TWITTEROAUTH";

```



```

private Twitter twitter;
private OAuthProvider provider;
private OAuthConsumer consumer;

private String CONSUMER_KEY = "v7xwr2xFzDEsgDvg962Paw";
private String CONSUMER_SECRET =
↳"YlbknWgtA9t5u4q6q18IwHTQ2vt0lHesVB09u5EtEy4";
private String CALLBACK_URL = "myapp://twitteroauth";

private TextView tweetTextView;
private Button buttonLogin;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    tweetTextView = (TextView)findViewById(R.id.tweet);
    buttonLogin = (Button)findViewById(R.id.ButtonLogin);
    buttonLogin.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
            askOAuth();
        }
    });
}

/**
 * Direct to Twitter to authenticate the user
 */
private void askOAuth() {
    try {
        // Use Apache HttpClient for HTTP messaging
        consumer = new CommonsHttpOAuthConsumer(CONSUMER_KEY,
↳CONSUMER_SECRET);
        provider = new
↳CommonsHttpOAuthProvider("https://api.twitter.com/oauth/request_token",
↳"https://api.twitter.com/oauth/access_token",
↳"https://api.twitter.com/oauth/authorize");
        String authUrl =
↳provider.retrieveRequestToken(consumer, CALLBACK_URL);
        Toast.makeText(this, "Authorize this app!",
↳Toast.LENGTH_LONG).show();
        this.startActivity(new Intent(Intent.ACTION_VIEW,
↳Uri.parse(authUrl)));
    } catch (Exception e) {
        Log.e(APP, e.getMessage());
        Toast.makeText(this, e.getMessage(),
↳Toast.LENGTH_LONG).show();
    }
}

```

```

/**
 * Get the verifier from the callback URL.
 * Retrieve token and token_secret.
 * Feed them to twitter4j along with consumer key and secret
 */
@Override
protected void onNewIntent(Intent intent) {

    super.onNewIntent(intent);

    Uri uri = intent.getData();
    if (uri != null && uri.toString().startsWith(CALLBACK_URL)) {

        String verifier =
↳uri.getQueryParameter(oauth.signpost.OAuth.OAUTH_VERIFIER);

        try {
            // Populate token and token_secret in consumer
            provider.retrieveAccessToken(consumer,
↳verifier);

            // TODO: you might want to store token and
↳token_secret in you app settings!
            AccessToken a = new
↳AccessToken(consumer.getToken(), consumer.getTokenSecret());

            // Initialize Twitter4J
↳ConfigurationBuilder();
            ConfigurationBuilder confbuilder = new
            confbuilder.setOAuthAccessToken(a.getToken())

↳.setOAuthAccessTokenSecret(a.getTokenSecret())
                .setOAuthConsumerKey(CONSUMER_KEY)
                .setOAuthConsumerSecret(CONSUMER_SECRET);
            twitter = new
↳TwitterFactory(confbuilder.build()).getInstance();

            // Create a tweet
            Date d = new Date(System.currentTimeMillis());
            String tweet = "Twitter API 24 HRS: TwitterOAuth
↳works on Android " + d.toLocaleString();

            // Post the tweet
            twitter.updateStatus(tweet);

            // Show message
            tweetTextView.setText(tweet);
            Toast.makeText(this, tweet,
↳Toast.LENGTH_LONG).show();
            buttonLogin.setVisibility(Button.GONE);

        } catch (Exception e) {
            Log.e(APP, e.getMessage());
        }
    }
}

```

```

        Toast.makeText(this, e.getMessage(),
↳Toast.LENGTH_LONG).show();
    }
}
}
}
}
}

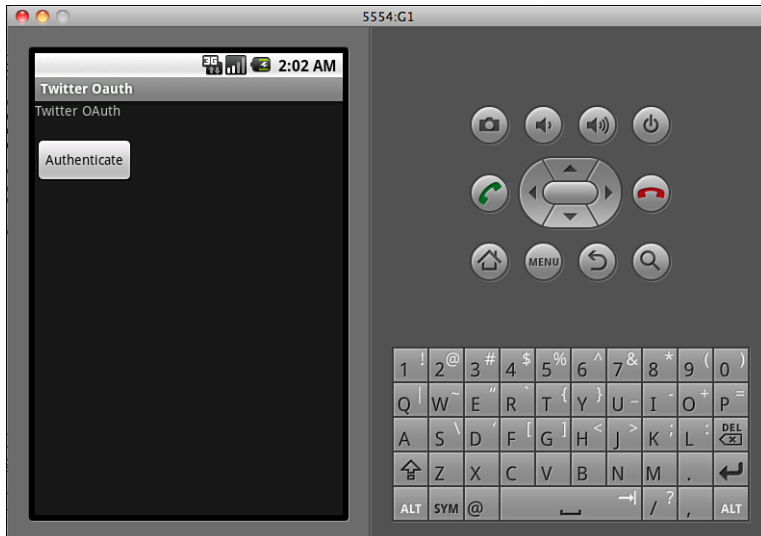
```

Did You Know?

Older versions of Twitter4J library use the `setOAuthAccessToken(AccessToken accessToken)` method. This method is deprecated. Use the `TwitterFactory.getInstance(twitter4j.http.Authorization)` method. Keep in mind that the Twitter4J library author is updating the library regularly, including changing methods without notice on each update. It's important that you understand how this works and be able to make any necessary changes in your own code.

Whew, quite a lot of code, eh? But now we are ready to build and run the Android app. By now, you should know the steps to run the Android app in Android Emulator inside Eclipse. In this Android example, we are creating a simple app with a text label and an Authenticate button, as shown in Figure 22.3. Once the button is pressed, it calls the method we explained previously.

FIGURE 22.3
Launching the Twitter OAuth in the Android emulator.



In this Android example, we are using OAuth to authenticate the user by redirecting to the Android browser that is pointing to the Twitter OAuth mobile page, as shown in Figure 22.4.

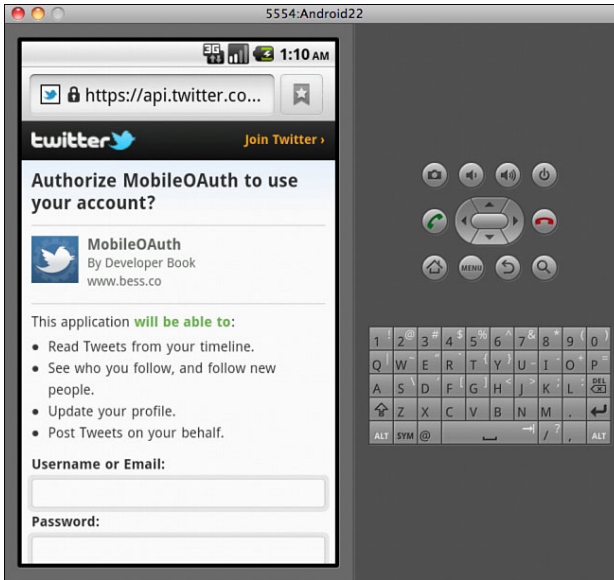


FIGURE 22.4 Showing Twitter authentication in the Android emulator.

As we can see in Figure 22.4, Twitter shows an OAuth page that is optimized for mobile devices. It shows the Twitter application name, developer name, and website. It also shows what the Twitter application is allowed to do if the user grants authentication. This is where you enter your Twitter account username and password, as shown in Figure 22.5.

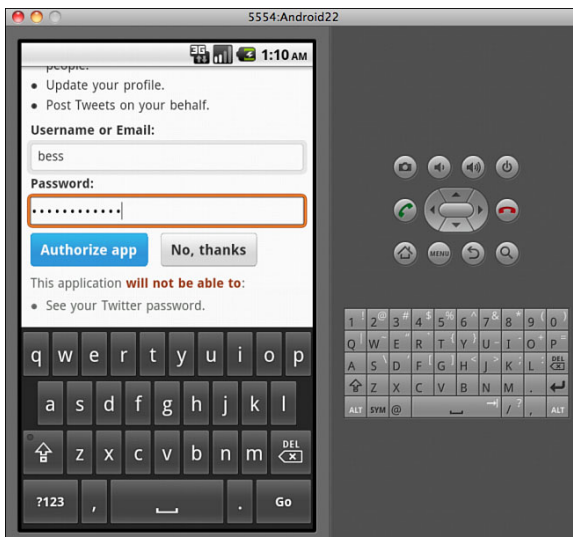
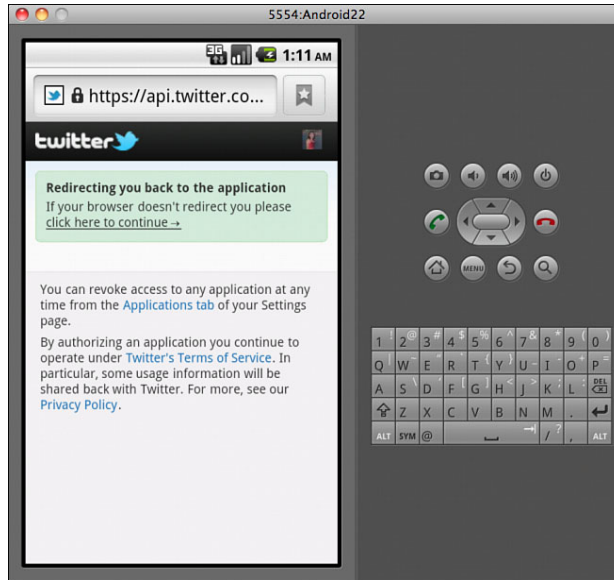


FIGURE 22.5 Entering your Twitter account and pressing the Authorize button in the Android emulator.

Scroll down to see the bottom of the Twitter OAuth page. After entering your username or email and password, press Authorize app button.

The Confirm dialog box will appear, asking if the user wants to keep the password on the browser or not. After closing this dialog box, you will be re-directed to another Twitter mobile page, as shown in Figure 22.6.

FIGURE 22.6
Showing the Twitter redirecting page to the app in the Android emulator.



After waiting a few seconds, you will be redirected from the device browser back to the Twitter Android app you defined in the code:

