

Prepared exclusively for RubyLearning Fans on Facebook

## Using Twitter with Ruby

Copyright © 2009-2010 RubyLearning  
<http://rubylearning.org/>

### All Rights Reserved

You are not authorized to modify or remove any of the contents in this guide. Nor are you authorized to monetize this guide. However if you wish to pass along this guide or offer this guide as a bonus or gift, you are permitted to do so.

### Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty of fitness is implied. The information provided is on an "as is" basis. The authors 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.

Revised Edition - 23<sup>rd</sup> Apr. 2009  
First Edition - Mar. 2009

## **PLEASE SUPPORT RUBYLEARNING**



## Table of Contents

Silver Sponsors .....	3
Acknowledgements .....	6
Introduction.....	7
Who is the eBook for? .....	7
Concept and Approach.....	7
How to Use This eBook.....	7
About the Conventions Used in This eBook.....	8
Resources.....	9
Using Twitter with Ruby.....	10
What's Twitter? .....	10
Some Twitter concepts.....	10
A brief look at REST .....	11
Twitter API .....	12
What's cURL? .....	13
Downloading and installing cURL .....	14
Using cURL .....	15
cURL and the Twitter API.....	17
Parsing Twitter XML.....	22
What is Hpricot? .....	23
Net::HTTP.....	26
Tweeting .....	29
Following People .....	31
On Your Own.....	36
Abstractions and The Twitter Gem.....	36
The Twitter Gem.....	38
Exercises .....	38
About the Author .....	40

## Silver Sponsors



ProjectLocker provides tools and services to help software development teams quickly set up an enterprise-quality infrastructure at an affordable price. Using ProjectLocker, teams can spend less time finding, configuring, and managing tools and more time doing what is most important – writing great software to meet customer needs.

ProjectLocker's suite of software development tools features source control with both Git & Subversion hosting, and bug tracking, wiki, and project management with hosted Trac. These solutions are available almost immediately after signing up online and come in an array of subscription packages:

- ProjectLocker Free - an ad-supported option that offers up to 300MB of space for 2 users that's available at no cost to the clients, who can then earn more space & users through invitations and referrals.
- ProjectLocker Lite - the basic paid service with multiple levels (Seed, Angel, Venture, and Equity) tailored to the growth stages of a software company that allows clients to gracefully expand as needed, starting at \$2.50 per month for 1GB of storage space.
- ProjectLocker Enterprise - for full scale IT departments seeking to lower the total cost of ownership of their development infrastructure, where clients have no preset data limits and can purchase both ProjectLocker's standard Trac & SCM offerings as well as a number of proprietary productivity tools on an *a la carte* basis.

In addition, ProjectLocker integrates with many of the most popular and successful development tools in the cloud, including Basecamp, Rally, FogBugz, Twitter, Jabber, and Mor.ph. These features, along with the availability of unlimited projects and repositories for all service levels, lets ProjectLocker provide everything teams need to communicate with each other and with their customers.

ProjectLocker has a record of customer service and reliability that is unmatched in the hosted tools space. ProjectLocker provides both e-mail and telephone technical support to its clients. It also performs hourly backups of all client data that can be used to implement up to bare metal restores if necessary and has maintained 99.9% uptime for its hosted servers since 2003.

ProjectLocker's web-based delivery model allows teams to access their tools securely anywhere they can access the Internet — even through most client firewalls.

For more information, contact ProjectLocker by calling +1-866-GO-GET-PL (+1-866-464-3875), e-mailing sales (at) projectlocker.com, or by visiting <http://www.projectlocker.com>.



Twitter, twitter! Are you twittering all around and nobody hears your sweet voice? That's strange. Perhaps Twitter isn't your marketing tool. However for Railsware (<http://www.railsware.com/>), Twitter became that real call to action when the team of Ruby on Rails developers preaching methods of agile programming came together in 2005. Until that time Twitter had not become so 'rich and famous' as it is now, developers were open to new ideas and the goal was somewhere far away.

With its very first project, Railsware got focused on fast implementation of web applications using agile methods (<http://www.railsware.com/be-agile>). As another example, 37signals became one of those companies who succeeded in elegant and powerful Rails development. Their book "Getting Real" became a universal guide for Railsware (<http://railsware.com/services>) to building successful web applications.

Open-source Rails community gives developers an opportunity to discover new methods and tools, to meet requirements not met by the licensed software. Twitter is the same. Even today Twitter is probably the biggest Rails site on the web and its growing scalability is a loud fact of its open-source nature.

Do you remember that Email, IM, RSS, SMS and the web are just transports as well as Twitter? Railsware gives every startup-idea a fast scalable 'ware' delivering solution.

## Acknowledgements

I would like to thank Amanda for giving me a nudge when I needed one and letting me know how I can share my knowledge. I would also like to thank Satish for the excellent idea for a Twitter course.

# Introduction

## *Who is the eBook for?*

This "Using Twitter with Ruby" eBook is a starting point for people new to the Twitter API and a guide to help learn it as quickly and easily as possible.

To work with the Twitter API, some knowledge of the following technologies is recommended -

- The Ruby programming language
- Hpricot
- Net::HTTP library

We shall cover both Hpricot and Net::HTTP in this eBook.

## *Concept and Approach*

I have tried to organize this eBook so that each chapter builds upon the skills acquired in the previous chapters, to make sure that you will never be called upon to do something that you have not already learned. This eBook not only teaches you how to do something, but also provides you with the chance to put those morsels of knowledge into practice with exercises. It therefore contains several exercises, or assignments, to facilitate a hands-on learning approach.

## *How to Use This eBook*

I recommend that you go through the entire eBook chapter by chapter, reading the text, running the provided examples and doing the assignments along the way. This will give you a much broader understanding of how things are done (and of how you can get things done), and it will reduce the chance of anxiety, confusion, and worse yet, mistakes.

It is best to read this eBook and complete its assignments when you are relaxed and have time to spare. Nothing makes things go wrong more than working in a rush. And keep in mind that the assignments in this eBook are fun, not just work exercises. So go ahead and enjoy them.

### ***About the Conventions Used in This eBook***

Explanatory notes (generally provide some hints or gives a more in-depth explanation of some point mentioned in the text) are shown shaded like this:

This is an explanatory note. You can skip it if you like - but if you do so, you may miss something of interest!

When you find some text as follows:

```
$ gem install sinatra
```

What you need to type is the text in bold. In the above example it is:  
**gem install sinatra**

Though we will be discussing Ruby 1.8.6 on the Windows platform, these notes should be appropriate for Linux/Mac OS X too unless explicitly stated otherwise.

Finally, if you notice any errors or typos, or have any comments or suggestions or good exercises I could include, please email Satish at: [mail@satishtalim.com](mailto:mail@satishtalim.com).