```
private String CALLBACK_URL = "myapp://twitteroauth";
```

After you launch and authenticate the Android application in the Android emulator, you will be redirected to the `CALLBACK_URL` that you define in the `TwitterOAuth.java` file. This overrides the callback URL that you save in your Twitter Application setting.

`Twitter4J` library allows developers to re-redirect the browser to the Android app without any additional modifications. In this chapter, we intend to show you the simplest method to authenticate OAuth. Ideally, you would create a `WebView` to show the Twitter OAuth screen within the same Android app.

After the user successfully grants authorization by verifying his or her Twitter account and then get redirected to the Twitter app, we hide the Authenticated button. Instead, we show him or her the posted tweet in the `TextView` in the place of the

Authenticated button and the Toast indicator at the bottom of the app, as shown in Figure 22.7. A successful posted tweet includes the date time stamp at the end of the tweet string you defined in the code.

Congratulations! You have a working Twitter OAuth app in Android.

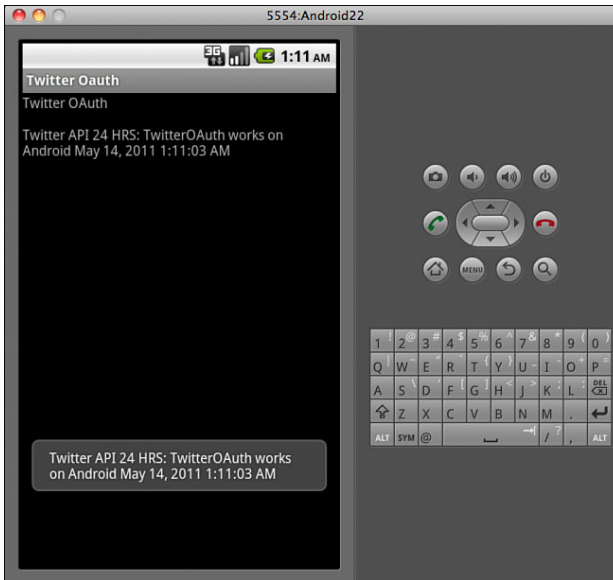


FIGURE 22.7 Showing the Twitter redirecting page to the app in the Android emulator.

xAuth

xAuth is alternative authenticate method. This method is available only to desktop or mobile apps. xAuth requires the developer to request permission to use it. Twitter will review your xAuth submission before granting its use. Your xAuth enabled or disabled status will not show in your Twitter application settings.

xAuth for Twitter is not the same as Xauth (<http://xauth.org/>). xAuth is still OAuth. The only difference is you skip the request_token and authorize steps of the OAuth flow. xAuth allows the mobile app to exchange the username and password for an OAuth access token.

According to Twitter's recommendation, once the access token is retrieved, Twitter suggests for developers to dispose of the username and password. Twitter also suggests for you to consider the standard web OAuth flow or PIN-code out-of-band flow before considering xAuth. xAuth requires you to use header-based OAuth authentication against an SSL access token endpoint, using the POST HTTP method. More details are at <http://dev.twitter.com/pages/xauth>.

The Twitter4J library does support xAuth as well as OAuth. In order to implement xAuth, you would have to create a new layout to provide a user interface to enter the Twitter account username and password. Due to the limitation of this hour's space, we decided not to cover xAuth. As such, we decided to use the OAuth method in our example.

The Twitter4J library enables you to use a CALLBACK_URL to redirect user from the Android browser to the Android app on the device. Twitter4J library supports the OAuth method without requiring the user to perform any steps to open the same Android app and without requiring developers to do any extra work to close the browser. Thus, because of this simplicity, we decided to go with the OAuth method.

Summary

Congratulations! You've added Twitter Java library and OAuth Java library to your Android OAuth project. You've created an Android application using the OAuth Consumer key and Consumer secret. And last, you have posted a tweet from your Android application using Android Emulator.

Q&A

- Q.** *What are the supporting Java libraries for Twitter API integration?*
- A.** Twitter4J library and JTwitter library.
- Q.** *What is the most accepting Java OAuth library for Android?*
- A.** Oauth-signpost library.

Workshop

Quiz

1. What authentication type does Twitter not support?
 - A. Basic Auth
 - B. OAuth
 - C. xAuth

2. What must be included in an Android application to request an OAuth token?
 - A. Consumer key
 - B. Consumer secret
 - C. Both
3. Which tab should you use to add third-party Java libraries (JARs) in the project's properties of Java Build Path?
 - A. Source
 - B. Projects
 - C. Libraries
 - D. Order and Export

Quiz Answers

1. A. Twitter does not accept Basic Auth due to security concerns.
2. C. You are required to include Consumer key and Consumer secret from your Twitter application settings.
3. C. You use the Libraries tab of Java Build Path inside Project's properties to add all the third-party Java libraries (JARs) for OAuth and Twitter libraries.

Exercises

1. Add Twitter Java libraries into your Android application.
2. Create a new Twitter application in your Twitter developer account. Save the Consumer key and Consumer secret into your Android application.
3. Create an Android application to send a tweet using OAuth.

This page intentionally left blank

HOOR 23

Getting Started with Twitter Using iOS

What You'll Learn in This Hour:

- ▶ Building your first Hello World application
- ▶ Using the Xcode and Interface Builder
- ▶ Requesting Twitter xAuth
- ▶ Using Twitter Objective-C library
- ▶ Verifying your Twitter xAuth

Introducing iOS

In 2007, Apple released iOS as its mobile operating system. In early 2008, Apple released the beta iPhone SDK for native application development. Apple announced the iPad as a tablet in early 2010 and iPhone 4 in the summer of 2010. Apple also renamed iPhone SDK to iOS SDK. iOS SDK supports iPhone, iPod Touch, iPad, and iPhone 4. iOS SDK is written in C, C++, and Objective-C. iOS SDK includes Xcode, Interface Builder, Instruments, and iPhone and iPad Simulator.

To learn about the iOS SDK and Xcode, we will start by creating a Hello World application on the iPhone. A simple Hello World application contains the basic iPhone application functionality and prints out the text “Hello World” on the screen—typically the first program you would create when learning any language. Because setting up the iOS environment is a bit tricky, we will spend a bit more time trying to get you up and running.

Creating a Hello World Application

We are sure you already know this, but you need to have a computer that can run the Apple OS X operating system running Snow Leopard or better, and you must have a license to publish iOS applications. Currently, a license will run you \$99.00 a year. You can download a copy of the iOS SDK without the license, but you will not be able to put anything in the store. So, if you have not done so already, download and install Xcode OS4 and let's get going.

Open Xcode to create a new project. Xcode will create all the required files for an iOS iPhone application. Follow these steps to create a new project:

1. In Xcode, select File, New, New Project...
2. In the New Project window, select iOS, Application on the OS panel, as shown in Figure 23.1.

FIGURE 23.1
Selecting View-based Application.



3. In the template panel, select View-based Application. Click Next.
4. In the Project Options window, enter helloworld in the Product Name field. Leave the Company Identifier field as it is as shown in Figure 23.2.
5. Select iPhone in the Device Family drop-down menu. Click Next.
6. Select the directory to which you would like to save the Xcode project. Save the project by clicking Create.
7. In the Xcode's Project Navigator inside the Xcode project, the template files are created inside the Group folder named helloworld, as shown in Figure 23.3.

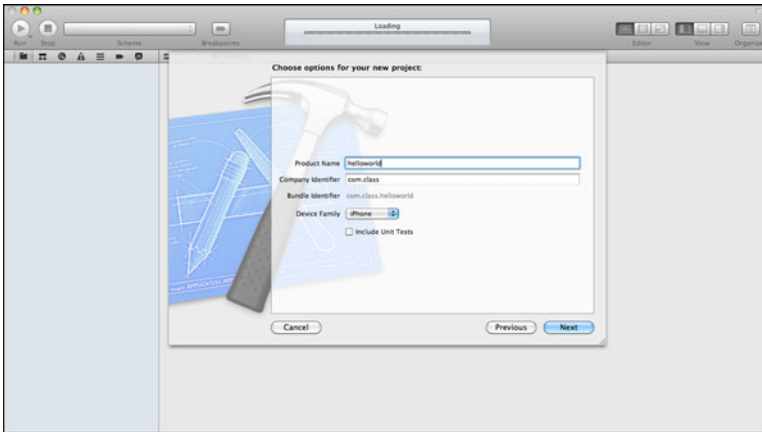


FIGURE 23.2
Naming the helloworld in the Product Name field.

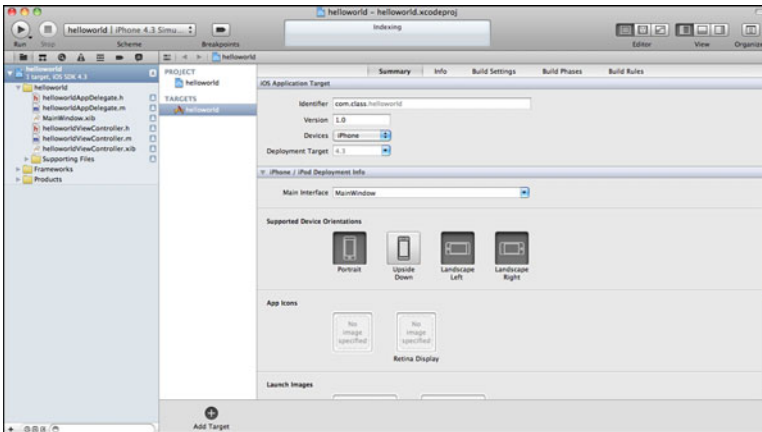
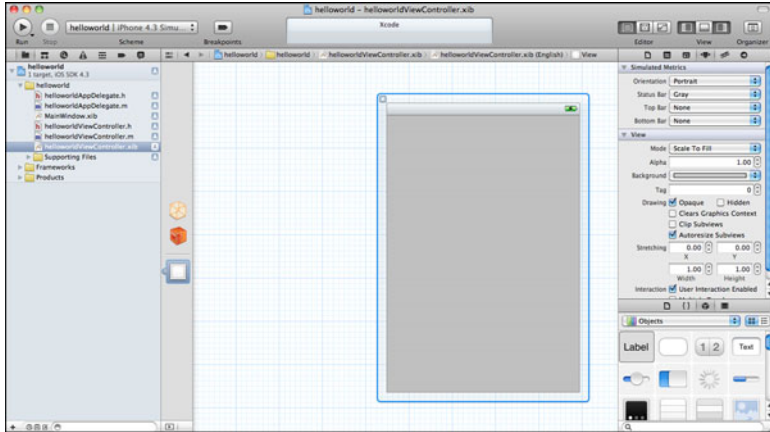


FIGURE 23.3
Showing Xcode project summary panel after creating the project.

8. Click the file helloworldViewController.xib to launch Interface Builder in the editor area of the workspace window.
9. Click the View Right Panel icon in the Selector Bar to show the Utility area.
10. Select the Attributes icon in the Inspector Selector Bar to show the Attributes Inspector.
11. Click on the Interface Builder object View inside the Editor area. Attributes Inspector will display the UIView's details.
12. In the View section, select Default in the Background drop-down menu to switch away from the gray background. It gives a white background.
13. Select the Object icon in the Library selector bar to show the Object Library.

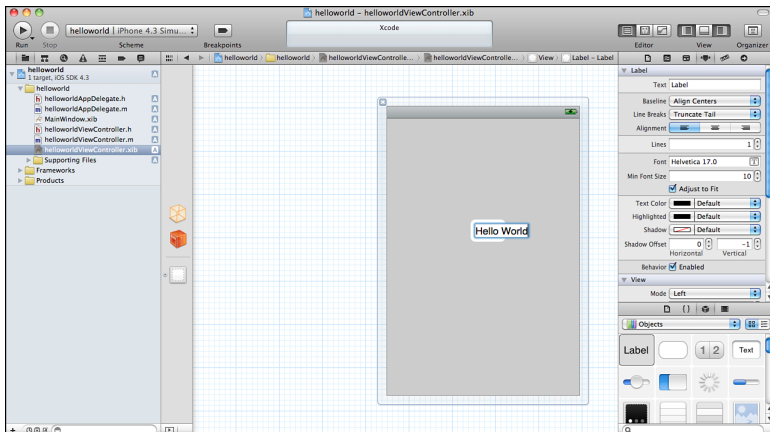
14. Select the Icon View icon or List View icon to show the Object Library objects, as shown in Figure 23.4.

FIGURE 23.4
Showing Attributes Inspector and Object Library in Utility area.



15. Select the Label and drag it to the View inside the Editor area.
16. Double-click the Label object until the entire label is selected in the View object.
17. Enter the text "Hello World" inside the Label object (as shown in Figure 23.5) and press Enter on the keyboard.

FIGURE 23.5
Showing the "Hello World" label on the view in the Editor area.



18. Select File, Save or hold both <Command>+S key buttons to save the file.

19. In the Xcode, select Product, Run, or click the Run button at the top left of the Xcode window.

If everything went well, you should have a working application displaying the text “Hello World,” as shown in Figure 23.6.



FIGURE 23.6
Showing “Hello World” in the iPhone Simulator.

Objects will not be displayed on the iPhone Simulator in the Run and Build process until the objects inside the Interface Builder are properly saved. When the file icon is highlighted in gray in the Project Navigator, it indicates that the file changes are not saved in Xcode. Build and Run won't render the updates.

Did You Know?

Creating a Twitter Application for xAuth Request

Before you can request and activate xAuth, you need to create a Twitter application. You should have done this already in Hour 8, “Twitter OAuth,” but if not, be sure to do this now.

To do so, go to <http://twitter.com/apps>.

1. Click the link “Register a new application” at the bottom of the page.
2. Complete the application form.
3. Click Save.

Requesting Twitter xAuth

In Hour 8, we discussed registering your application and getting an OAuth number. In most cases, this should be fine. However, you can also request something called xAuth, which is more tailored for mobile apps. However, this spec is ever-changing and evolving. Be sure to check Twitter for the latest updates with xAuth.

You can request for xAuth activation by opening a support ticket at <http://support.twitter.com/forms>. Select “Everything else” in the menu. Include your App ID of your app in your support ticket. Your App ID is the number at the end of the URL of your app:

http://twitter.com/oauth_clients/details/<your app id>

Twitter xAuth API

After it is approved, you will be able to make a call to the following API method:

http://api.twitter.com/oauth/access_token

and send three parameters:

```
x_auth_username - username  
x_auth_password - password  
x_auth_mode - set the value to "client_auth"
```

This API method returns the authorized access token similar to OAuth. xAuth authorized tokens can be saved and reused for any subsequent API calls because the tokens do not expire.

Retrieving Consumer Key and Consumer Secret

After Twitter API approves your xAuth request and activates xAuth on your Twitter application, you can retrieve your Consumer key and Consumer secret by visiting <http://dev.twitter.com/apps/<your app id>> or http://twitter.com/oauth_clients/details/<your app id>.

Now let's log into your Twitter account to retrieve your App ID, the Consumer key, and Consumer secret and copy them into a text file, as we did in Hour 8. We will use them in the code later:

1. Log in to your Twitter account.
2. Select your xAuth-activated application at <http://twitter.com/apps>.

After selecting your application, you should see the Application Details. This is where you find the Consumer key and Consumer secret.

3. Retrieve your App ID by looking up the application URL. Your App ID is the number at the end of the URL of your app:
`http://twitter.com/oauth_clients/details/<your app id>`.
4. Copy the Consumer key and Consumer secret and store them into a notepad.
5. Click Edit Application Settings.

The Application Settings is where you define the Application Type and Default Access Type. You should choose Client as Application Type to support xAuth on iPhone app. You should choose Read & Write on Default Access Type in this application.
6. Select Client as the Application Type.
7. Select Read & Write on Default Access Type.

Downloading MGTwitterEngine Library

The Twitter developer website suggests a list of Twitter Libraries at <http://dev.twitter.com/pages/libraries>. MGTwitterEngine is the most accepted Objective-C Cocoa library by iOS developers. This Twitter library includes OAuth and xAuth support for Mac OS X and iOS.

MGTwitterEngine is an Objective-C Cocoa library for the Twitter API. You can download the latest version of MGTwitterEngine at <https://github.com/mattgummell/MGTwitterEngine>.

xAuth implementation in iOS application requires additional resources in addition to the MGTwitterEngine library.

Downloading MGTwitterEngineDemo

MGTwitterEngineDemo is a complete Xcode project packaged with additional resources to support Twitter xAuth on an iPhone application. You can download the latest version of MGTwitterEngineDemo at <https://github.com/ara1/MGTwitterEngineDemo>.

It also allows you to verify your Twitter application xAuth activation and test your Consumer key and Consumer secret.

Twitter application settings do not support xAuth status. The only method to verify your xAuth approval and activation is the email confirmation from Twitter API.

Targeting Active SDK

The latest build on MGTwitterEngineDemo may not support your latest or previous iOS SDK you installed for your Xcode. You would need to take additional steps to select the Base SDK:

1. Launch the MGTwitterEngineDemo project in Xcode by opening MGTwitterEngineDemo.xcodeproj in the folder.
2. TARGETS should be selected by Default, showing the Summary tab.
3. Select the Building Settings tab.
4. In Architectures, Base SDK, select the latest iOS Device SDK version from the drop-down list.
5. Click on the Scheme button at the top left of Xcode. Select iPhone Simulator from the drop-down list to set the active scheme.
6. Click the Run button at the top left of the Xcode to run the project.

Verifying OAuth

MGTwitterEngineDemo requires OAuth-activated Consumer key and Consumer secret to call Twitter API. Twitter is no longer allowing apps to store Twitter usernames and passwords on the device to log into the user's Twitter account for the Basic Auth method. In order to integrate OAuth, you are required to include the Twitter OAuth Consumer key and Consumer secret in the app to authenticate the app.

Inside the MGTwitterEngineDemoViewController.h file, #define statement is where you define the constant of Twitter OAuth Consumer key and Consumer secret.

In MGTwitterEngineDemoViewController.h:

```
#define kOAuthConsumerKey    @" "
#define kOAuthConsumerSecret @" "
```

If you run the project without including your application Twitter OAuth Consumer key and Consumer secret, you will be receiving an alert box with a title "Missing OAuth details," as shown in Figure 23.7.



FIGURE 23.7
Showing the Missing OAuth details in iPhone Simulator alert.

Now you retrieve the Consumer key and Consumer secret you saved in the text file. Copy those values inside the Objective-C String. The value should be copied inside the double quotes after the @ symbol:

1. In `MGTwitterEngineDemoViewController.h`, enter your Consumer key in the Objective-C string @"@" of `kOAuthConsumerKey` constant.
2. Enter your Consumer secret in the Objective-C string @"@" of `kOAuthConsumerSecret` constant.
3. Click the Run button at the top left of the Xcode.

After running the project, you will see two empty text field input boxes in the iPhone Simulator. Enter your valid Twitter username and password under the title "Twitter account details," as shown in Figure 23.8.

FIGURE 23.8
Showing the
Twitter account
form in the
iPhone
Simulator.



4. Enter your valid Twitter username and password in the iPhone Simulator.
5. Click the Get xAuth Access Token button.

Selecting the Get xAuth Access Token button will send a Twitter API request to Twitter to authenticate your app based on your Twitter OAuth Consumer key and Consumer secret. Before selecting the Get xAuth Access Token button, the Send Test Tweet button is disabled and is grayed out. After the xAuth is successfully authenticated by Twitter, the Send Test Tweet button will be enabled.

6. Click the Send Test Tweet button when it is enabled.

The Send Test Tweet button will send a hard-coded Twitter message to your Twitter account. You will receive an alert box title "Tweet sent!," as shown in Figure 23.9.



FIGURE 23.9 Showing the success alert message in the iPhone Simulator.

7. Log in to your Twitter account to verify the successful tweet message in your Twitter timeline feed, as shown in Figure 23.10.

You can log into your Twitter account to see the test message on your Home Timeline status. A random number is generated at the end of the message to avoid Twitter from blocking any duplicated message.

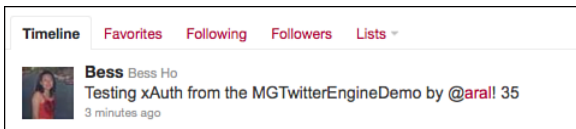


FIGURE 23.10 Showing the successful tweet message on Twitter feed.

Summary

Congratulations! You are now an iOS developer. You've created your first iPhone project. You also have created a Twitter application and requested xAuth from Twitter. Finally, you learned how to verify your xAuth tokens by testing your application's Consumer key and Consumer secret.

Q&A

- Q. What kind of a development environment do I need for creating an iPhone application?*
- A.** You need the latest iOS SDK, Xcode, Interface Builder, and iPhone Simulator.
- Q. What operating systems are supported for iOS application development?*
- A.** Mac OS X Snow Leopard.
- Q. What is xAuth?*
- A.** xAuth is Twitter's preferred authentication method for iOS native app after Twitter discontinued the support in Basic Authentication.

Workshop

Quiz

- Which programming languages do I use to code and develop an iPhone application?
 - Objective-C
 - Java
 - HTML / JavaScript / CSS
 - C++
- What are the types of authentications Twitter accepts?
 - Basic Authentication
 - OAuth
 - xAuth
 - B and C only
- How do you request Twitter xAuth?
 - Twitter generates xAuth when you create your Twitter developer account.
 - Twitter generates xAuth when you create a new Twitter application.
 - Email Twitter at api@twitter.com with your application ID to request xAuth.

Quiz Answers

1. A. Use Objective-C to write the code for the iOS application.
2. D. Twitter accepts OAuth and xAuth and bans the use of Basic Authentication.
3. C. Twitter manually approves xAuth on each request.

Exercises

1. Register the Apple developer account. Install your iOS SDK. Create a view-based template project in Xcode. Create a hello iPhone application.
2. Build and run the application in the iPhone Simulator.
3. Create a new Twitter application. Find your application's Consumer key and Consumer secret, and your App ID.
4. Request xAuth from Twitter.
5. Verify your xAuth after Twitter activates your xAuth.

This page intentionally left blank

HOURL 24

Building an iPhone and iPod Touch Application with Twitter

What You'll Learn in This Hour:

- ▶ Creating an iPhone xAuth application
- ▶ Adding Objective-C library
- ▶ Creating layout in Interface Builder
- ▶ Using Objective-C xAuth library
- ▶ Posting a Tweet from iPhone application

Introducing Twitter xAuth

Twitter introduces and extends xAuth to support mobile application and authenticates the mobile application without storing the username and password in the application.

In the iOS application, the Twitter OAuth authentication process forces the user to exit the app and to authorize the request on the Twitter website in an opened Safari browser. Initially, the Twitter OAuth web interface was not optimized for mobile. The option of using UIWebView to request and exchange OAuth tokens within the same iOS application is considered a hacked approach. Developers have reported that it takes a default preset 10 seconds for Twitter to redirect to the OAuth successful page within UIWebView. Twitter has not responded to any suggestion to shorten or optimize the 10-second redirect for mobile native apps.

Because Twitter bans the use of Basic Auth and disallows storing Twitter usernames and passwords on devices, the next option Twitter embraced for mobile native apps is xAuth.

xAuth is a preferred authentication method for iOS application. Developers can exchange both usernames and passwords for authorized tokens in one API call using xAuth.

Benefits of Using Twitter xAuth

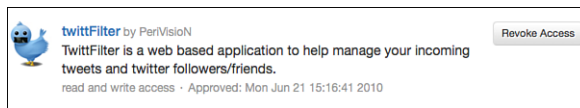
Using xAuth enhances your organic marketing effort by displaying the source parameter at the bottom of every tweet (display as “via My App”).

Did You Know?

Twitter initially allowed applications to create a source parameter for non-OAuth use. Twitter no longer accepts new registrations for source request. However, applications using pre-OAuth source parameters will remain active.

xAuth improves security in case of a stolen device. You can revoke access to your application by visiting your Twitter account settings; select the Connections tab at <http://twitter.com/settings/connections> to see the list of Twitter applications. Use the “Revoke Access” button on the right side of the application to enable Revoke Access, as shown in Figure 24.1.

FIGURE 24.1
Showing the Revoke Access in Connections panel for Twitter account settings.



Selecting Twitter Objective-C Libraries

MGTwitterEngine is one of the few Objective-C libraries supporting Twitter APIs. MGTwitterEngine supports OAuth and xAuth, as well as iOS SDK for iPhone, iPod, and iPad applications. MGTwitterEngineDemo contains the JSON library, OAuth library, and helper files to support MGTwitterEngine Twitter library.

Creating xAuth Application

MGTwitterEngineDemo is a nice way to verify your xAuth activation, validate your Consumer key and Consumer secret, and confirm your Twitter username and password. However, it is not for designing commercial application containing many alerts for debugging. Let's start with a new project:

1. In Xcode, select File, New, New Project....
2. Select the View-based Application template in the New Project window.
3. Click Next.
4. Enter xauth in the Product Name input box. Select iPhone in the drop-down menu of Device Family.
5. Click Next.
6. Select the directory to save the Xcode project.
7. Click Create.
8. Click Build Phases tab under Project TARGETS.
9. Expand the arrow icon next to Link Binary With Libraries item.
10. Click + icon to open the Frameworks & Libraries drop-down list.

MGTwitterEngine requires additional frameworks to support the Twitter OAuth library. Security.framework and libxml2.dylib are added to the Project TARGETS in the Build Phases tab. You can add each framework one by one or you can hold down the Command key on the keyboard to select more than one framework, as shown in Figure 24.2.

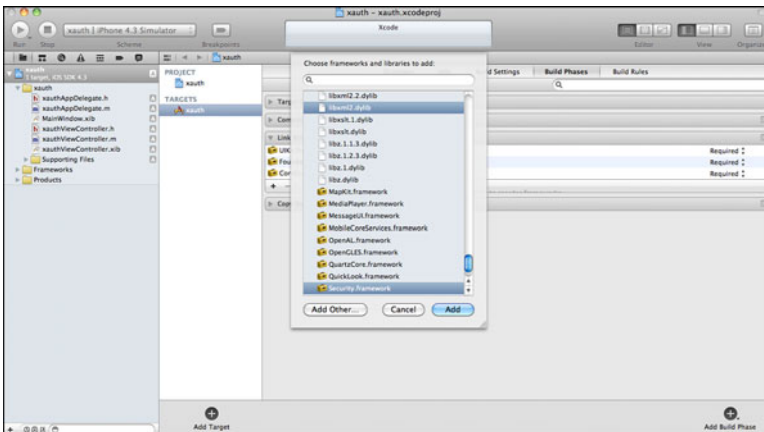
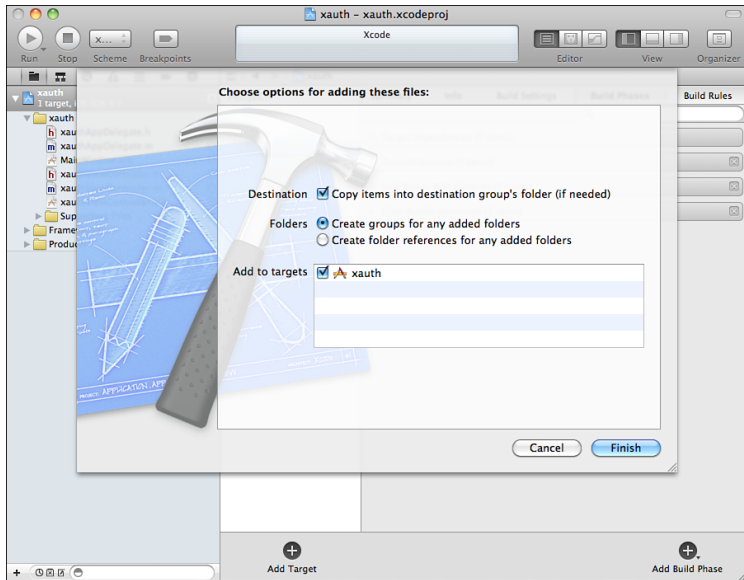


FIGURE 24.2
Adding Frameworks to Xcode project.

11. Select Security.framework and click Add.
12. Select libxml2.dylib and click Add.

13. Select both Security.framework and libxml2.dylib and drag them into the Frameworks Group.
14. Open the MGTwitterEngineDemo Xcode project. Select the Libraries Group from the Project Navigator panel. Drag the entire Libraries Group from the MGTwitterEngineDemo Xcode project into your xauth Group of your new Xcode project inside the Project Navigator panel. You should see a green + icon when you drag the folder into the Project Navigator panel.
15. Check the Destination box Copy items into destination group's folder (if needed).
16. The Folder radio button should be selected for Create groups for any added folders, as shown in Figure 24.3.

FIGURE 24.3
Copying the library.



17. Click Finish.
18. Click Build Settings tab under Project TARGETS.
19. Click the search box.
20. Enter Header Search Paths into the search box.
21. Enter return.
22. Select Header Search Paths in the Setting Column, as shown in Figure 24.4.
23. Add $\$(SDKROOT)/usr/include/libxml2$ in the column Value.

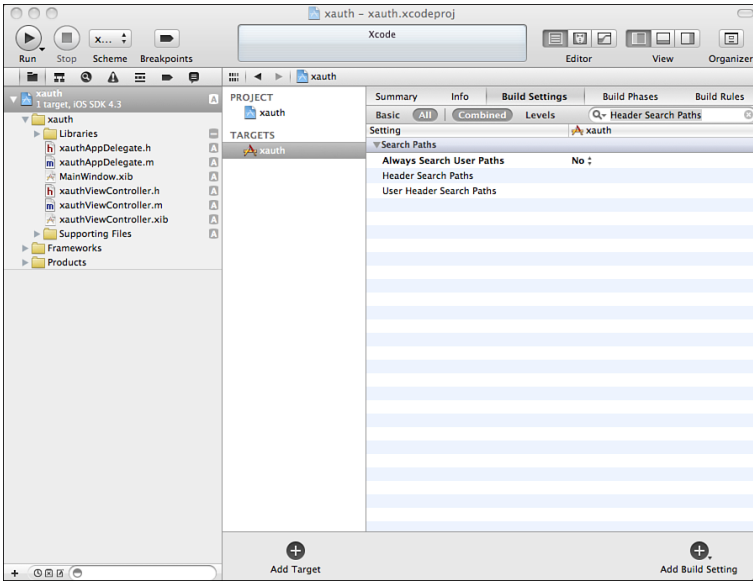


FIGURE 24.4
Targeting Header Search Paths in the Build tab.

If you build the project now without libXML framework libxml2.dylib, you will get hundreds of errors. The classes from MGTwitterEngineDemo require libXML to be a target of the project build. Make sure you add the framework libxml2.dylib.

Watch Out!

The MGTwitterEngineDemo library uses third-party libraries OAuth Consumer library to support OAuth and scifihifi-iphone-security library to store account information in the keychain for security. Both OAuth Consumer library and scifihifi-iphone-security library require iOS SDK Security framework (Security.framework).

MGTwitterEngineDemo library is required to parse Twitter XML responses. Based on the MGTwitterEngineDemo recommendation, LibXML parser is preferred over the NSXMLParser. Libxml2 included by Apple in the iOS SDK is used to perform tree-based parsing. Because libxml2 is a .dylib and is not an iOS SDK framework, it is necessary to include libxml2 path in the Project Target’s Build Settings tab in the “Header Search Paths”.

Now we have added the missing Security Framework and LibXML parser .dylib to the project. We can explore the files inside the MGTwitterEngineDemo library.

MGTwitterEngineDemo library is consisting of several third-party libraries. These libraries are grouped together inside Twitter+OAuth and CocoaHelpers Groups.

MGTwitterEngineDemo library includes Twitter+OAuth and CocoaHelpers folders.

MGTwitterEngineDemo library contains the following:

- ▶ **MGTwitterEngine library**—It supports Twitter APIs.
- ▶ **TouchJSON library**—It supports JSON parsing.
- ▶ **OAuthConsumer library**—It supports Twitter OAuth.
- ▶ **scifihifi-iphone-security library**—It supports Keychain storage and retrieval. Source code is at <https://github.com/ldandersen/scifihifi-iphone/tree/master/security/>.
- ▶ **CocoaHelpers**—It is a collection of Objective-C class files to generate alerts to inform users.

Did You Know?

LibXML is a native parser. Unlike NSXMLParser, LibXML is faster with a smaller memory footprint.

YAJL is JSON library written in ANSI C. YAJL is a small event-driven (SAX-style) JSON parser and a validating JSON generator. You can find more info about YAJL library at <http://lloyd.github.com/yajl/>.

The MGTwitterEngine library does not include the source files from the TouchJSON library and YAJL library. The MGTwitterEngine library requires users to include the source files in your Xcode project. Because both TouchJSON and YAJL libraries are JSON parsers, there is no reason to keep two similar JSON libraries in the same Xcode project. In this xAuth implementation, the TouchJSON library is used as a JSON parser instead of YAJL library. It is necessary to remove all the YAJL dependency files from the compile sources in the project target in order to run the xAuth version of MGTwitterEngine in MGTwitterEngineDemo.

So, let's remove those dependency files in the following steps:

1. Click Build Phases tab of Project TARGETS.
2. Click the arrow next to Compile Sources item.
3. Delete the following YAJL dependency files by selecting the file and click the minus sign button to delete the file:
 - ▶ MGTwitterMessagesYAJLParser.m
 - ▶ MGTwitterStatusesYAJLParser.m
 - ▶ MGTwitterYAJLParser.m
 - ▶ MGTwitterSearchYAJLParser.m

- ▶ MGTwitterUsersYAJLParser.m
- ▶ MGTwitterMiscYAJLParser.m

4. Click Run icon to run the project.

The latest release of YAJL is not compatible with the MGTwitterEngine library. It is best to remove all the YAJL dependency files at the target.

**Watch
Out!**

Exploring ViewController.h

The view-based application template generates both header (.h) and implementation (.m) files for your <project name>ViewController in Xcode project. In the Header file, you declare variable instances, properties, and methods.

In xauthViewController.h:

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

@interface xauthViewController : UIViewController {
}

@end
```

Importing Libraries to Header Files

Add the libraries from MGTwitterEngine and OAuth in the ViewController.h file.