## Resources

1. Twitter -

<http://www.twitter.com/>

2. RESTful

[http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer)

3. Twitter API doc -

<http://apiwiki.twitter.com/Twitter-API-Documentation>

4. cURL -

<http://curl.haxx.se/>

5. cURL download and install -

<http://curl.haxx.se/dlwiz/>

[http://twittut.netsensei.nl/?page\\_id=8](http://twittut.netsensei.nl/?page_id=8)

## Using Twitter with Ruby

### *What's Twitter?*

Twitter<sup>1</sup> is a popular micro-blogging website. Instead of using this blogging service to write long, in-depth articles, Twitter users (or “Tweeple”) post single-sentence updates at a time. Twitter isn't used as formally as many other blogs, but rather to pass small pieces of information, links, thoughts or comments to your friends. Also, instead of people having to visit your Twitter page to check your updates, they instead visit a page that contains the updates of everyone they're following, allowing them to keep up with everyone with the minimum of effort.

However, another thing Twitter is famous for is having an open API, allowing people to write software that interacts with Twitter. This can be done without “screen scraping” (or parsing HTML pages), without worrying Twitter will change its API, and with the blessing of Twitter itself. It's also quite easy to write software that interacts with Twitter. This is why you so many Twitter clients around, they're just so easy to write!

If you don't have one already, go ahead and create a free account on Twitter. We shall need the same for the examples in this eBook.

### **Some Twitter concepts**

Twitter updates (or “Tweets”) are limited to a length of 140 characters. When you think about it, there's a lot you can say in 140 characters, but using Twitter changes the way you say it. Updates are short and to the point, often only consisting of a simple statement and a link. Also, any links you post in the update have to fit in the 140 characters, so URL shortening services (such as [tinyurl.com](http://tinyurl.com)) are used. These save the ever-precious characters from being wasted on lengthy URLs.

Updates are viewed in collections called a “Timeline.” The default timeline you'll see on Twitter is your friend's timeline, but there are also public timelines and a specific user's timelines. A timeline is an ordered collection of updates, with the newest appearing at the top. In order to have someone's updates appear on your friend's timeline, you have to “follow” them. People who wish to view your updates will, in turn, follow you.

Twitter is also a two-way street. When you write an update, you often get replies from those following you. These are called @replies, just one of the many commands you can use in Twitter.

### ***A brief look at REST***

The Twitter API is what's called a RESTful API. RESTful (or “Representational State Transfer”) is a way of organizing the interface to your web application by representing everything as resources. To post an update, you create a new update resource. To follow someone, you create a new friendship resource. REST provides a uniform and predictable interface to all features of Twitter.

In addition to creating resources, each resource can be uniquely addressed, destroyed, etc using the same URL interface. This, again, makes all features uniform and predictable.

There are four primary actions in a RESTful application.

- Create - Create a new instance of the resource. For example, create a new update or friendship.
- Destroy - Destroy an instance of the resource. For example, delete an update or “unfollow” another Twitter user.
- Update - Change the contents of a resource. In the Twitter API, this is usually expanded to changing one part of a resource, such as changing the background image on your profile or changing your email address.
- Show - This is what most API calls in Twitter end up calling. The show action simply returns data about a resource. This is used to view timelines, individual updates, information about users, etc.

This is also the only idempotent action (meaning it makes no changes to the resource), so show-related API calls are the only calls you'll make to resources owned by other users. You can only call Create, Destroy or Update on resources you own.

Being RESTful is one of the key features that make the Twitter API so easy to use.

## ***Twitter API***

The Twitter API is built on HTTP, REST and XML.

API calls are made via HTTP. The API call itself is the URL of the request. Any additional parameters are passed as either GET parameters in the URL, or as POST parameters in the request body. API responses are returned via HTTP, both in the HTTP response status and the response body. Additionally, HTTP is used for authentication. When every request is made, the HTTP basic access authentication headers will be used to identify you. Since the same Twitter API call made by two different people can mean drastically different things, it's essential to pass authentication information with every Twitter request. Cookies aren't used with the Twitter API either, so Twitter doesn't remember who you are between HTTP requests.

REST defines the basic structure of the API calls. Most API calls create, destroy or view resources. The basic structure of Twitter API call URLs is ***/resource/action/id.format***. For example, to see the contents of a single status update with the id of 1312750710 in XML format, a request to the following URL would be made: <http://twitter.com/statuses/show/1312750710.xml>. Similarly, to see information about the user with the id of 21949890 in XML format, a request to the following URL would be made: <http://twitter.com/users/show/21949890.xml>. As you can see, the requests are very similar. The resource has changed (we're not requesting information about a status, we're requesting information about a user), the ID has changed to reflect the user we want to request information about, but the rest has stayed the same. The action still remains as 'show' (since we're requesting information about something)

and the format remains XML. It's because of REST that things remain so uniform.

Finally, XML is used to encode all information returned from Twitter. XML is universal; there isn't a major programming language that doesn't have at least one XML parser available to it. XML parsers are also relatively easy to use. There are even Ruby libraries to quickly map XML text to Ruby objects for very short and expressive programs.

It's important to note that Twitter provides access to its API free of charge and without requiring you to apply for special licenses. This is done in good faith, and it's expected you'll use this service responsibly. Though there are some limitations (such as rate limiting on the commands you can use to 60 per hour), Twitter is still easily abused. Please exercise caution when submitting API calls to Twitter, double-check your software for any bugs that would generate excessive Twitter API calls and use the parameters available in many API calls to retrieve only the information you need. For example, to check if there are any new updates on a timeline, don't request the entire timeline, but request only the updates newer than the previous updates you received.

## ***What's cURL?***

The HTTP protocol is too complex to work by hand. Though it is possible to use nothing but a tool such as netcat or telnet to connect to a web server and make HTTP requests, it's not practical. There are too many headers to remember, it's too easy to make a mistake and some requests include 5 to 10 separate headers. cURL is a tool and library designed to give you a user-friendly but low-level interface to making HTTP requests. cURL also supports many other protocols related to uploading and downloading files, but we won't be using those.

cURL will be used to explore the Twitter API without getting into how to make HTTP requests in Ruby. It's important to understand how to make API calls in a tool that's easy to understand before trying to tackle the Twitter API with a more complicated HTTP interface.

## Downloading and installing cURL

cURL is available on many platforms, including Windows (all variants), OSX (all versions) and Linux (all distros). cURL is available from the cURL website at <http://curl.haxx.se/>.

To install cURL on Linux, you should first check for cURL binaries in your distributions package repository. For exact instructions on how to do this, refer to your distribution's documentation. On Ubuntu and other Debian variants, you can install cURL by issuing the following command.

```
$ sudo apt-get install curl
```

If cURL is not available in your distribution's package repository or you're running another operating system, you can find the right cURL package by following the download wizard located at <http://curl.haxx.se/dlwiz/>. Simply select the "curl executable" option, input information about your computer and operating system and it will direct you to the correct file download. Once downloaded, extract any files in the archive to an easily accessible file (for example, c:\cURL on Windows).

cURL is a command-line tool. This means it is run from the command prompt (in Windows) or terminal (in Linux or OSX) and its results are displayed in the terminal window. There is no graphical interface. So in order to use cURL, you first have to open a command-line window.