In xauthViewController.h, add the import statements of header files:

```
#import "MGTwitterEngineDelegate.h"
#import "OAToken.h"
```

Add the following constants:

```
#define kOAuthConsumerKey           @"<Consumer key>"
#define kOAuthConsumerSecret       @"<Consumer secret>"
#define kTokenKey                  @"tokenKey"
#define kHaveCachedToken           @"haveCachedToken"
#define kMGTwitterEngineDemoServiceName @"MGTwitterEngineDemoService"
```

Add class MGTwitterEngineDelegate:

```
@class MGTwitterEngine;
```

Add MGTwitterEngineDelegate protocol to the @interface:

```
@interface xauthViewController : UIViewController
<MGTwitterEngineDelegate> {

}
```

Declare Variable Instances

Before we can use any objects such as TextField, TextView, or Button in ViewController, we need to declare each object type and object variable instance in @interface directive.

In xauthViewController.h, add the following variable instances:

```
@interface xauthViewController : UIViewController
<MGTwitterEngineDelegate> {
    UITextField *usernameTextField;
    UITextField *passwordTextField;
    UITextView *messageTextView;
    UIButton *postButton;
    MGTwitterEngine *twitterEngine;}

```

Declare Properties and Methods

We declare the property types and add IBOutlet to each object. We also declare the method used in ViewController's implementation .m file.

In xauthViewController.h, add the following @property and methods:

```
@property (nonatomic, retain) IBOutlet UITextField *usernameTextField;
@property (nonatomic, retain) IBOutlet UITextField *passwordTextField;
@property (nonatomic, retain) IBOutlet UITextView *messageTextView;
@property (nonatomic, retain) IBOutlet UIButton *postButton;
@property (nonatomic, retain) MGTwitterEngine *twitterEngine;

- (IBAction)postMessage;
```

Exploring ViewController.m

The implementation .m file of ViewController is where you execute the Objective-C methods. The view-based template you selected in creating the new Xcode project would generate some templates files including xauthViewController.m file. The codes you see in xauthViewController.m are default codes from the view-based template.

In xauthViewController.m:

```
#import "xauthViewController.h"
@implementation xauthViewController
```

```
- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}
- (void)viewDidUnload {
}
- (void)dealloc {
    [super dealloc];
}
@end
```

Importing Libraries to Implementation File

Include the supporting libraries once with the #import directive" above the @implementation directive in xauthViewController.m:

```
#import "MGTwitterEngine.h"
#import "SFHFKeychainUtils.h"
#import "UIAlertView+Helper.h"
#import "OAToken.h"
```

After the @implementation directive, add the following to xauthViewController.m:

```
@synthesize usernameTextField;
@synthesize passwordTextField;
@synthesize messageTextView;
@synthesize postButton;
@synthesize twitterEngine;
```

@synthesize directive will generate getter and setter methods for the property you defined in the header .h file.

Adding Pragma Mark Directive

#pragma mark is a simple directive that helps us to organize our implementation code. It functions similar to bookmarks. It helps developers to jump to user-defined sections by adding #pragma mark in the project navigation toolbar above the Xcode's Editor area. #pragma mark adds a line, and #pragma mark <title> adds a title to the project navigation toolbar drop-down menu.

Add the following Pragma Mark directives to xauthViewController.m:

```
#pragma mark -
#pragma mark Memory management
```

Managing Memory

The viewDidUnload method is called when controller's view is released from memory. This method resets the objects on the view.

The `dealloc` method is called when controller deallocates the memory occupied by the receiver. This method releases the memory of the objects on the view.

After `@synthesize`, add the following in `xauthViewController.m`:

```
- (void)viewDidUnload {
    self.usernameTextField = nil;
    self.passwordTextField = nil;
    self.messageTextView = nil;
    self.postButton = nil;
}

- (void)dealloc {
    [usernameTextField release];
    [passwordTextField release];
    [messageTextView release];
    [postButton release];
    [twitterEngine release];
    [super dealloc];
}
```

Initializing MGTwitterEngine

The `initWithCoder:` method is used to create a variable instance of the `MGTwitterEngine` object called `twitterEngine`.

Add `#pragma` mark directives and the `initWithCoder:` method in `xauthViewController.m`:

```
-(id)initWithCoder:(NSCoder *)aDecoder{
    self = [super initWithCoder:aDecoder];
    if (self)
    {
        // Custom initialization
        self.twitterEngine = [[[MGTwitterEngine alloc]
initWithDelegate:self] autorelease];
    }
    return self;
}
```

Loading xAuth Token

The `viewDidLoad` method is called after the controller's view is loaded into memory. This method is where we handle the success and error if there is any missing Consumer key and Consumer secret in the application, and if `xAuth` token is cached and loaded from the Keychain.

Token key is saved in NSUserDefaults and token secret is stored in the Keychain using SFHFKeychainUtils. SFHFKeychainUtils offers better security than storing both token keys and secrets in NSUserDefaults.

The xAuth access token is stored as an NSString object. This cached token is used for future sessions.

Twitter apps that use multiple accounts, or apps allowing users to change the account in apps with a single account, update the xAuth access token manually within the session.

The view-based template generates commented viewDidLoad method by default. Delete the entire default viewDidLoad method and add our own viewDidLoad method instead.

Add the following viewDidLoad method in xauthViewController.m:

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Sanity check
    if ([kOAuthConsumerKey isEqualToString:@""] || [kOAuthConsumerSecret
    ↪isEqualsToString:@""])
    {
        NSLog(@"Twitter Consumer key or Consumer secret is missing!");
    }
    else
    {
        [self.twitterEngine setConsumerKey:kOAuthConsumerKey
    ↪secret:kOAuthConsumerSecret];
    }

    NSUserDefaults *userDefaults = [NSUserDefaults
    ↪standardUserDefaults];
    if ([userDefaults boolForKey:kHaveCachedToken])
    {
        // Get the cached (saved) token.
        NSString *tokenKey = [userDefaults
    ↪objectForKey:kTokenKey];

        NSError *error = nil;
        NSString *tokenSecret = [SFHFKeychainUtils
    ↪getPasswordForUsername:tokenKey
    ↪andServiceName:kMGTwitterEngineDemoServiceName
    ↪sharedKeychainAccessGroupName:nil error:&error];
        if (error)
        {
            // Error loading oAuth token from keychain
            NSLog( Can't load OAuth token from Keychain %d: %@",
    ↪[error code], [error localizedDescription]);
        }
        else
    }
}
```

```

    {
        // Success in loading OAuth token from keychain
        OAuthToken *token = [[OAuthToken alloc] initWithKey:tokenKey
        ↪secret:tokenSecret];

        self.twitterEngine.accessToken = token;

        NSLog( xAuth token cached");
        self.postButton.enabled = YES;
    }

    // Set initial focus.
    [self.usernameTextField becomeFirstResponder];
}

```

Did You Know?

MGTwitterEngineDemo uses UIAlertView to display messages for debugging. NSLog is replacing UIAlertView displaying messages in Console for debugging.

Posting Tweet

A new method `postMessage` is created to connect the UIButton named `postButton` created in Interface Builder via (IBAction).

First, the username, password, and message are stored in NSString objects. Then, the `getOAuthAccessTokenForUsername:password:` method retrieves xAuth access token from Twitter API.

To avoid Twitter's 403 Status Is a Duplicate error, we add a random number to the end of the message using random generator `arc4random()%144`. Display the entire message in an NSString object named `tweetText`.

Finally, use `sendUpdate:` method to post the message to Twitter feed in Timeline. Immediately release all the NSString object memory.

Add `#pragma` mark directives and a `postMessage` method in `xauthViewController.m`:

```

#pragma mark -
#pragma mark Actionable methods

- (IBAction)postMessage {
    // Retrieve your username and password from Textfields
    NSString *username = self.usernameTextField.text;
    NSString *password = self.passwordTextField.text;
    NSString *message = self.messageTextView.text;

    // Retrieve xAuth access token from username and password

```

```

    NSLog(@" Password: %@", username, password);
    NSLog(@" %@", message);
    [self.twitterEngine getXAuthAccessTokenForUsername:username
    ↪password:password];

    // Adding random number to the tweet to avoid Twitter's 403 "Status is
    ↪a duplicate" error.
    NSString *tweetText = [NSString stringWithFormat:@"%@ %d", message,
    ↪arc4random()%144];

    NSLog(@"About to post message to Twitter: \"%@\\"", tweetText);

    [self.twitterEngine sendUpdate:tweetText];
    [username release];
    [password release];
    [message release];
    [tweetText release];
}

```

Adding MGTwitterEngine Delegate Methods

These are the MGTwitterEngine Delegate methods that would return the resulting responses from Twitter APIs. Each Delegate method will return a specific message in an alert window:

- ▶ statusesReceived:(NSArray *)statuses forRequest:(NSString *)connectionIdentifier
- ▶ accessTokenReceived:(OAuthToken *)token forRequest:(NSString *)connectionIdentifier
- ▶ requestSucceeded:(NSString *)connectionIdentifier
- ▶ requestFailed:(NSString *)connectionIdentifier withError:(NSError *)error

Add #pragma mark directives and MGTwitterEngine Delegate method in xauthViewController.m:

```

#pragma mark -
#pragma mark MGTwitterEngineDelegate methods

- (void)statusesReceived:(NSArray *)statuses forRequest:(NSString *)connectionIdentifier{
    // Since we're just sending a tweet in this example, we can
    ↪assume that's the tweet that's returned
    // and use this as a success handler.
    UIAlertViewQuick(@"Tweet sent!", @"The tweet was successfully
    ↪sent. Everything works!", @"OK");
}

```

```

}

- (void)accessTokenReceived:(OAToken *)token forRequest:(NSString
*)connectionIdentifier{
    //
    // We've got an oAuth access token from Twitter. Let's save it.
    //
    NSString *tokenKey = token.key;
    NSString *tokenSecret = token.secret;

    // Save the token securely in the keychain.
    // (Note: this SFHFKeychainUtils method doesn't return a value.)
    NSError *error = nil;
    [SFHFKeychainUtils storeUsername:tokenKey andPassword:tokenSecret
    ↪forService-Name:kMGTwitterEngineDemoServiceName
    ↪sharedKeychainAccessGroupName:nil updateExisting:YES error:&error];
    if (error)
    {
        NSString *errorMessage = [NSString
    ↪stringWithFormat:@"Error saving to-ken", @"I couldn't save the oAuth
    ↪token to the keychain. %d: %@", [error code], [errorlocalized
    ↪Description]];
        UIAlertViewQuick(@"Error saving token", errorMessage,
    ↪@"OK");
    }
    else
    {
        // Save the token key and flag that we have a cached
    ↪token.

        NSLog(@"Got the oAuth token and about to save it.");
        NSUserDefaults *userDefaults = [NSUserDefaults
    ↪standardUserDefaults];
        [userDefaults setObject:tokenKey forKey:kTokenKey];
        [userDefaults setBool:YES forKey:kHaveCachedToken];
        [userDefaults synchronize];
    }

    // Set the access token on the twitter engine
    // (Why doesn't MGTwitterEngine do this automatically?)
    self.twitterEngine.accessToken = token;
}

- (void)requestSucceeded:(NSString *)connectionIdentifier{
    NSLog(@"Twitter request succeeded: %@", connectionIdentifier);
}

- (void)requestFailed:(NSString *)connectionIdentifier withError:(NSError
↪*)error{
    NSLog(@"Twitter request failed: %@ with error:%@",
    ↪connectionIdentifier, error);

    if ([[error domain] isEqualToString: @"HTTP"])
    {

```

```

switch ([error code]) {

    case 401:
    {
        // Unauthorized. The user's
        ↪credentials failed to verify.
        UIAlertViewQuick(@"Oops!", @"Your
        ↪username and pass-word could not be verified. Double check that you
        ↪entered them correctly and tryagain.", @"OK");
        break;
    }

    case 502:
    {
        // Bad gateway: twitter is down or
        ↪being upgraded.
        UIAlertViewQuick(@"Fail whale!",
        @"Looks like Twitteris down or being updated. Please wait a few seconds and
        try again.", @"OK");
        break;
    }

    case 503:
    {
        // Service unavailable
        UIAlertViewQuick(@"Hold your taps!",
        @"Looks likeTwitter is overloaded. Please wait a few seconds and try
        ↪again.", @"OK");
        break;
    }

    default:
    {
        NSString *errorMessage = [[NSString
        alloc] initWith-Format: @"%d %@", [error code], [error
        localizedDescription]];
        UIAlertViewQuick(@"Twitter error!",
        ↪errorMessage, @"OK");
        [errorMessage release];
        break;
    }
}

}
else
{
    switch ([error code]) {

        case -1009:
        {
            UIAlertViewQuick(@"You're offline!",
            @"Sorry, it lookslike you lost your Internet connection. Please reconnect
            and try again.", @"OK");
        }
    }
}

```

```

                break;
            }

            case -1200:
            {
                UIAlertViewQuick(@"Secure connection
failed", @"I couldn't connect to Twitter. This is most likely a temporary
issue, please try again.", @"OK");
                break;
            }

            default:
            {
                NSString *errorMessage = [[NSString
➔alloc] initWith-
Format:@"%@@ xx %d: %@", [error domain], [error code], [error
➔localizedDescription]];
                UIAlertViewQuick(@"Network Error!",
errorMessage ,@"OK");
                [errorMessage release];
            }
        }
    }
}
}

```

Now we finish all the codes in .h and .m files for the xauthViewController.

Creating Objects in Interface Builder

It's time to create all the objects required in the Interface Builder for the xauthViewController:

1. Click the interface builder file named xauthViewController.xib in Xcode's Project Navigator.
2. Select View, Utilities, Object Library or click on View Right Panel icon at the top right of Xcode to show the Utility panel and select the Object Library icon to open the Object Library, as shown in Figure 24.5.
3. Create a UILabel by dragging a Label from the Object Library to the View within the Interface Builder Editor.
4. Create a UITextField for username by dragging a TextField from the Objective Library to the View. Create another text field for password.

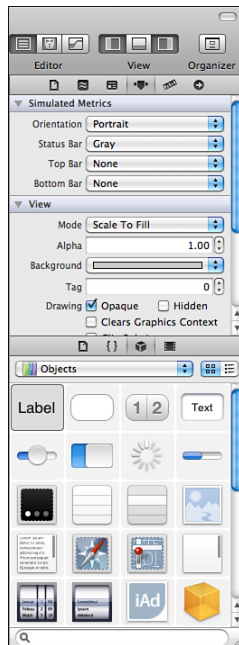


FIGURE 24.5
Selecting the UILabel in Library.

5. Create a UITextView view for the message by dragging a TextView from the Object Library to the View.
6. Create a UIButton by dragging a Button from the Object Library to the View.
7. Select File, Save.
8. Click Run icon to run the project.

The View in Interface Builder is laid out to give sufficient space for the Keyboard to display in the lower half of View. The Keyboard in iPhone OS is 216 pixels high. You can use Size Inspector to guide you on the layout dimension.

Did You Know?

UITextField object is designed for one line only. To display multiple lines, you would have to use UITextView object. UITextView object is filled with random text by default. The default text is longer than Twitter messages, which are restricted to 140 characters. The example is using only 140 characters from the default random text to guide the layout. 140 characters fit well in four lines in the UITextView object.

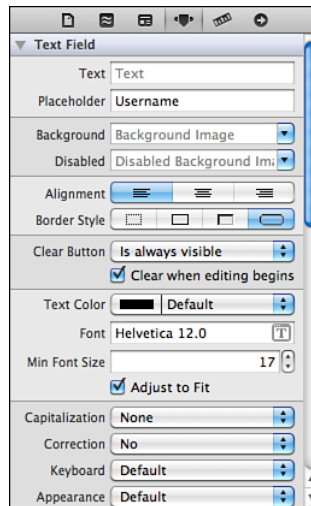
Did You Know?

Defining Object Attributes in Interface Builder

After the objects are created on the View, it's time to define the object attributes:

1. Click `xauthViewController.xib` in the Xcode's Project Navigator to open the file.
2. Select the username UITextField object in the View inside Interface Builder Editor.
3. Select the Attributes icon to open the Attributes Inspector in the Utilities area.
4. In the Attributes Inspector, enter Username in Placeholder, as shown in Figure 24.6.

FIGURE 24.6
Setting the properties for username Text Field in Attributes Inspector.



5. In the Clear Button menu, select Is Always Visible.
6. Click the check box Clear When Editing Begins.
7. In the Correction menu, select No.
8. Select the password UITextField object in the View inside Interface Builder Editor.
9. In the Attributes Inspector, enter Password in Placeholder.
10. In the Clear Button menu, select Is Always Invisible.

11. Click the check box Clear When Editing Begins.
12. In the Correction menu, select No.
13. Click the check box Secure.
14. Select the message UITextView object in the View within Interface Builder Editor.
15. Click the check box Editable.
16. Delete all the characters longer than 140 characters in the Text box. Adjust the TextView object size by dragging the object handlers until it is 300 pixels wide and 98 pixels high. Now remove all the characters in the Text box.
17. In the Correction menu, select No.
18. In the Return Key menu, select Done.
19. In Scrollers of the Scroll View section, remove all the default settings from the Scroller check boxes.
20. Select Button object in the View within Interface Builder Editor.
21. In the Attributes Inspector, enter Post in Title.
22. Save all the changes by selecting File, Save in File menu.

Connecting Objects in Interface Builder

Everything is in place now. It's time to connect the objects in Interface Builder to the methods in Xcode:

1. Select the username UITextField object in the View within Interface Builder Editor, as shown in Figure 24.7.
2. Click the Connections icon to open the Connections Inspector in the Utilities area. In the Connections Inspector, mouse over the circle next to New Referencing Outlet until it appears as a plus (+) sign (see Figure 24.8).
3. Click and hold the mouse down. Drag it outside the circle until you see a blue line. Drag the blue line across the interface builder Editor area to the orange cube File's Owners icon. It is the first icon at the top of the vertical bar in the interface builder dock bar.

FIGURE 24.7
 Selecting
 Username
 UITextField in
 the View in
 Interface
 Builder.

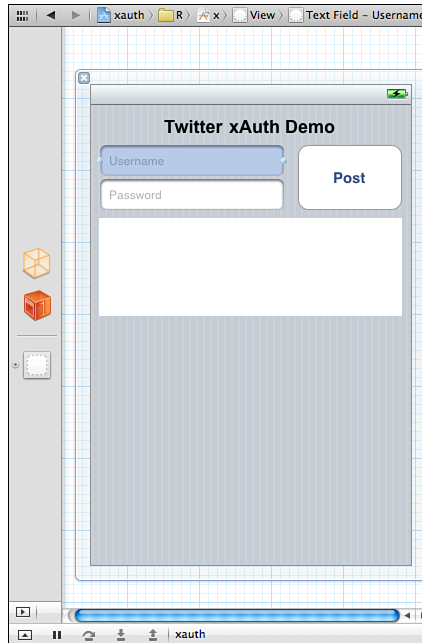
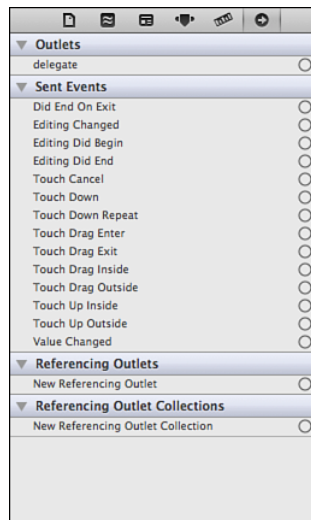


FIGURE 24.8
 Selecting New
 Referencing
 Outlet for Text
 Field in
 Connections
 Inspector.



4. Release the mouse until you see a pop-up menu showing the Text Field objects, as shown in Figure 24.9.
5. Select usernameTextField to connect the username UITextField object. You should see usernameTextField is connected to File's Owner in the Connections Inspector.

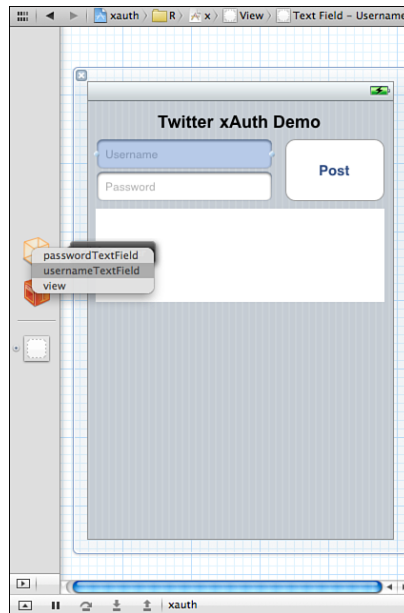
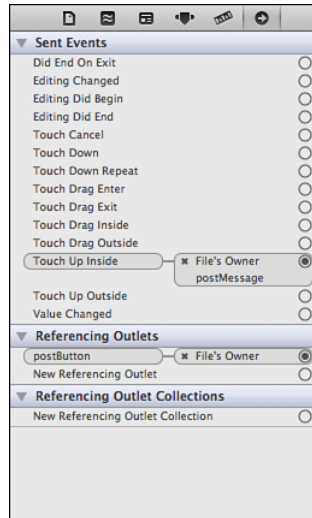


FIGURE 24.9
Selecting
usernameText
Field in File's
Owner.

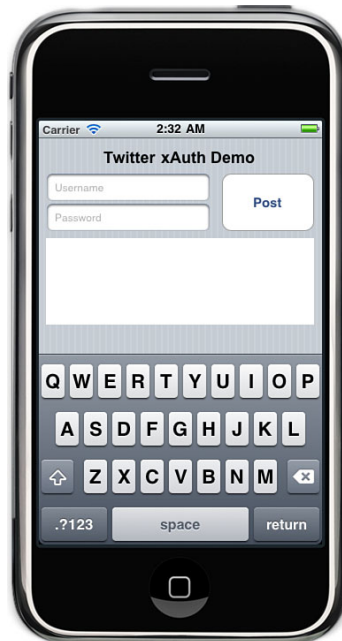
6. Repeat the same for the password UITextField object. Connect it to passwordTextField of the File's Owner icon.
7. Select the UITextView object in the View within Interface Builder Editor.
8. Connect it from Referencing Outlet in Connections Inspector to messageTextView at the File's Owner icon.
9. Select the UIButton object in the View within Interface Builder Editor.
10. Connect it from Referencing Outlet in Connections Inspector to postButton at the File's Owner icon.
11. Select the UIButton object in the View again within the Interface Builder Editor.
12. Connect it from Touch Up Inside under Events in the Connections Inspector to postMessage at the File's Owner icon.
13. Select File, Save in the File menu. If you open the Connections Inspector, you should see that the postButton method is connected to the File's Owner of UIButton object, and the postMessage method is connected to the Touch Up Inside on the Sent Events list, as shown in Figure 24.10.

FIGURE 24.10
Showing the connections of UIButton in the Connections Inspector.



14. Click Run icon to run the project and launch the iPhone Simulator. You should see the app is showing the username, password, message, and post button objects inside the iPhone Simulator, as shown in Figure 24.11.

FIGURE 24.11
Showing the completed app in iPhone Simulator.



15. In the iPhone Simulator, enter a valid Twitter username and password.
16. Enter a text message under 140 characters.
17. Click the Post button. You should see the successful alert message in the app, as shown in Figure 24.12. It indicates that your xAuth is properly implemented and your Twitter account is valid.



FIGURE 24.12 Showing the Successful message in Alert view on the iPhone Simulator.

18. Log in to your Twitter account to view the Send Tweet submitted from the iPhone Simulator. Bask in the glow of your success.

Summary

Congratulations! You've added Twitter Objective-C library, JSON Objective-C library, and OAuth Objective-C library to your iPhone xAuth project. You've created an iPhone xAuth application using your activated xAuth Consumer key and Consumer secret. Last, you have posted a Tweet from your iPhone application in the iPhone Simulator.

Q&A

- Q.** *Why is xAuth the preferred authentication method in the iOS application?*
- A.** xAuth library authenticates the iOS application without requiring users to visit the Twitter OAuth page. It is considered a better user experience.
- Q.** *Why is the xAuth token stored in the Keychain?*
- A.** It is a better security practice to store the xAuth token in the Keychain than in user defaults.

Workshop

Quiz

- Which parsing method(s) are integrated in processing Twitter resulting responses in MGTwitterEngine library?
 - XML
 - JSON
 - Both
- What must be included in the iOS application to request an xAuth token?
 - Consumer key
 - Consumer secret
 - Both
- What Inspector do you use to connect the interface builder object to the File's Owner?
 - Attributes Inspector
 - Size Inspector
 - Connection Inspector
 - Identity Inspector

Quiz Answers

1. C. libxml2.dylib framework is added to parse Twitter XML results, and TouchJSON is included to parse Twitter JSON results.
2. C. Twitter requires both the Consumer key and Consumer secret to generate xAuth access tokens.
3. C. The Connection Inspector provides the interface to connect the interface builder object in the view to the File's Owner.

Exercises

1. Create an xAuth Xcode project using the View-based template.
2. Add the entire MGTwitterDemo library into your iPhone Xcode project.
3. Add the application's Consumer key and Consumer secret into your iPhone application.
4. Complete an iPhone application to send a Tweet using xAuth.
5. Add another View Controller to show your Twitter timeline feed.

This page intentionally left blank

Index

Symbols

- # (hashtag), 2, 8
- #pragma mark, 301
- \$ (dollar sign), 23
- @ (at symbol), 258
- + (plus symbol), 258
- 200 OK response code, 242
- 304 Not Modified response code, 230
- 400 Bad Request response code, 230
- 401 Unauthorized response code, 230
- 403 Forbidden response code, 230
- 404 Not Found response code, 230
- 406 Not Acceptable response code, 230
- 420 Enhance Your Calm response code, 230, 235
- 500 Internal Server Error response code, 230
- 502 Bad Gateway response code, 230

- 503 Service Unavailable response code, 230

A

- accessing other user information, 155
- accessTokenReceived: method, 305-308
- account methods
 - account/end_session, 201
 - account/update_profile, 201
 - account/update_profile_background_image, 201
 - account/update_profile_colors, 201
 - account/update_profile_image, 201
- adding to applications
 - header.inc, 199-200
 - parseTwitter.php, 200
 - twitteroauth.php, 200
- list of, 199

account/end_session method

account/end_session method, 201

accounts

- private accounts, 2
- rate limiting, 11, 18
- setting up, 12-15
- whitelisting, 11-12, 31

account/update_profile method, 201

account/update_profile_background_image method, 201

account/update_profile_colors method, 201

account/update_profile_image method, 201

active SDK, targeting, 285

ADT (Android Development Tools) plug-in, 242, 252

Android applications, 241-242

- ADT (Android Development Tools) plug-in, 242, 252
- Android OAuth application, 255-261
 - creating, 256
 - intent filters and permission, 259-260
 - layout, 257-259
 - Twitter Java libraries, 256
 - XML resources, 261

AVD (Android Virtual Device), creating, 242

development environments, 252

Hello Android project

AndroidManifest.xml, 248-249

AVD (Android Virtual Device), 250

creating, 243-244

helloandroid.java, 245-246

launching, 246-248

SDK issues, 250-251

supported API levels, 249-250

views, 246

importing packages, 261-275

adding OAuth, 262-264

authenticating application, 264-266

responding after authentication, 266-269

TwitterOAuth.java, 269-275

OAuth-signpost, 251

supported operating systems, 252

Twitter4J library, 251

xAuth, 275-276

Android Development Tools (ADT) plug-in, 242, 252

Android OAuth application

importing packages

adding OAuth, 262-264

authenticating application, 264-266

responding after authentication, 266-269

TwitterOAuth.java, 269-275

Twitter Java libraries, 256

Android SDK, downloading, 241

Android Virtual Device (AVD), 242, 250

android:layout_marginBottom attribute, 258

android:layout_width attribute, 258

AndroidManifest.xml, 248-249

@Anywhere and Tweet Button (dev.twitter.com), 215

Apache, 34

API calls

account methods, 199

block methods, 198

blocks/blocking/ids, 198

blocks/create, 198

blocks/destroy, 198

blocks/exists, 198

blocks/unblocking, 198

catching API requests, 232-233

creating

cURL, 53-55, 58

in PHP, 53-57

for Twitter function calls, 75-78

user_timeline API, 55-57

in XML, 49-52

DELETE calls, 136-137

destroy, 132-133

direct message methods, 109-110

favorites API methods. See favorites

Friendships methods

explained, 193

supporting in applications, 194-197

geo/search, 187-190

GET calls, 136-137