On Windows, go to Start -> Run and enter "cmd" (without quotes) into the dialog and press enter.

On OSX, open the Terminal application. You'll find this in the Applications/Utilities folder.

On Linux, open a Bash Prompt or Terminal window. Again, since all distributions are different, refer to your distributions documentation for more detailed information. On Ubuntu, you can open a bash prompt by going to Applications -> Accessories -> Terminal.

Though Windows, OSX and Linux all run different command-line shells (the program that interprets commands on the terminal and runs executables), the commands presented here will work on all of them.

First, change the directory to where you extracted the cURL archive. If you installed cURL using an installer or a package on Linux, cURL will already be in your path (meaning it can be run even though you're not currently in the same directory) so this isn't needed. For example, if I placed cURL in C:\cURL on my Windows computer, I would issue the following command.

```
> cd C:\cURL
```

You can now issue cURL commands. On Linux or OSX, you may have to prepend `./` to your commands. For example, instead of `curl http://some/url`, the command you'd use would be `./curl http://some/url`.

To test cURL, enter the following command.

```
curl -I http://twitter.com/
```

You should see HTTP response headers. The first line should be "HTTP/1.1 200 OK." This means you've made a successful request to Twitter using cURL. You're now ready to begin exploring the Twitter API.

## Using cURL

cURL is a command-line tool. This means it's launched from a command line with a number of switches and parameters. Switches change how the cURL command will act, and begin with the `-` character. For example, the previous example used the `-I` switch, which means to display the HTTP response headers only, and not the HTTP response body. Parameters pass information to either a switch, or the cURL

command itself. In the previous example, the Twitter URL was a parameter.

There are a number of switches that will be useful to you while exploring the Twitter API. Though cURL has a great many switches, not all of them will be covered here. If you want to know more about cURL, see the cURL documentation page at <http://curl.haxx.se/docs/>.

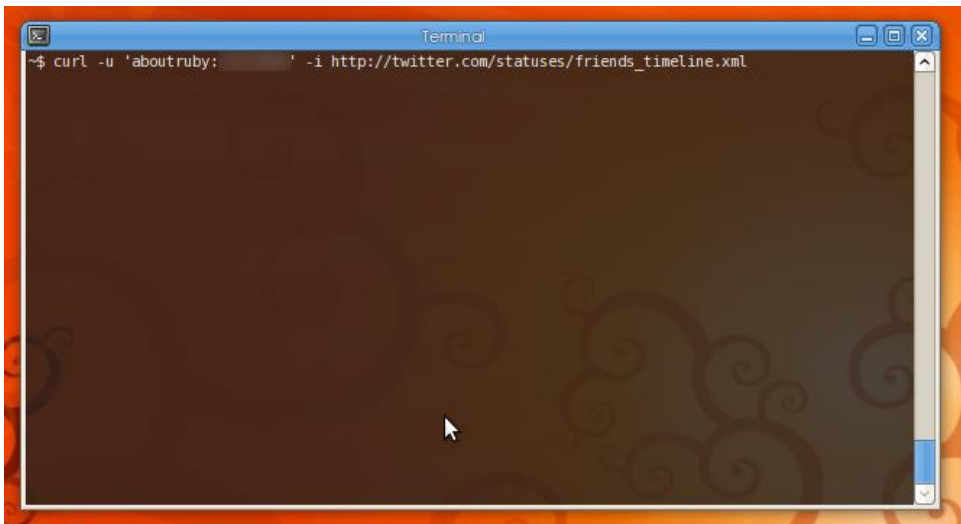
- -l - Display only HTTP response headers, don't display the HTTP response body. This is useful for simple tests and with certain commands. For example, a command that posts a new update will return an HTTP OK status, but the body isn't very interesting.
- -i - Display the HTTP response headers as well as the HTTP response body. Unless you're using the -l switch, you should be using the -i switch since, when working with the Twitter API, the response headers are always of interest to us.
- -u username:password - Every Twitter API call has to be authenticated, and this provides us a way to tell cURL to use the following username and password for HTTP basic authentication. This will be a part of every cURL command when working with the Twitter API. Obviously, replace username and password with your Twitter username and password.
- -d var=data&var2=data - Use the following data as POST data. When creating new resources, the data will be used to create the new resource. This will be used primarily when creating new update resources.

The cURL commands used to interface with Twitter will have a -u switch, a -d switch and either an -i or -l switch as well as the URL for the Twitter API we wish to access. For example, to view my friend's timeline, I would issue the following cURL command.

```
curl -u username:password -i  
http://twitter.com/statuses/friends\_timeline.xml
```

See the following screenshot:





Refer the following for a bigger screenshot -  
<http://rubylearning.com/images/curl/curl.png>

## cURL and the Twitter API

cURL can be used to make API calls to Twitter quite easily. Since, as discussed before, API calls to Twitter are made by making requests to certain URLs, all that's needed is to make some requests and examine the output.

Different Twitter API calls require different HTTP request methods. Though there are four request types in HTTP (GET, POST, PUT and DELETE) that can be mapped directly to REST actions, since web browsers only include support for two of those (GET and POST), that's all that needs to be used. The rule of thumb for knowing which request type to use for any Twitter API call has to do with *idempotence*. An idempotent action is an action that makes no changes to the state of the application. For example, requesting a timeline doesn't change anything, so the action is idempotent. Idempotent API calls are made by GET requests. All other API calls (that, for example, create a new resource) use POST requests. The request type doesn't have to be explicitly defined in the cURL command-line though, but any cURL

command-line with the `-d` switch will automatically be made into a POST request.

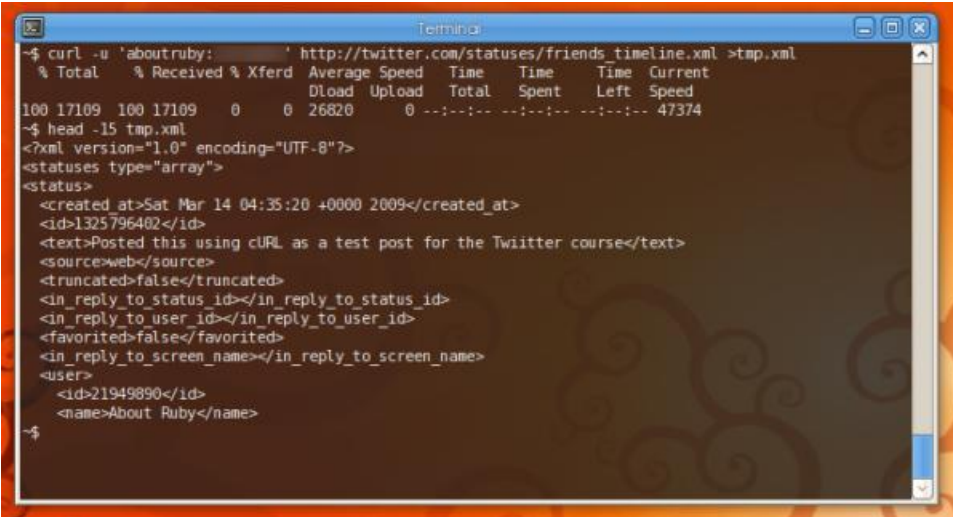
So let's take a look at a typical command: viewing a timeline. Since this doesn't make any changes, it's an idempotent request and therefore a GET request. The API call for this command is `http://twitter.com/statuses/friends_timeline.xml`. We want this in XML format, so we'll specify this with `.xml` at the end. When you make this API call from the command-line, you'll get quite a long XML output. For more comfortable viewing, it's recommended that you pipe the output of the `cURL` command into a file. The file can then be opened in a text editor and examined more closely.

Piping something on the command-line is to take the output of a program and either send it to the input of another program (forming a pipeline) or to a file. In this case, we'll be sending the output of the command to the file `tmp.xml`.

```
$ curl -u username:password  
http://twitter.com/statuses/friends_timeline.xml >tmp.xml
```

Refer to the Screenshot:

<http://rubylearning.com/images/curl/curlpipe.png>

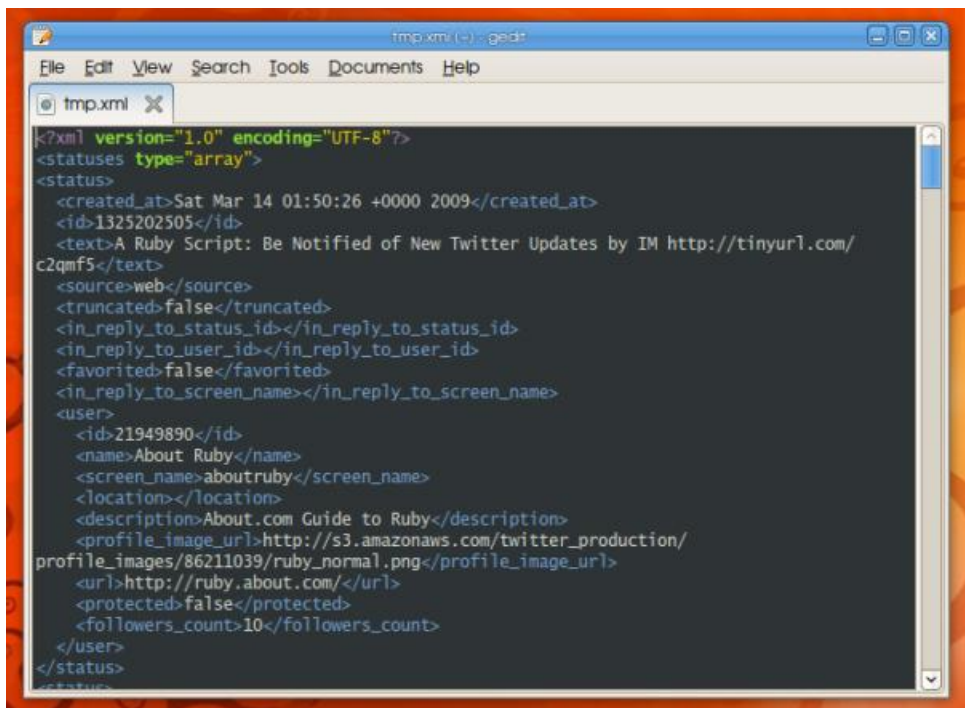
A screenshot of a terminal window titled "Terminal". The terminal shows a command being executed: `curl -u 'aboutruby:' 'http://twitter.com/statuses/friends_timeline.xml' >tmp.xml`. Below the command, there is a table showing progress statistics: % Total, % Received, Xferd, Average Speed, Time, Time, Time, Current. The output shows 100% completion. Then, the command `head -15 tmp.xml` is executed, displaying the first 15 lines of the XML response. The XML is a status update from "About Ruby" created on Sat Mar 14 04:35:20 +0000 2009, containing the text "Posted this using cURL as a test post for the Twitter course".

```
Terminal  
~$ curl -u 'aboutruby:' 'http://twitter.com/statuses/friends_timeline.xml' >tmp.xml  
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current  
           Dload  Upload  Total   Spent    Left   Speed  
100 17109 100 17109    0     0 26820    0 --:--:-- --:--:-- --:--:-- 47374  
~$ head -15 tmp.xml  
<?xml version="1.0" encoding="UTF-8"?>  
<statuses type="array">  
<status>  
  <created_at>Sat Mar 14 04:35:20 +0000 2009</created_at>  
  <id>1325796402</id>  
  <text>Posted this using cURL as a test post for the Twitter course</text>  
  <source>web</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>21949890</id>  
    <name>About Ruby</name>  
  </user>  
</status>  
</statuses>  
~$
```

As you can see from the above screenshot, the advantage of looking at this XML data in a text editor is obvious. First, its syntax highlighted. This means certain portions of the file are colored differently, such as the element names and XML declarations. This makes it quite a lot easier to read. Second, some terminal windows can't be scrolled easily or their buffers aren't large enough to hold the entire XML stream. Text editors are available for free for all platforms. The editor in the screenshot is gedit on Ubuntu Linux. Windows users can download Notepad++ for free from the following URL: <http://notepad-plus.sourceforge.net/>.

Refer the screenshot -

<http://rubylearning.com/images/curl/curlgedit.png>



Now check the XML generated from the previous command. As you can see, it's rather straightforward. This XML snippet represents one complete update. Each element of XML carries its own piece of information. For example, the `created_at` element stores a string representing the time the update was posted and the `text` element holds the contents of the update. Roughly half of this snippet is taken up by the `user` element, which describes the user that posted this update. For convenience, every status element contains a user element. Even though this is repetitive, it prevents clients from making additional Twitter API requests to get information about each user.

```
<status>
  <created_at>Sat Mar 14 01:50:26 +0000 2009</created_at>
  <id>1325202505</id>
  <text>A Ruby Script: Be Notified of New Twitter Updates by IM
http://tinyurl.com/c2qmf5</text>
  <source>web</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>21949890</id>
    <name>About Ruby</name>
    <screen_name>aboutruby</screen_name>
    <location></location>
    <description>About.com Guide to Ruby</description>

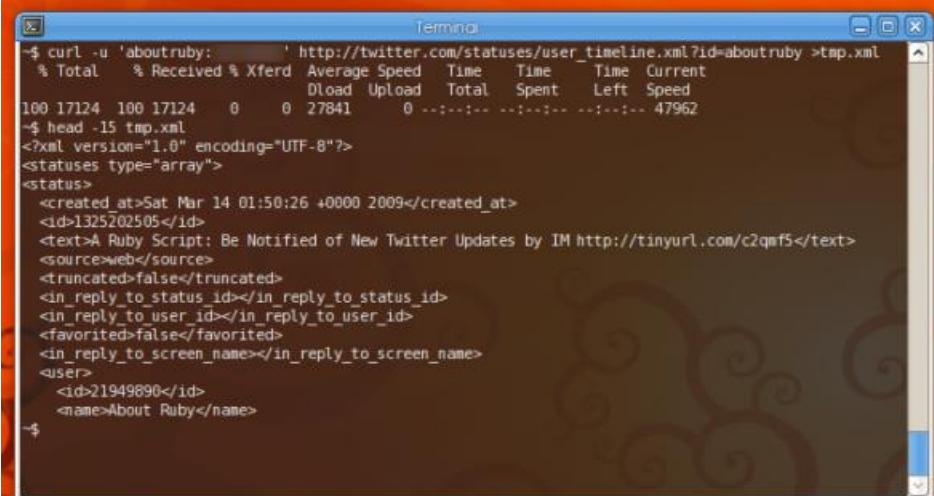
    <profile_image_url>http://s3.amazonaws.com/twitter_production/prof
ile_images/86211039/ruby_normal.png</profile_image_url>
    <url>http://ruby.about.com/</url>
    <protected>>false</protected>
    <followers_count>10</followers_count>
  </user>
</status>
```