list calls, 135-136

List Members resources, 143

List resources, 142

List Subscribers resources, 143-144

call_timeline() function

notification methods, 197

parameters

- explained, 73-75
- multiple parameters, 80

POST calls, 136-137

retweet methods

- id/retweeted_by, 122
- retweet, 119-123
- retweeted_by_me, 118
- retweeted_of_me, 118-119
- retweeted_to_me, 118-119
- retweets/id, 122

search. See Search

streaming methods, 222-226

Trends. See trending topics

types of, 113-114

user API methods

- accessing other user information, 155
- adding to applications, 153-154
- list of, 153
- statuses/followers, 156
- statuses/friends, 156
- thumbnail viewer, creating, 156-158
- users/profile_image, 156
- users/search, 155
- users/show, 153-154
- users/suggestions, 155
- users/suggestions/:slug, 155-156

API levels (Android), 249-250

API parameters, 73-75

\$api_call variable, 98

\$api_url variable, 53

apiwiki.twitter.com, 84, 211

applications, 25

- mashups, 25-27
- platforms, 30-31
- pure chat applications, 29
- registering, 15-16, 82-83
- structured displays, 29
- Twitter clients. See Twitter clients, creating
- Twitter statistics, 29-30
- widgets, 25-26

askOAuth() method, 264-266

at symbol (@), 258

attributes, 237

authenticating Android applications, 264-266

Authentication section (dev.twitter.com), 213

AVD (Android Virtual Device), 242, 250

B

base.js, 107-110

- direct message support, 128
- favorites support, 149, 152
- Friendships methods support, 195
- list support, 138
- Retweet button support, 122
- search support, 169-170

Basic Authentication, 255**#blamedrewscancer, 5**

block methods, 198

- blocks/blocking/ids, 198
- blocks/create, 198
- blocks/exists, 198
- blocks/unblocking, 198

blocking users, 198

blocks/blocking/ids method, 198

blocks/destroy method, 198

blocks/exists method, 198

blocks/unblocking method, 198

bots, 24-25

BreakingNews, 21

browsers

- Chrome, 42
- Firefox, 41
- Internet Explorer, 42

business logic, 27

buttons

- Retweet, 122
 - base.js, 122
 - id/retweeted_by API call, 122
 - id/retweeted_by/ids API call, 123
 - main.css, 122
 - parseTwitter.php, 121
 - render.php, 121
 - sendMessages.php, 120
 - twitteroauth.php, 120
- Tweet,
 - adding with iframe, 210
 - adding with JavaScript, 210
 - customizing, 210-211

C

call_timeline() function, 101

call_direct() function, 101, 140

call_search() function, 167

call_showList() function, 140

call_timeline() function, 97

call_trends() function, 178-179

call_trends_daily() function

call_trends_daily() function, 183

call_users() function, 157

**callback attribute (search),
162, 171**

callPage() function, 107-108

calls (API). See API calls

callTwitter() function, 63-64, 75

capabilities of Twitter, 1-2

catching API requests, 232-233

character limit for tweets, 8

chatters, 22-23

choosing passwords, 12-13

Chrome, 42

classes

advantages of, 82

class files, storing, 93

explained, 81-82

OAuth

adding functions to, 90-92

creating, 83

twitterOAuth

creating, 85-87

`getUserTimeline()`
function, 90-92

`oauth_index.php`, 87-88

Twitter connection errors,
handling, 92-93

`twitteroauth.php`, 88-92

when to use, 93

**clients (Twitter), creating, 16-18,
27-28, 59-60**

Android applications,
241-242

ADT (Android Development
Tools) plug-in, 242

Android OAuth application,
255-261

AVD (Android Virtual
Device), 242

development

environments, 252

Hello Android project,
243-251

importing packages,
261-275

supported operating
systems, 252

xAuth, 275-276

API calls. See API calls

application architecture
diagram, 231-232

block methods, 198

blocks, 198

catching API requests,
232-233

direct messages

adding API support for,
101-102, 109-110,
127-130, 131-132

`call_direct()` function, 101

`callPage()` function,
107-108

deleting, 132-133

destroy API call, 132-133

`direct()` function, 128

`friendshipExists()` function,
129-130

`getMessages()` function,
101, 131

`renderTweets()` function,
128

sanitizing, 110

`sendMessage()` function,
107-108

testing, 126-127

UI elements, adding,
125-126

favorites

adding to applications,
148-149

`createFavorite()` function,
150

creating, 149

definition of, 147

`destroyFavorites()`
function, 152

destroying, 152

`favorite()` function, 149

`showFavorites()` function,
148

Friendships methods

explained, 193

supporting in applications,
194-197

home timeline, 97-99

HTTP response codes, 65-66

`index.php`, 60-61, 69, 95-96

input text fields

`base.js`, 107-108, 110

`createMessage.php`, 106

`index.php`, 105-106

`main.css`, 106-107

`sendMessage.php`, 108

iOS. See iOS

lists

API support for, 135-136

creating, 137-141

definition of, 135

List Members resources,
143

List resources, 142

List Subscribers

resources, 143-144

`main.css`, 61-63

development environments

- main.php, 63, 69-70
 - mentions
 - adding support for, 99-101
 - call_timeline() function, 101
 - getMentions() function, 99-101
 - notifications
 - disabling, 197
 - enabling, 197
 - parseTwitter.php, 66-67
 - render.php, 67-71
 - retrying if Twitter is down, 233-236
 - Retweet button
 - base.js, 122
 - id/retweeted_by API call, 122
 - id/retweeted_by/ids API call, 123
 - main.css, 122
 - parseTwitter.php, 121
 - render.php, 121
 - retweets/id API call, 122
 - sendMessage.php, 120
 - twitteroauth.php, 120
 - Search. *See* Search
 - streaming, 219
 - advantages of, 226
 - limits on, 221
 - pre-launch checklist, 221-222
 - site streams, 220
 - Streaming API, 219
 - streaming methods, 222-226
 - user streams, 219-220
 - when to use, 220
 - tabs. *See* tabs
 - trending topics
 - call_trends() function, 178-179
 - definition of, 166-167
 - recent, daily, and weekly trends, 180-185
 - showTrends() function, 178-179
 - supporting in applications, 177-180
 - Tweet button, 209
 - adding with iframe, 210
 - adding with JavaScript, 210
 - customizing, 210-211
 - Twitter clients, creating
 - ADT (Android Development Tools) plug-in, 252
 - getMessagesSent() function, 132
 - twitterAPI.php, 64-65, 70
 - colors (profile), 201-203
 - commandLine.php
 - favorites support, 150
 - Friendships methods support, 195
 - list support, 138-139
 - commands, \$ _GET, 95
 - config.php, 85
 - configuring
 - accounts, 12-15
 - local web servers, 34-38
 - connection errors, handling, 92-93
 - console page (dev.twitter.com), 207
 - Consumer key, retrieving, 284-285
 - Consumer secret, retrieving, 284-285
 - Count parameter, 74
 - count parameter (getMentions() function), 99
 - create_message.php, 168-169
 - createFavorite() function, 150
 - createList() function, 139
 - createListItem() function, 138
 - createMessage.php, 106
 - cURL, 53-55, 58
 - \$curl_handle variable, 53, 64
 - curl_setopt() function, 92
 - currentTimeMillis() method, 267
 - customizing Tweet button, 210-211
- ## D
- daily trends, 180-185
 - call_trends_daily() function, 183
 - header.inc, 183
 - parseTwitter.php, 183-184
 - showTrends_daily() function, 184
 - twitteroauth.php, 184-185
 - declaring
 - properties/methods, 300
 - variables, 300
 - DELETE calls, 136-137
 - deleting direct messages, 132-133
 - destroy API call, 132-133
 - destroyFavorites() function, 152
 - destroying favorites, 152
 - development environments
 - for Android, 252
 - LAMP stacks
 - explained, 33-34

development environments

- popularity of, 46
- XAMPP, 35-38
- local web servers, configuring, 34-38
- tools
 - Chrome, 42
 - Firebug, 41-42
 - Firefox, 41
 - IDEs (integrated development environments), 43-44
 - Internet Explorer, 42
 - phpMyAdmin, 42
 - recommended toolbox, 45
 - revision control systems, 44-45
 - text editors, 43
- web server security, 38-41
 - MySQL, 40-41
 - phpMyAdmin, 41
 - XAMPP pages, 40
 - XAMPP security console, 39-40
- development tools**
 - Chrome, 42
 - Firebug, 41-42
 - Firefox, 41
 - IDEs (integrated development environments), 43-44
 - Internet Explorer, 42
 - phpMyAdmin, 42
 - recommended toolbox, 45
 - revision control systems, 44-45
 - text editors, 43
- dev.twitter.com website, 211**
 - @Anywhere and Tweet Button, 212
- Authentication, 212
- console page, 207
- Ecosystem, 216
- Guidelines and Terms, 213
- REST API and General, 214
- start page, 206
- Streaming API Documentation and Search API, 215
- direct() function, 128**
- direct messages, 125**
 - adding API support for, 101-102, 109-110
 - base.js, 128
 - header.inc, 132
 - parseTwitter.php, 131-132
 - render.php, 127-128
 - sendMessage.php, 129
 - twitteroauth.php, 129-132
 - call_direct() function, 101
 - callPage() function, 107-108
 - deleting, 132-133
 - destroy API call, 132-133
 - direct() function, 128
 - friendshipExists() function, 129-130
 - getMessages() function, 101, 131
 - getMessagesSent() function, 132
 - input text fields. See input text fields
 - renderTweets() function, 128
 - sanitizing, 110
 - sending message to Twitter, 108-109
 - sendMessage() function, 107-108
- testing, 126-127
- UI elements, adding
 - header.inc, 125-126
 - index.php, 125
- direct messages (DMs), 2**
- directives**
 - #pragma mark, 301
 - @interface, 300
- disabling notifications, 197**
- Display Guidelines (dev.twitter.com), 213**
- DMs (direct messages), 2**
- documentation**
 - apiwiki.twitter.com, 211
 - dev.twitter.com website, 211
 - @Anywhere and Tweet Button, 212
 - Authentication, 212
 - console page, 207
 - Ecosystem, 216
 - Guidelines and Terms, 213
 - REST API and General, 214
 - start page, 206
 - Streaming API Documentation and Search API, 215
- dollar sign (\$), 23**
- Dorsey, Jack, 2-3**
- downloading**
 - Android SDK, 241
 - MGTwitterEngine library, 285
 - MGTwitterEngineDemo, 285
 - Oauth-signpost, 251
 - Twitter4J library, 251

E**Eclipse, 43-44, 241****Ecosystem section
(dev.twitter.com), 216****editing**

index.php, 105-106

twitteroauth.php, 90-92

editors (text), 43**Egyptian revolution, tweets sent
during, 5-7****enabling notifications, 197****ending sessions, 201****errors, Twitter connection errors,
92-93****exceptions**

definition of, 92

Twitter connection errors,
handling, 92**F****Facebook, compared to Twitter, 4****FailWhale, 229-230****favorite() function, 149****favorites**

adding to applications, 148

base.js, 149

header.inc, 148

parseTwitter.php, 148

twitteroauth.php, 148

createFavorite() function, 150

creating, 149

base.js, 152

commandLine.php, 150

render.php, 149-152

twitteroauth.php,
150, 152

definition of, 147

destroyFavorites() function,
152

destroying, 152

favorite() function, 149

showFavorites() function, 148

fields, input text fields

base.js, 107-110

createMessage.php, 106

index.php, 105-106

main.css, 106-107

sendMessage.php, 108

files. See also specific files

class files, storing, 93

organizing, 72

Firebug, 41-42**Firefox, 41****follow() function, 195****followers, 2****following, 2****friendshipExists() function,
129-130, 141****Friendships methods**

explained, 193

supporting in applications

base.js, 195

commandLine.php, 195

parseTwitter.php, 196-197

render.php, 194

twitteroauth.php, 195

functions. See also API callsadding to twitterOAuth class,
90-92

askOAuth(), 264-266

call_timeline(), 101

call_direct(), 101, 140

call_search(), 167

call_showList(), 140

call_timeline(), 97

call_trends(), 178

call_trends_daily(), 183

call_users(), 157

callPage(), 107-108

callTwitter(), 63-64, 75

createList(), 139

createListItem(), 138

curl_setopt(), 92

currentTimeMillis(), 267

direct(), 128

favorite(), 149

follow(), 195

friendshipExists(), 129-130,
141

getData(), 266

getHomeTimeline(), 89

getMentions(), 233

code listing, 100

parameters, 99-100

getMessages(), 101, 131

getMessagesSent(), 132

getPublicTimeline(), 78-79

getQueryParameter(), 266

getRTByMe(). *See also* API
callsgetRTOfMe(). *See also* API
callsgetRTToMe(). *See also* API
calls

getTwitterData(), 97

getUserRate(), 200

getUserTimeline(), 75-78,
90-92

functions

htmlentities(), 54
 leave(), 195
 makeText(), 265, 267
 OAuthRequest(), 88
 onCreate(), 261
 onNewIntent(), 266
 parseTwitter(), 70
 parseTwitterReply(), 63
 parseTwitterReply (), 66
 phpinfo(), 40
 postMessage(), 304-305
 renderLists(), 141
 renderTweets(), 63, 67, 128
 responsefromServer(), 108
 search(), 167-168
 sendMessage(), 107-108
 sendSearch(), 169
 setContentView(), 261
 setText(), 267
 setVisibility(), 268
 showFollowers(), 158
 showFriends(), 158
 showgeo_search(), 232
 showLists(), 141
 showTrends(), 178-179
 showTrends_current(), 184
 showTrends_daily(), 184
 showTrends_weekly(), 184
 showUser(), 154
 SimpleXMLElement(), 97
 toLocaleString(), 267
 updateStatus(), 108, 267

future of Twitter API, 236-237

G

Geo Developer Guidelines
 (dev.twitter.com), 213
GEO tag, 187-190
 geocode attribute (search), 163,
 165, 172
 geo/search API call, 187-190
GET calls, 136-137