Some Twitter API calls require you to pass certain parameters along with it. There are two ways to do this, depending on whether the API request is a GET or POST request. If the request is a GET request, the parameters will simply be added to the end of the URL. For example, to get a specific user's timeline, the following Twitter API call can be made.

```
$ curl -u username:password  
http://twitter.com/statuses/user_timeline.xml?id=aboutruby  
>tmp.xml
```

Refer screenshot - curlGETparameters.png -

<http://rubylearning.com/images/curl/curlGETparameters.png>



```
terminal  
~$ curl -u 'aboutruby:' 'http://twitter.com/statuses/user_timeline.xml?id=aboutruby' >tmp.xml  
% Total % Received % Xferd Average Speed Time Time Time Current  
Dload Upload Total Spent Left Speed  
100 17124 100 17124 0 0 27841 0 --:--:-- --:--:-- --:--:-- 47962  
~$ head -15 tmp.xml  
<?xml version="1.0" encoding="UTF-8"?>  
<statuses type="array">  
<status>  
<created_at>Sat Mar 14 01:50:26 +0000 2009</created_at>  
<id>1325202505</id>  
<text>A Ruby Script: Be Notified of New Twitter Updates by IM http://tinyurl.com/c2qmf5</text>  
<source>web</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>21949890</id>  
<name>About Ruby</name>  
~$
```

Additional parameters can be added with the & character. For example, if I only wanted to retrieve 5 updates from the aboutruby user, the Twitter API URL would be -

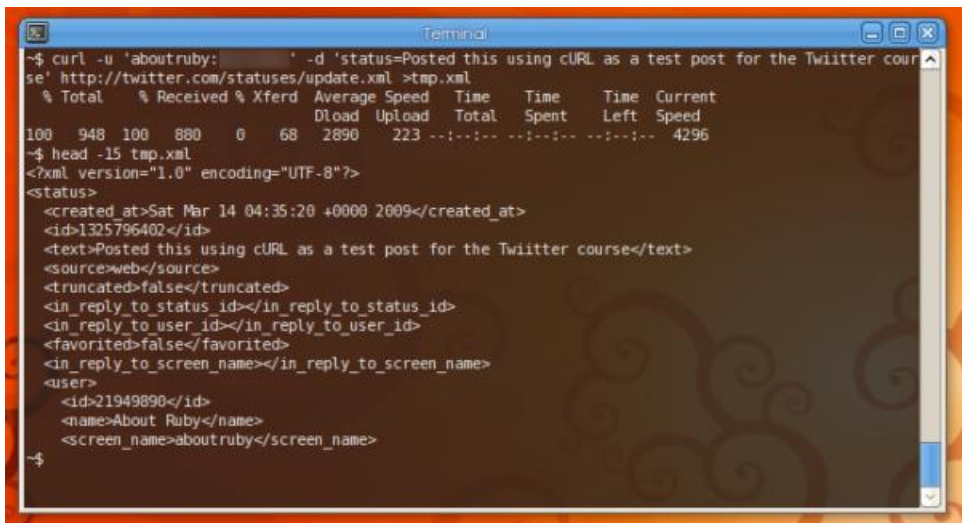
[http://twitter.com/statuses/user\\_timeline.xml?id=aboutruby&count=5](http://twitter.com/statuses/user_timeline.xml?id=aboutruby&count=5)

A different method is used for HTTP POST requests. Instead of appending them to the URL, cURL's -d switch is used. For example, to post a new update, the following curl command-line can be used.

```
$ curl -u username:password -d 'status=Posted this using cURL!'  
http://twitter.com/statuses/update.xml
```

Refer screenshot - curlupdate.png

<http://rubylearning.com/images/curl/curlupdate.png>



```
Terminal
~$ curl -u 'aboutruby:' -d 'status=Posted this using cURL as a test post for the Twitter course' http://twitter.com/statuses/update.xml >tmp.xml
% Total    % Received % Xferd  Average Speed   Time    Time     Current
                                 Dload  Upload   Total   Spent    Left     Speed
100  948    100  880    0   68   2890    223  --:--:-- --:--:-- --:--:--  4296
~$ head -15 tmp.xml
<?xml version="1.0" encoding="UTF-8"?>
<status>
  <created_at>Sat Mar 14 04:35:20 +0000 2009</created_at>
  <id>1325796402</id>
  <text>Posted this using cURL as a test post for the Twitter course</text>
  <source>web</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>21949890</id>
    <name>About Ruby</name>
    <screen_name>aboutruby</screen_name>
  </user>
</status>
```

Now that you know how to make requests to the Twitter API using cURL, as well as how to supply the needed parameters on GET and POST requests, you can begin to explore the Twitter API on your own. The Twitter API Wiki has an excellent document explaining all the Twitter API commands available. This document is available from - <http://apiwiki.twitter.com/REST+API+Documentation>

## ***Parsing Twitter XML***

The data that any Twitter API call returns will be encoded in either XML or JSON. This format is defined in the URL, any URL ending in .xml will return XML data. This XML data must be parsed in order to be used. In short, it must be turned from textual representation into data structures we can query for specific pieces of information.

While there are several options available when parsing XML in Ruby, I've chosen Hpricot only because I'm most familiar with it. Ruby comes with REXML, a suitable parser, and there is another called Nokogiri, which provides an Hpricot-like interface and is (supposedly) faster, but harder to install.

## What is Hpricot?

Hpricot is a clever little HTML and XML parsing library by *why the lucky stiff*. It features a quick parsing library written in C and a compact and expressive interface. It's quite easy to pick out just the tag you need using a CSS-like syntax. It's easy to learn, it's faster than REXML (which is written in pure Ruby) and it's an excellent tool to have around.

Note that Hpricot is not meant to be a formal XML parser. No attempt to validate the XML is performed. In fact, it isn't meant to be an XML parser at all, but rather an HTML parser. Since XML is a subset of HTML, this works just fine. However, many things valid in HTML are invalid in XML, and Hpricot might make sense out of something that's certainly invalid XML.

Installing Hpricot is slightly more complicated than installing a normal gem. It requires a small C library to be compiled, so you must either have a C compiler installed on your system or a binary gem compiled for your system. Linux users can install the GCC compiler and Ruby development libraries (on Ubuntu, install the *build-essential* and *ruby1.8-dev* packages). Windows users will have no problems, as a binary gem is provided. For more information about installing Hpricot, see the "Installing Hpricot" page at this URL:

<http://wiki.github.com/why/hpricot/installing-hpricot>

So once you have Hpricot installed, it's time to dive in. Let's fetch a timeline from Twitter for something to test on. Run the following command to download the timeline into the file *timeline.xml*.

```
$ curl http://twitter.com/statuses/user_timeline.xml?id=aboutruby  
>timeline.xml
```

To load this into Hpricot, use the *Hpricot* method to create an Hpricot document. Pass it the XML text you wish to parse. In this case, use *File.read* to quickly read in the contents of *timeline.xml*.

```
#!/usr/bin/env ruby
require 'rubygems'
require 'hpricot'

doc = Hpricot( File.read('timeline.xml') )
```

At this point, you're ready to start reading elements from the XML document. One of the beautiful things about Hpricot is its simple interface. It's just one line to read a file, parse the contents and create a document. After that, it's just one line to grab any particular element or iterate over a collection of elements. Hpricot makes a lot of things very easy.

First, you should learn how to iterate over a collection of elements. If you look at the structure of *timeline.xml*, you can see that there is a *statuses* element, and in this element is a number of *status* elements. When parsing a timeline, you'll usually want to iterate over the *status* elements and do something with each of them. The following program will print the text of each status update.

```
#!/usr/bin/env ruby
require 'rubygems'
require 'hpricot'

doc = Hpricot( File.read('timeline.xml') )

(doc/'status').each do |st|
  puts (st/'text').inner_html
  puts "-" * 50
end
```

The basic way of querying an Hpricot document is to use the */* operator. Pass to this the name of an element you wish to find. If Hpricot finds more than one of these elements, it will return an *Hpricot::Elements* object, which may be iterated over with the *each* method. For each of these elements iterated over, you can do further queries on them with the */* operator. For each element, we want to



retrieve the *text* element, so this is what we query for. Finally, since we don't want the element itself but only the text inside the element, call the *inner\_html* method on what the second query returned. This method will return any text inside the element, which is just what we want.

Note that an extra pair of parentheses is needed when doing queries in Hpricot. The code *doc/'status'* is perfectly valid, but the code *doc/'status'.each* doesn't do what you'd expect it to. Since the dot operator has a higher precedence than the / operator, this code would be equivalent to *doc/('status'.each)*. To get around this, the extra set of parentheses to make the / operator work is required, so it's *(doc/'status').each*.

You can query deeper into the XML hierarchy as well. Getting the text of an update is easy, it's only one level deep inside the *status* element, but getting the name of the user that posted that update is a bit harder. Since these query strings are CSS-style, you can query for an element nested in another element by putting the two elements separated by a space. In this example, we want the *name* element inside of the *user* element as well as the *text* element.

```
#!/usr/bin/env ruby
require 'rubygems'
require 'hpricot'

doc = Hpricot( File.read('timeline.xml') )

(doc/'status').each do |st|
  name = (st/'user name').inner_html
  text = (st/'text').inner_html

  puts "#{name} says: #{text}"
  puts "-" * 50
end
```

Believe it or not, this is about all you need to know about Hpricot. You can now query your timelines for any information you'll need, and iterate over the major elements. If you want to know more about Hpricot, visit the Hpricot homepage at <http://wiki.github.com/why/hpricot/>.

## ***Net::HTTP***

Now that we know how to parse the XML returned by Twitter, it's time to fetch that XML directly from our Ruby code. After all, parsing XML stored in a file by cURL is not very exciting or practical.

Before jumping in, I'd like to reiterate that the Twitter API is free and open for all to use. Please don't abuse it intentionally or accidentally. Check all code before running it, make sure there are proper delays between any Twitter API calls you make, and request only the information you need.

Ruby comes with a library for making HTTP queries called `Net::HTTP`. It's written in pure Ruby and doesn't require you to install anything. Though there are some really neat libraries out there that can help you (such as *httparty*), these are beyond the scope of this course.

To begin using `Net::HTTP`, require the `'net/http'` library. You'll also need to have your username and password for Twitter handy and stored in variables. Depending on where your scripts reside, it may not be a good idea to keep these hard-coded in the file. If someone else may see them, prompt for them instead of storing them.

The following example creates a very primitive *twitter* method. It takes a single argument, the Twitter API command to run. It doesn't support any parameters to the Twitter API calls, nor does it support POST commands. It serves to illustrate the GET request with `Net::HTTP`.

```
#!/usr/bin/env ruby
require 'net/http'

$username = 'your_username'
$password = 'your_password'

def twitter(command)
  twitter = Net::HTTP.start('twitter.com')

  req = Net::HTTP::Get.new(command)
  req.basic_auth($username, $password)

  return twitter.request(req).body
end

puts twitter(ARGV[0])
```

The *Net::HTTP.start* method opens a TCP connection to twitter.com. This is the channel through which any requests you make are made.

The *Net::HTTP::Get* class represents an HTTP GET request. The *basic\_auth* method tells the GET request to use HTTP Basic auth with the following username and password. Finally, the request is made by passing the request object to the *twitter.request* method. This performs the actual request, and the HTTP result object is returned. Since we're only interested in the body, call the *body* method on this object and return it.

It's as easy as that, just a few lines. However, to make this code useful, we're going to have to add two features. First, you need to be able to make both GET and POST requests. Second, you need to be able to pass parameters to both the GET and POST requests.

The following code demonstrates on how this can be done. The default request type is GET, so most requests won't need the optional second parameter. Note that the API for defining GET and POST arguments is different, so a *case* statement is used. The *case* statement doesn't introduce a new scope, so any variables (such as *req*) created inside the *case* statement are available outside of the *case* statement.

In order to implement parameters for the GET request type, a quick and

dirty hack is needed. Remember the basic format for the argument string is *?key1=value&key2=value*, which is appended to the URL we wish to access. To do this, the *opts* hash is mapped to *key=value* pairs and joined with the & character. If *opts* is empty, a single ? character will be appended to the URL. This is harmless and won't effect how Twitter parses the URL you submit.

Other than that, making a POST request is much the same as making a GET request. The parameters are much easier to implement, simply call the *set\_form\_data* method on the request object.