\$_GET command, 95
Get xAuth Access Token button,
288
get_public_timeline.php, 53
getData() method, 266
getHomeTimeline() function, 89
getMentions() function, 233
 code listing, 100
 parameters, 99-100
getMessages() function, 101, 131
getMessagesSent() function, 132
getPublicTimeline() function,
78-79
getQueryParameter() method,
266
getRTByMe() function, 119
getRTOFMe() function. See also
 API calls
getRTToMe() function. See also
 API calls
getTwitterData() function, 97
getUserRate() function, 200
getUsersTimeline() function,
90-92
getUserTimeline() function, 75-78
getXAuthAccessTokenForUsername:
password: method, 304
Git, 44
Global System for Mobile
Communications (GSM), 2-3

Google Chrome, 42

GSM (Global System for Mobile
Communications), 2-3

Guidelines and Terms section
 (dev.twitter.com), 213

H

hashtag, 2, 8

header.inc, 114-115

account methods, 199-200
 direct message support,
 125-126, 132
 favorites support, 148
 list support, 139
 recent, daily, and weekly
 trends, 183
 search support, 166
 thumbnail viewer, creating,
 156-157
 trending topics support, 178
 user API methods, 154

Hello Android project

AndroidManifest.xml,
 248-249
 AVD (Android Virtual Device),
 250
 creating, 243-244
 helloandroid.java, 245-246
 launching, 246-248
 OAuth-signpost, 251
 SDK issues, 250-251
 supported API levels,
 249-250
 Twitter4J library, 251
 views, 246

- Hello World application (iOS), 280-283
 - helloandroid.java, 245-246
 - high-frequency users, 24
 - history of Twitter, 2-3
 - home page (Twitter), 16-18
 - home timeline, creating, 97-99
 - htmlentities() function, 54
 - HTTP response codes
 - catching, 232-236
 - creating, 65-66
 - supported codes, 230
 - Hypertext Coffee Pot Control Protocol, 230
- I**
- \$i counter, 97
 - ID parameter, 74
 - IDEs (integrated development environments), 43-44
 - id/retweeted_by API call, 122
 - id/retweeted_by/ids API call, 123
 - iframe, adding Tweet button with, 210
 - images, profile images, 201
 - importing
 - libraries
 - to header files, 299-300
 - to implementation files, 301
 - packages, 261-275
 - adding OAuth, 262-264
 - authenticating application, 264-266
 - responding after authentication, 266-269
 - TwitterOAuth.java, 269-275
 - include_entities parameter (getMentions() function), 100
 - include_rtf parameter (getMentions() function), 100
 - index.php, 95-96, 105-106
 - creating, 60-61, 69
 - direct message support, 125
 - expanding to support tabs, 117
 - initializing MGTwitterEngine library, 302
 - initWithCoder: method, 302
 - input text fields
 - base.js, 107-110
 - createMessage.php, 106
 - index.php, 105-106
 - main.css, 106-107
 - sendMessage.php, 108
 - installing XAMPP, 35-38
 - on Linux, 37-38
 - on Mac OS, 37
 - troubleshooting installation, 38
 - on Windows, 35-37
 - integrated development environments (IDEs), 43-44
 - intent filters, adding to Android OAuth application, 259-260
 - IntentFilter objects, 259-260
 - Interface Builder
 - connecting objects in, 309-311
 - creating objects in, 308-315
 - defining object attributes in, 309-311
 - @interface directive, 300
 - Internet Explorer, 42
 - iOS, 279
 - active SDK, targeting, 285
 - Consumer key and Consumer secret, retrieving, 284-285
 - Hello World application, 280-283
 - memory management, 301-302
 - MGTwitterEngine library
 - Delegate methods, 305-308
 - downloading, 285
 - initializing, 302
 - objects, creating in Interface Builder, 308-315
 - #pragma mark, 301
 - tweets, posting, 304-305
 - ViewController.h, 299
 - declaring properties and methods, 300
 - declaring variable instances, 300
 - importing libraries to header files, 299-300
 - ViewController.m, 300-301
 - xAuth
 - advantages of, 294
 - creating xAuth application, 294-299
 - definition of, 293-294
 - explained, 284
 - loading xAuth tokens, 302-304
 - requesting, 284
 - Twitter application for xAuth request, 283
 - verifying, 286-289

iPhone platform

iPhone platform, 30-31. *See also* iOS
 iPod Touch platforms. *See* iOS

J-K

Java libraries, 256
 JavaScript, adding Tweet button with, 210
 JSON, parsing, 166-167
 JTwitter library, 256
 Krikorian, Raffi, 185

L

LAMP stacks
 explained, 33-34
 popularity of, 46
 XAMPP, 35
 installing, 35-38
 security console, 39-40
lang attribute (search), 162, 165, 171
launching Hello Android project, 246-248
layout of Android OAuth application, 257-259
leave() function, 195
libraries
 cURL, 53-55, 58
 importing to implementation files, 301
 MGTwitterEngine, initializing, 302

MGTwitterEngine library
 Delegate methods, 305-308
 downloading, 285
 OAuth-signpost, 251
 Twitter Java libraries, 256
 Twitter4J, 251
limits on Twitter use, 11, 18
Linux, 33
 XAMPP installation, 37-38
 XAMPP security console, 40
List Members resources, 143
List Subscribers resources, 143-144
list.php, 137-138
lists
 API support for, 135-136
 List Members resources, 143
 List resources, 142
 List Subscribers resources, 143-144
 creating
 base.js, 138
 call_direct() function, 140
 call_showList() function, 140
 commandLine.php, 138-139
 createList() function, 139
 createListItem() function, 138
 friendshipExists() function, 141
 header.inc, 139
 list.php, 137-138
 parseTwitter.php, 140-141
 renderLists() function, 141
 showLists() function, 141
 twitteroauth.php, 139, 141
 definition of, 135
loading
 xAuth tokens, 302-304
 XML resources, 261
local web servers, configuring, 34-38
locale attribute (search), 162, 171
logic, business logic, 27
Lu, Yiyang, 229

M

Mac OS. See also iOS
 XAMPP installation, 37
 XAMPP security console, 40
main.css, 106-107
 creating, 61-63
 Retweet button support, 122
main.php, creating, 63, 69-70
makeText() method, 265, 267
mashups, 25-27
max_id parameter (getMentions() function), 99
memory management, iOS, 301-302
mentions, adding support for, 99-101
messages
 compared to statuses, 58
 direct messages, 125
 adding API support for, 101-102, 109-110, 127-130, 131-132

- call_direct() function, 101
- callPage() function, 107-108
- deleting, 132-133
- destroy API call, 132-133
- direct() function, 128
- friendshipExists() function, 129-130
- getMessages() function, 101, 131
- renderTweets() function, 128
- sanitizing, 110
- sendMessage() function, 107-108
- testing whether messages can be sent, 126-127
- UI elements, adding, 125-126
- getMessagesSent() function, 132
- sending message to Twitter, 108-109
- metadata mode (Search), 173**
- methods. See specific methods**
- MGTwitterEngine library**
 - Delegate methods, 305-308
 - downloading, 285
 - initializing, 302
- MGTwitterEngineDemo, 285**
- MGTwitterEngineDemoViewControlller.h, 280-283**
- microbloggers, 24**
- mobile platforms, 30-31**
- Mubarek, Muhammed Hosni Sayed, 5-7**
- multiple parameters, 80**
- MySQL, 34, 40-41**

N

- NAT (Network Address Translation), 38**
- navs.cc, 115-117**
- Netbeans, 44**
- Network Address Translation (NAT), 38**
- new users, 24**
- news readers, 21-22**
- NewsSnacker, 21-22**
- Notepad++, 43**
- notification methods, 197**
- notifications**
 - disabling, 197
 - enabling, 197
- notifications/follow method, 164**
- notifications/leave method, 197**

O

- OAuth, 255-261**
 - adding, 262-264
 - Android OAuth application
 - creating, 256
 - intent filters and permission, 259-260
 - layout, 257-259
 - Twitter Java libraries, 256
 - XML resources, 261
 - definition of, 82
 - flow overview, 84
 - OAuth class, creating, 83
 - Twitter connection errors, handling, 92-93

- twitterOAuth class
 - adding functions to, 90-92
 - creating, 85-87
 - getUserTimeline() function, 90-92
 - oauth_index.php, 87-88
 - twitteroauth.php, 88-92
- OAuth class, creating, 83**
- oauth_index.php, 87-88**
- oauthRequest() function, 88**
- OAuth-signpost, 251**
- objects**
 - creating in Interface Builder, 308-315
 - definition of, 81-82
- Odeo, 2-3**
- onCreate() function, 261**
- onNewIntent() method, 266**
- organizing files, 72**

P

- packages, importing, 261-275**
 - adding OAuth, 262-264
 - authenticating application, 264-266
 - responding after authentication, 266-269
 - TwitterOAuth.java, 269-275
- page attribute (search), 163-165, 171**
- Page parameter, 74, 100**
- parameters**
 - explained, 73-75
 - for getMentions() function, 99-100

parseTwitter() function**parseTwitter() function, 70****parseTwitter.php, 118**

account methods, 200

creating, 66-67

direct message support, 132

favorites support, 148

Friendships methods support,
196-197

list support, 140-141

parsing JSON, 166-167

recent, daily, and weekly
trends, 183-184

Retweet button support, 121

search support, 166-167

thumbnail viewer, creating,
157trending topics support,
178-179

user API methods, 154

**parseTwitterReply() function,
63, 66****parseTwitterReply.php, 70****parsing JSON, 166-167****passwords, choosing, 12-13****Perl, 33****permissions, adding to Android
OAuth application, 259-260****PHP, 33-34, 53-57****phpinfo() function, 40****phpMyAdmin, 41-42****platforms, 30-31****plus symbol (+), 258****POST calls, 136-137****posting tweets (iOS), 304-305****postMessage() method, 304-305****power users, 23****PR managers, 23****private accounts, 2****\$profile_image_url variable, 66****profiles**

profile colors, 201-203

profile images, 201

properties, declaring, 300**protocols**Hypertext Coffee Pot Control
Protocol, 230NAT (Network Address
Translation), 38SMS (Short Message
System), 2-3**public relations managers, 23****pure chat applications, 27-28****Python, 33****Q-R****q attribute (search), 171****rate limiting, 11, 18****readers. See clients (Twitter)****recent trends, 180-185**

header.inc, 183

parseTwitter.php, 183-184

twitteroauth.php, 184-185

recommended toolbox, 45**refreshing search results, 173****registering applications, 15-16,
82-83****renderLists() function, 141****render.php**

creating, 67-68, 70-71

direct message support,
127-128

favorites support, 149-152

Friendship methods support,
194

Retweet button support, 121

**renderTweets() function, 63,
67, 128****@reply, 2, 8****requestFailed: method, 305-308****requesting Twitter xAuth, 284****requests (Search), 164-165****requestSucceeded: method,
305-308****responding after authentication,
266-269****response codes (HTTP). See HTTP
response codes****responsefromServer() function,
108****REST API and General section
(dev.twitter.com), 214****result_type attribute (search),
163, 172****retrieving Consumer key and
Consumer secret, 284-285****retrying if Twitter is down,
233-236****retweet API call, 119-123****Retweet button**

base.js, 122

id/retweeted_by API call, 122

id/retweeted_by/ids API call,
123

main.css, 122

parseTwitter.php, 121

render.php, 121

retweets/id API call, 122

sendMessages.php, 120

twitteroauth.php, 120

- retweeted_by_me API call, **118**
 - retweeted_of_me API call, **118-119**
 - retweeted_to_me API call, **118-119**
 - retweets, **2**
 - retweets/id API call, **122**
 - revision control systems, **44-45**
 - roid:layout_height attribute, **258**
 - rpp attribute (search), **163, 165, 171**
 - RTs (retweets), **2**
 - Rules of the Road (dev.twitter.com), **213**
- S**
- Sagolla, Dom, **2-3**
 - sanitizing messages, **110**
 - SCMs (source code management) systems, **44-45**
 - Scoble, Robert, **229**
 - Screen_name parameter, **74**
 - \$screen_name variable, **66**
 - SDK issues (Android), **250-251**
 - Search, **161**
 - adding to applications
 - base.js, **169-170**
 - call_search() function, **167**
 - create_message.php, **168-169**
 - header.inc, **166**
 - parseTwitter.php, **166-167**
 - search() function, **167-168**
 - sendSearch() function, **169**
 - twitteroauth.php, **167-168**
 - metadata mode, **173**
 - refreshing search results, **173**
 - search attributes, **162-164, 170-172**
 - search requests, **164-165**
 - Twitter's stance on, **161-162**
 - usage notes, **172-173**
 - search API method. *See* Search
 - search() function, **167-168**
 - security for web servers, **38-41**
 - MySQL, **40-41**
 - phpMyAdmin, **41**
 - XAMPP pages, **40**
 - XAMPP security console, **39-40**
 - Send Test Tweet button, **288**
 - sending direct messages, **125**
 - API support for, **127-130**
 - testing whether messages can be sent, **126-127**
 - UI elements, adding, **125-126**
 - sendMessage() function, **107-108**
 - sendMessage.php, **108, 129**
 - sendMessages.php, **120**
 - sendSearch() function, **169**
 - sendUpdate: method, **304**
 - servers, web. *See* web servers
 - sessions, ending, **201**
 - setContentView() function, **261**
 - setOAuthAccessToken() method, **268**
 - setText() method, **267**
 - setting up
 - accounts, **12-15**
 - local web servers, **34-38**
 - setVisibility() method, **268**
 - SFHFKeychainUtils, **303**
 - Short Message System), **2-3**
 - shortcuts, **111**
 - show_user attribute (search), **163, 165, 172**