The *twitter* method now calls *Hpricot* on the returned body as well. Since this will almost always be done just after calling the *twitter* method, this saves a step for the end user.

```
#!/usr/bin/env ruby
require 'rubygems'
require 'hpricot'
require 'net/http'

$username = 'your_username'
$password = 'your_password'

def twitter(command, type=:get, opts={})
  twitter = Net::HTTP.start('twitter.com')

  case type
  when :get
    command << "?" + opts.map{|k,v| "#{k}=#{"#{v}"}.join('&')}
    req = Net::HTTP::Get.new(command)

  when :post
    req = Net::HTTP::Post.new(command)
    req.set_form_data(opts)
  end

  req.basic_auth($username, $password)

  return Hpricot( twitter.request(req).body )
end

doc = twitter('/statuses/friends_timeline.xml')

(doc/'status').each do|st|
  user = (st/'user name').inner_html
  text = (st/'text').inner_html

  puts "#{user} said #{text}"
end
```

## ***Tweeting***

From the previous sections, we know how to send HTTP requests to Twitter and parse any XML it returns. All that remains is to apply this and do something useful.

First, so we don't have to repeat ourselves in every program, take the *twitter* method from the previous examples and save it in a file called *twitter.rb*. Then, any program that wants to use the Twitter method just has to *require 'twitter.rb'* and define the *\$username* and *\$password*

variables. As your Twitter programs grow, you will probably want to come up with a more sophisticated abstraction for working with Twitter. For the time being, this simple code does the job nicely.

```
# File: twitter.rb
require 'rubygems'
require 'net/http'
require 'hpricot'

def twitter(command, type=:get, opts={})
  twitter = Net::HTTP.start('twitter.com')

  case type
  when :get
    command << "?" + opts.map{|k,v|
      "#{k}=#{"#{v}"}.join('&')}
    req = Net::HTTP::Get.new(command)

  when :post
    req = Net::HTTP::Post.new(command)
    req.set_form_data(opts)
  end

  req.basic_auth($username, $password)

  return Hpricot( twitter.request(req).body )
end
```

The following example code will post an update. Viewed from a REST perspective, posting an update is creating an update resource. So, knowing that, we know this is going to be a POST request. The API call for creating a new update resource is */statuses/update.xml*. This API call takes only two parameters, but we only need to worry about one: the *status* parameter. This parameter holds the text the update will contain.

To call the *twitter* method correctly, we now have to use all three arguments. The first to define which API call we want to run. The second to define which type of HTTP method we want. The third uses a small Ruby syntax trick. Any hash pairs passed will automatically be assigned to the last argument. This lets us skip the curly braces and we get a cleaner looking method call.

The following program really can't be any simpler. It'll tweet a simple message taken from the command-line. You could even keep this around to post quick updates from the command-line while you're working.

```
#!/usr/bin/env ruby
require 'twitter.rb'

$username = 'your_username'
$password = 'your_password'

twitter( '/statuses/update.xml', :post, 'status' => ARGV[0]
)
```

Wow! This can't be any simpler. Anything with Twitter is just a one-liner away. Now, to use this script, you have to pass an additional argument on the command-line. Note that this argument will most likely contain spaces, so it will have to be put in quotes. Here is the command I used to post an actual update.

```
$ ./tweet.rb "Posted from Ruby, for the Ruby course"
```

## ***Following People***

All Twitter commands follow more or less the same pattern. There's an API call (which may be a GET or a POST call), and there may also be some parameters. As I said before, every function of Twitter is just one line away.

Following people is another POST request. How can you tell if something is a POST or GET request offhand? It all comes down to REST. You can think of following someone as creating a *friendship* resource. However, it's often easier to think of idempotence. Following someone changes the state of the Twitter application. Anything that changes the state of the Twitter application will be a POST request. Anything that simply requests information will be a GET request.

This example program will create a friendship between you and the username listed on the command-line.

```
#!/usr/bin/env ruby
require 'twitter.rb'

$username = 'your_username'
$password = 'your_password'

doc = twitter( "/friendships/create/#{ARGV[0]}.xml", :post )
```

You'll note one difference. The "parameter" here is part of the URL. The format for this command is */friendships/create/id.xml* where the *id* portion is either the numerical ID or, more commonly, the screen name of the user.

Following someone is not very exciting unless you can see their updates. This next example program will check someone's timeline periodically and if there are any new updates, display them.

I've mentioned before about being nice to Twitter. We're going to be doing two things here that are potentially wasteful to Twitter. We're going to be repeatedly issuing the same command. If we issue it too often, there will be no new information. If we issue it too slowly, we won't be able to respond to information very quickly. A good timer for an application like this is 5 minutes. Every 5 minutes, we check for updates.

Twitter tries to prevent itself against malicious users and erroneous scripts by implementing rate limiting. If you try to send too many commands at once, or too many within a certain amount of time, Twitter will start ignoring your commands. Putting repetitive commands like asking for updates on a timer prevents you from hitting these limits. Also, Twitter buffers the timelines, so even if you were somehow able to request a new timeline every 10 seconds, it wouldn't contain anything new right away.

Another way we can be wasteful here is to request updates we've already seen. Initially, we'll be requesting the default timeline size of



20 updates. However, most of the requests after that should return very few updates, most likely zero updates for many of them. To accomplish this, we remember the ID of the latest update we've seen, and request only the updates newer than this ID.

Finally, this is the first automated program we've seen in the examples. The other programs have been a run once, then examine their effects kind of program. This program is designed to just run in the background, so there's something we're missing: error detection. What to do if we send a command wrongly? What if Twitter is down? We can't see the "fail whale" (the cartoon whale Twitter displays while the site is down), so how do we detect when something has gone wrong?

The answer is in the HTTP return codes. If everything went well, Twitter returns a code 200 (HTTP-speak of everything is OK). If anything goes wrong, it will return something else. So, all we have to do is check the HTTP return code from the HTTP result object in the *twitter* method and, if it's not 200, throw an exception. This requires only a few lines of extra code in the *twitter* method.

There are two error messages in particular you should pay attention to: 502 and 503. If you get the 502 error message, you haven't done anything wrong. Twitter is down, you should sleep and try to run the command again. Similarly, you will get 503 if Twitter is too busy to respond to your request.

When a Twitter API call fails, it will also return an error in XML format. We can grab some information about this error and include it in our error message as well.

```
# File: twitter.rb
require 'rubygems'
require 'net/http'
require 'hpricot'

class TwitterError < Exception; end

def twitter(command, type=:get, opts={})
  twitter = Net::HTTP.start('twitter.com')

  case type
  when :get
    command << "?" + opts.map{|k,v| "#{k}=#{"v"}"}.join('&')
    req = Net::HTTP::Get.new(command)

  when :post
    req = Net::HTTP::Post.new(command)
    req.set_form_data(opts)
  end

  req.basic_auth($username, $password)
  res = twitter.request(req)

  unless res.kind_of?(Net::HTTPSuccess)
    doc = Hpricot(res.body)
    raise TwitterError.new( "#{res.code}:
#{(doc/'error').inner_html}" )
  end

  return Hpricot(res.body)
end
```

All the pieces are now in place. We just have to periodically call the API method */statuses/friends\_timeline.xml* to get the updates of everyone we're following. This script is longer than any we've seen thus far. However, the Twitter-related functions are just as plain and simple. There are also sections you've seen before, such as iterating over the status messages and printing them out. Also, notice that the *rescue* block checks for the 503 "Fail Whale" error message and will retry until Twitter comes back online.

Rate limiting is accomplished by using the *sleep* method. It's done in two places, once at the end of the loop and once in the *rescue* statement. This is because if there's an error, the end of the loop will be skipped.

The *last\_id* holds the ID off the last update you've seen. This is sent as the *since\_id* parameter to the */statuses/friends\_timeline.xml* API call.

```
#!/usr/bin/env ruby
require 'twitter.rb'

$username = 'your_username'
$password = 'your_password'

last_id = '1'

while true do
  begin
    doc = twitter(
      '/statuses/friends_timeline.xml',
      :get,
      'since_id' => last_id
    )

    # If there were any updates
    if (doc/'status').length > 0
      last_id = (doc/'status id').first.inner_html

      # Print in reverse order, we want the newest
      # at the bottom for this script
      (doc/'status').reverse.each do|st|
        user = (st/'user name').inner_html
        text = (st/'text').inner_html

        puts "#{user} said #{text}"
        puts "-" * 50
      end
    end

    # Wait for 5 minutes
    sleep 300

  rescue TwitterError => e
    # Fail whale, wait and try again
    if e.message =~ /^50(2|3)/
      sleep 300
      retry
    end
  end
end
```

## ***On Your Own***

From this point, you should have all the tools you need to make any Twitter application you wish. There's only one resource you really need: the Twitter REST API document. It's available the following URL.  
**`http://apiwiki.twitter.com/Twitter-API-Documentation`**

In this document is the description of every API call Twitter provides, its use, whether it's a POST or GET request and any parameters it takes. In short, this is the Twitter programmer's bible.

The abstractions used up to this point have been rather minimal. However, abstractions are not one-size fits all. It's up to you how to adapt this code to your needs. You could leave it as it is, wrap it all up in a module, or provide a class. Future improvements might have a look-up table to tell which API calls are GET and which are POST so the user doesn't have to worry about it. You could even make custom methods for each, so the user doesn't have to remember the Twitter API URLs. The possibilities are endless.

## ***Abstractions and the Twitter Gem***

In the previous chapters, you've learned how to interface with Twitter using Ruby. You've learned how to use the HTTP interfaces to make requests to Twitter, how to parse XML returned by Twitter and how to make any API calls you wish. The rest is up to you.

If you're going to be making Twitter applications, you should probably expand the *twitter* method to be a little nicer. Ideally, interfacing with Twitter should be a nicely *abstracted* process. Users of the method shouldn't have to think about API URLs, request methods and what the parameters are called. When you want to post an update, you should have to call a single method. The depth of your chosen abstraction depends on how much you'll be using this code. For a few simple scripts, a simple abstraction will do.

If you're writing reusable code, the first thing you should do is

*encapsulate*. Make a Twitter module to put all your classes and methods in. In the example scripts, global variables named *\$username* and *\$password* were used. This is far, far less than ideal in a program larger than the scripts presented.

The next thing you have to do is decouple the Twitter abstraction from the authentication process. As it is right now, you can only interact with one Twitter account at a time, since each request uses the username and password global variables. Create a Twitter class that holds the authentication information and allows you to create more than one Twitter API object at the same time.

The level of friendliness is up to you. In its current state, in order to use the *twitter* method, you have to have some familiarity with the Twitter API and how it works. If you are writing reusable code, ask yourself the following question: If I take a three month break from Twitter, am I going to remember all the Twitter API URLs and the parameters they take? The answer is probably no.

One thing you can do is make a hash of actions to URLs. The "action" is something like *friends\_timeline* or *post\_update*. The URLs are the corresponding Twitter API URLs needed to perform these actions. This gives the API calls easier names to remember and a handy table to look at that documents them all.

If you want to go further, you can make a hash of URLs and request types. The request types remember if the API call needs a GET or a POST request, so the user doesn't have to. This would go a long way towards making the abstraction easier to use.

Finally, make a table of parameter names, so the user need only pass the parameters and won't have to worry about their names. This requires some careful thought, but can be beneficial.

There's also work to be done on the XML side. You could pre-parse all of the data and store it in hashes, or in Struct objects. This could reduce the amount of code needed to pull data from the returned XML.

Designing an abstraction can quickly get out of hand though. Ruby

makes abstractions really easy, but as mentioned before, it depends on how much you'll be using these abstractions. It's very easy to start solving problems that don't exist though, so think about every feature before implementing. Ask yourself: Is this really necessary?

## ***The Twitter Gem***

Such an abstraction has already been made. The *twitter* gem by John Nunemaker can be installed using *gem install twitter* and has many of the features you'd want. It's quite easy to use, has abstracted names and takes all of the guesswork and documentation reading out of interfacing with Twitter.

The following is an example program written with the Twitter gem. It will log in, fetch the friends timeline and print the name and text of every update.

```
require 'rubygems'
require 'twitter'

twitter = Twitter::Base.new( 'your_username',
  'your_password')
twitter.timeline(:friends).each do |st|
  puts "#{st.user.name} says #{st.text}"
end
```

More information can be found at the Twitter gem homepage.

<http://twitter.rubyforge.org/>

## ***Exercises***

### **Section 2 - cURL Exercise**

Use the cURL command to explore the Twitter API further. Research and perform the following tasks. All information required to perform these tasks has been presented in this document and in the Twitter REST API documentation at:

<http://apiwiki.twitter.com/REST+API+Documentation>.

- Add the user 'aboutruby' to your friends list.
- View the status update with the ID of 1374254546.
- Post a new status update as a reply to the above.
- Add the status update with the ID of 1374254546 to your favorites.

### Section 3 - Twitter XML Exercise

Write a Ruby program that fetches the status update with the ID of 1374254546 and displays the following information.

- The text of the update.
- The screen name and ID of the user that posted the update.
- The date and time the update was created.
- The Homepage URL of the user that posted the update.

### Section 4 - Twitter Exercise

It's time to write a practical Twitter application. Write separate scripts that achieve the following.

- Retrieve a list of people following you on Twitter and a list of people you follow. For every user following you that you aren't yet following, follow that user. Print a list of users you've just added to your friends. Bonus points for making a "blacklist," people who are following you that you do not wish to follow. This blacklist should be stored in a file and not hard-coded.
- Periodically retrieve your friends timeline. For every update that includes the keyword "ruby," retweet that update. A "retweet" is simply copying that update and adding "RT @username" to the beginning. For example, if about ruby tweets "I like Ruby" and you want to retweet that, you would post an update that says "RT @aboutruby I like Ruby". Bonus points if you can detect if the retweet is going to be too long, and add an ellipsis (a ...) at the end.

## About the Author

Michael Morin is a life-long computer programmer and computer hobbyist. His fascination with programming started at an early age with BASIC on a Commodore 64 computer. No day has been complete without tinkering with programs on the computer.

He discovered Ruby when it was relatively unknown. Seeing it was a perfect fit for his programming style, he hasn't turned back.

In 2007, he became the webmaster of The About.com Guide to Ruby at <http://ruby.about.com/> and has been writing articles there ever since.