 - showFavorites() function, **148**
 - showFollowers() function, **158**
 - showFriends() function, **158**
 - showgeo_search() function, **232**
 - showLists() function, **141**
 - showTrends() function, **180**
 - showTrends_current() function, **184**
 - showTrends_daily() function, **184**
 - showTrends_weekly() function, **184**
 - showUser() function, **154**
 - Signpost, **251**
 - SimpleXMLElement() function, **97**
 - since_id attribute (search), **163, 165, 171**
 - Since_ID parameter, **74**
 - since_id parameter (getMentions() function), **99**
 - site streams, **220**
 - SMS (Short Message System), **2-3**
 - source code management (SCM) systems, **44-45**
 - spotting the FailWhale, **229-230**
 - statistics (Twitter), **29-30**
 - statuses, compared to messages, **58**
 - statuses/filter method, **223-224**

statuses/firehose method

statuses/firehose method, 224-225
statuses/followers API method, 156
statuses/friends API method, 156
statuses/links method, 225
statusesReceived: method, 305-308
statuses/retweet method, 225
statuses/sample method, 225-226
Stenberg, Daniel, 86
streaming, 219
 advantages of, 226
 limits on, 221
 pre-launch checklist, 221-222
 site streams, 220
 Streaming API, 219
 streaming methods, 222-226
 statuses/filter, 223-224
 statuses/firehose, 224-225
 statuses/links, 225
 statuses/retweet, 225
 statuses/sample, 225-226
 user streams, 219-220
 when to use, 220
Streaming API, 219
Streaming API Documentation and Search API section (dev.twitter.com), 215
structured displays, 29
Subversion (SVN), 44
SVN (Subversion), 44

T

tabs, supporting, 114-117
 header.inc, 114-115
 index.php, 95-96, 117
 navs.cc, 115-117
targeting active SDK, 285
testing direct messages, 126-127
text editors, 43
text fields. *See* input text fields
TextMate, 43
thumbnail viewer, creating
 header.inc, 156-157
 parseTwitter.php, 157
 twitteroauth.php, 158
time zone, displaying, 68
toLocaleString() method, 267
trending topics
 call_trends() function, 178-179
 definition of, 2, 177
 recent, daily, and weekly trends, 180-185
 call_trends_daily() function, 183
 header.inc, 183
 parseTwitter.php, 183-184
 showTrends_current() function, 184
 showTrends_daily() function, 184
 showTrends_weekly() function, 184
 showTrends() function, 178-179
 supporting in applications
 header.inc, 178
 parseTwitter.php, 178-179
 twitteroauth.php, 180
 Trends/available API call, 185-187
 twitteroauth.php, 184-185
trends. *See* trending topics
Trends/available API call, 185-187
trim_user parameter (getMentions() function), 100
troubleshooting
 Twitter HTTP response codes, 65-66
 XAMPP installation, 38
Tweet button, 209
 adding with iframe, 210
 adding with JavaScript, 210
 customizing, 210-211
tweets
 character limit for, 8
 definition of, 2
 use case studies
 #blamedrewscancer, 5
 Egyptian revolution, 5-7
Twitter API, future of, 236-237
Twitter clients, creating, 16-18, 59-60, 199
 Android applications, 241-242
 ADT (Android Development Tools) plug-in, 242, 252
 Android OAuth application, 255-261
 AVD (Android Virtual Device), 242
 development environments, 252
 Hello Android project, 243-251
 importing packages, 261-275

Twitter clients, creating

- supported operating systems, 252
- xAuth, 275-276
- API calls. *See* API calls
- application architecture diagram, 231-232
- block methods, 198
- blocks, 198
- catching API requests, 232-233
- direct messages
 - adding API support for, 101-102, 109-110, 127-130, 131-132
 - call_direct() function, 101
 - callPage() function, 107-108
 - deleting, 132-133
 - destroy API call, 132-133
 - direct() function, 128
 - friendshipExists() function, 129-130
 - getMessages() function, 101, 131
 - getMessagesSent() function, 132
 - renderTweets() function, 128
 - sanitizing, 110
 - sending message to Twitter, 108-109
 - sendMessage() function, 107-108
 - testing, 126-127
 - UI elements, adding, 125-126
- favorites
 - adding to applications, 148-149
 - createFavorite() function, 150
 - creating, 149-152
 - definition of, 147
 - destroyFavorites() function, 152
 - destroying, 152
 - favorite() function, 149
 - showFavorites() function, 148
- Friendships methods
 - explained, 193
 - supporting in applications, 194-197
- home timeline, 97-99
- HTTP response codes, 65-66
- index.php, 60-61, 69, 95-96
- input text fields, 108
 - base.js, 107-108, 110
 - createMessage.php, 106
 - index.php, 105-106
 - main.css, 106-107
 - sendMessage.php, 108
- iOS. *See* iOS
- lists
 - API support for, 135-136
 - creating, 137-141
 - definition of, 135
 - List Members resources, 143
 - List resources, 142
 - List Subscribers resources, 143-144
- main.css, 61-63
- main.php, 63, 69-70
- mentions
 - adding support for, 99-101
 - call_timeline() function, 101
 - getMentions() function, 99-101
- notifications
 - disabling, 197
 - enabling, 197
- parseTwitter.php, 66-67
- render.php, 67-71
- retrying if Twitter is down, 233-236
- Retweet button
 - base.js, 122
 - id/retweeted_by API call, 122
 - id/retweeted_by/ids API call, 123
 - main.css, 122
 - parseTwitter.php, 121
 - render.php, 121
 - retweets/id API call, 122
 - sendMessages.php, 120
 - twitteroauth.php, 120
- Search. *See* Search
- streaming, 219
 - advantages of, 226
 - limits on, 221
 - pre-launch checklist, 221-222
 - site streams, 220
 - Streaming API, 219
 - streaming methods, 222-226
 - user streams, 219-220
 - when to use, 220
- tabs. *See* tabs

Twitter clients, creating

trending topics
 call_trends() function, 178-179
 definition of, 166-167
 recent, daily, and weekly trends, 180-185
 showTrends() function, 178-179
 supporting in applications, 177-180

Tweet button, 209
 adding with iframe, 210
 adding with JavaScript, 210
 customizing, 210-211
 twitterAPI.php, 64-65, 70

Twitter statistics, 29-30

Twitter xAuth. *See* xAuth

Twitter4J library, 251, 256

twitterAPI.php, creating, 64-65, 70

TwitterHolics, 24

twitterOAuth class

adding functions to, 90-92
 creating, 85-87
 getUserTimeline() function, 90-92
 oauth_index.php, 87-88
 Twitter connection errors, handling, 92-93
 twitteroauth.php, 88-92

TwitterOAuth.java, 269-275

twitteroauth.php, 88-92, 119

account methods, 200
 direct message support, 129-132
 favorites support, 148, 150, 152
 Friendships methods support, 195

list support, 139, 141
 recent, daily, and weekly trends, 184-185
 retrying if Twitter is down, 233-236
 Retweet button support, 120
 search support, 167-168
 thumbnail viewer, creating, 158
 trending topics support, 180
 user API methods, 154

\$twitterResponseData variable, 64

TwittFilter, 29-30

Twurl Web Console, 206

U

UITextField object, 309

unblocking users, 198

until attribute (search), 163, 171

\$update variable, 66

updateStatus() function, 109, 267

\$updateTime variable, 66

updating profile images, 201

URL shortening, 215

URLs

Twitter URLs, 12
 vanity URLs, 14, 18

user API methods

accessing other user information, 155
 adding to applications, 154
 header.inc, 154
 twitteroauth.php, 154
 list of, 153
 statuses/followers, 156

statuses/friends, 156
 thumbnail viewer, creating, 156-158
 header.inc, 156-157
 parseTwitter.php, 157
 twitteroauth.php, 158
 users/profile_image, 156
 users/search, 155
 users/show, 153-154
 users/suggestions, 155
 users/suggestions/:slug, 155-156

user streams, 219-220

User_ID parameter, 74

user_timeline API, 55-57

:user/:list_id/ subscribers /:id method, 144

:user/:list_id/create_all method, 143

:user/:list_id/members method, 143

:user/:list_id/subscribers method, 144

:user/:list_id/subscribers/:id method, 144

:user/:lists method, 142

:user/lists/:id method, 142

:user/lists/:id/statuses method, 142

:user/lists/memberships method, 142

:user/lists/subscriptions method, 142

users, 21

blocking, 198
 bots, 24-25
 chatters, 22-23
 high-frequency users, 24
 microbloggers, 24

- new users, 24
- news readers, 21-22
- power users, 23
- PR managers, 23
- unblocking, 198
- users/profile_image API method, 156**
- users/search API method, 155**
- users/suggestions API method, 155**
- users/suggestions/:slug API method, 155-156**

V

- vanity URLs, 14, 18**
- variables, declaring, 300. See also specific variables**
- verifying xAuth, 286-289**
- ViewController.h, 299**
 - declaring properties and methods, 300
 - declaring variable instances, 300
 - importing libraries to header files, 299-300
- ViewController.m, 300-301**
- viewDidLoad method, 302-304**
- views, Hello Android project, 246**
- Vim, 43**

W

- web interface, 237**
- web servers**
 - configuring, 34-38
 - security, 38-41
 - MySQL, 40-41
 - phpMyAdmin, 41
 - XAMPP pages, 40
 - XAMPP security console, 39-40
- websites, dev.twitter.com, 205**
- weekly trends, 180-185**
 - header.inc, 183
 - parseTwitter.php, 183-184
 - showTrends_weekly() function, 184
 - twitteroauth.php, 184-185
- Where On Earth ID (WOEID), 185-186**
- whitelisting, 11-12, 31**
- widgets, 25-26, 237**
- Williams, Abraham, 83**
- Windows**
 - XAMPP installation, 35-37
 - XAMPP security console, 39-40
- WOEID (Where On Earth ID), 185-186**

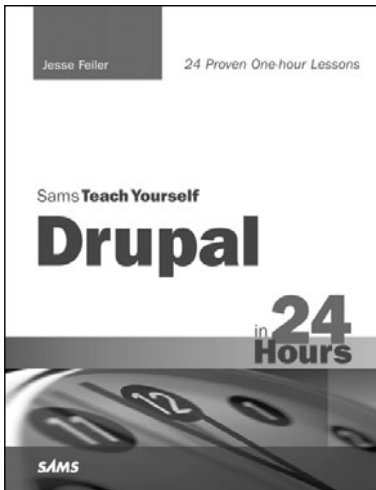
X-Y-Z

- XAMPP, 35**
 - installing, 35-38
 - on Linux, 37-38

- on Mac OS, 37
 - troubleshooting installation, 38
 - on Windows, 35-37
 - security console, 39-40
- xAuth, 275-276**
 - advantages of, 294
 - creating xAuth application, 294-299
 - definition of, 290
 - explained, 284
 - loading xAuth tokens, 302-304
 - requesting Twitter xAuth, 284
 - Twitter application for xAuth request, 283
 - verifying, 286-289
 - ViewController.h, 299
 - declaring properties and methods, 300
 - declaring variable instances, 300
 - importing libraries to header files, 299-300
 - ViewController.m, 300-301
 - xauthViewController.xib, 308-315
- xauthViewController.m, 300-301**
- xauthViewController.xib, 308-315**
- Xcode, 280-283**
- XML**
 - API calls, creating, 49-52
 - XML resources (Android OAuth application), 261

Sams **Teach Yourself**

When you only have time
for the answers™



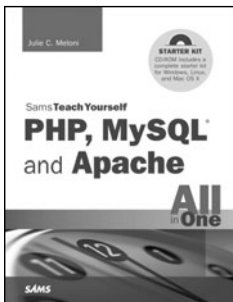
Whatever your need and whatever your time frame, there's a Sams **Teach Yourself** book for you. With a Sams **Teach Yourself** book as your guide, you can quickly get up to speed on just about any new product or technology—in the absolute shortest period of time possible. Guaranteed.

Learning how to do new things with your computer shouldn't be tedious or time-consuming. Sams **Teach Yourself** makes learning anything quick, easy, and even a little bit fun.

Drupal in 24 Hours

Jesse Feiler

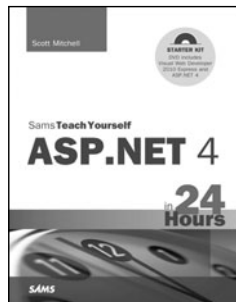
ISBN-13: 978-0-672-33126-8



PHP, MySQL and Apache All in One

Julie C. Meloni

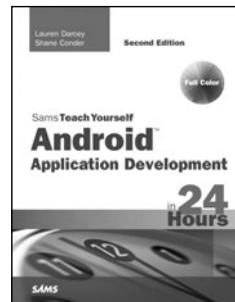
ISBN-13: 978-0-672-33543-3



ASP.NET 4 in 24 Hours

Scott Mitchell

ISBN-13: 978-0-672-33305-7

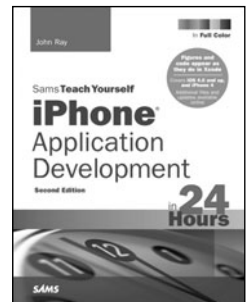


Android Application Development in 24 Hours, Second Edition

Lauren Darcey

Shane Conder

ISBN-13: 978-0-672-33569-3



iPhone Application Development in 24 Hours

John Ray

ISBN-13: 978-0-672-33220-3

Sams Teach Yourself books are available at most retail and online bookstores. For more information or to order direct, visit our online bookstore at informit.com/sams.

Online editions of all Sams Teach Yourself titles are available by subscription from Safari Books Online at safari.informit.com.

Try Safari Books Online FREE

Get online access to 5,000+ Books and Videos



Safari[®]
Books Online

FREE TRIAL—GET STARTED TODAY!
www.informit.com/safaritrial



Find trusted answers, fast

Only Safari lets you search across thousands of best-selling books from the top technology publishers, including Addison-Wesley Professional, Cisco Press, O'Reilly, Prentice Hall, Que, and Sams.



Master the latest tools and techniques

In addition to gaining access to an incredible inventory of technical books, Safari's extensive collection of video tutorials lets you learn from the leading video training experts.

WAIT, THERE'S MORE!



Keep your competitive edge

With Rough Cuts, get access to the developing manuscript and be among the first to learn the newest technologies.



Stay current with emerging technologies

Short Cuts and Quick Reference Sheets are short, concise, focused content created to get you up-to-speed quickly on new and cutting-edge technologies.





FREE Online Edition

Your purchase of **Sams Teach Yourself the Twitter API in 24 Hours** includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Sams book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, O'Reilly, Prentice Hall, and Que.

SAFARI BOOKS ONLINE allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

Activate your FREE Online Edition at www.informit.com/safarifree

- **STEP 1:** Enter the coupon code: GQPQFDB.
- **STEP 2:** New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

