

# The busy Java developer's guide to Scala: Scala + Twitter = Scitter

With Scala, you can network in a more "social" manner

Skill Level: Introductory

[Ted Neward \(ted@tedneward.com\)](mailto:ted@tedneward.com)  
Principal  
ThoughtWorks

05 May 2009

Scala is fun to talk about in the abstract, but for most of the readers of this column, using it in a practical way makes the difference between seeing it as a "toy" and using it on the job. In this installment, Ted Neward uses Scala to build the basic framework for a client library for accessing Twitter, a popular micro-blogging system.

Twitter has taken the Internet by storm. As you undoubtedly know, this nifty social-netowrking tool allows subscribers to offer brief status updates about themselves and their current doings. Followers receive updates to their "Twitter feed" in much the same way that blogs generate updates into a blog readers' feed.

## About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this [series](#), you'll learn what all the recent hype is about and see some of Scala's linguistic capabilities in action. Scala code and Java™ code will be shown side by side wherever comparison is relevant, but (as you'll discover) many things in Scala have no direct correlation to anything you've found in Java — and therein lies much of Scala's charm! After all, if Java could do it, why bother learning Scala?

In and of itself, Twitter is an interesting discussion of social networking and a new generation of users being "highly connected," complete with all the pros and cons you would think would be associated with that.

Because Twitter published its API early on, numerous Twitter client applications have exploded onto the Internet. Because that API is fundamentally based on a pretty straightforward and easy-to-understand basis, numerous developers have found it educational to build a Twitter client of their own in much the same way that developers learning Web technologies built their own blog servers as an educational exercise.

Given Scala's functional nature (which would seem to align well with Twitter's RESTful nature) and extremely good XML processing features, it seemed to me to be a good experiment to build a Scala client library for accessing Twitter.

## Twitter what?

Before we get too far into this one, if you haven't looked at the Twitter API (or used the technology), let's do that now.

Put simply, a Twitter is a "micro-blog" — a personalized feed of small statements about yourself, no more than 140 characters long, that anybody who wishes to "follow" will be able to receive either via Web updates, RSS, text message, and so on. (The 140 character limit is necessary only because text messages, one of Twitter's primary source channels, are similarly constrained.)

### What is mostly RESTful?

Some readers will be curious about my use of the phrase *mostly RESTful*; it deserves a bit of explanation. "The Twitter API attempts to conform to the design principles of Representational State Transfer (REST)." And to a large degree, it does. Roy Fielding, the originator of the idea, might disagree with Twitter's use of the term, but to be honest, Twitter's approach will fit most people's definition of REST. To be really honest, I just want to avoid angry commentary about what REST is. Hence, I'm using the qualifier "mostly."

From a programmatic standpoint, Twitter is a *mostly RESTful* API in that you use some kind of message format — XML, ATOM, RSS or JSON — to send and receive messages from the Twitter servers. The different URLs, combined with the different messages and their required and optional message parts, make the different API calls. For example, if you want to receive the complete list of all "Tweets" (Twitter updates) from everybody on Twitter (also known as the "public timeline"), you prepare either an XML, ATOM, RSS, or JSON message, send it to the appropriate URL, and consume the result in the same format as documented on the Twitter Web site, [apiwiki.twitter.com](http://apiwiki.twitter.com):

---

### **public\_timeline**

**Returns the 20 most recent statuses from non-protected users who have set a custom user icon. Does not require authentication.**

Note that the public timeline is cached for 60 seconds so requesting it more often than that is a waste of resources.

**URL:** `http://twitter.com/statuses/public_timeline.format`

**Formats:** xml, json, rss, atom

**Method(s):** GET

**API limit:** Not applicable

**Returns:** list of status elements

-----

From a programming perspective, this means that we sent a simple GET HTTP request to the Twitter server and we'll get back a list of "status" messages wrapped up in either an XML, RSS, ATOM, or JSON message. A "status" message is defined on the Twitter site as something that looks vaguely like Listing 1:

### Listing 1. Hey world, where you at?

```
<feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
  <title>Twitter / tedneward</title>
  <id>tag:twitter.com,2007:Status</id>
  <link type="text/html" rel="alternate" href="http://twitter.com/tedneward"/>
  <updated>2009-03-07T13:48:31+00:00</updated>
  <subtitle>Twitter updates from Ted Neward / tedneward.</subtitle>
  <entry>
    <title>tedneward: @kdellison Happens to the best of us...</title>
    <content type="html">tedneward: @kdellison Happens to the best of us...</content>
    <id>tag:twitter.com,2007:http://twitter.com/tedneward/statuses/1292396349</id>
    <published>2009-03-07T11:07:18+00:00</published>
    <updated>2009-03-07T11:07:18+00:00</updated>
    <link type="text/html" rel="alternate"
      href="http://twitter.com/tedneward/statuses/1292396349"/>
    <link type="image/png" rel="image"
      href="http://s3.amazonaws.com/twitter_production/profile_images/
        55857457/javapolis_normal.png"/>
    <author>
      <name>Ted Neward</name>
      <uri>http://www.tedneward.com</uri>
    </author>
  </entry>
</feed>
```

Most of the elements (if not all of them) in the status message are pretty straightforward, so we'll leave that to the imagination.

Because we can consume Twitter messages in one of three XML-based formats and because Scala has some very powerful XML features, including XML literals and XPath-like query syntax APIs, writing a Scala library that can send and receive Twitter messages is an exercise in some basic Scala coding. For example, using Scala to consume the Listing 1 message to pick out the title or content of the status update can make use of Scala's XML types and the `\` and `\\` methods, shown in Listing 2:

### Listing 2. Hey Ted, where you at?

```

<![CDATA[
package com.tedneward.scitter.test
{
  class ScitterTest
  {
    import org.junit._, Assert._

    @Test def simpleAtomParse =
    {
      val atom =
        <feed xml:lang="en-US" xmlns="http://www.w3.org/2005/Atom">
          <title>Twitter / tedneward</title>
          <id>tag:twitter.com,2007:Status</id>
          <link type="text/html" rel="alternate" href="http://twitter.com/tedneward"/>
          <updated>2009-03-07T13:48:31+00:00</updated>
          <subtitle>Twitter updates from Ted Neward / tedneward.</subtitle>
          <entry>
            <title>tedneward: @kdellison Happens to the best of us...</title>
            <content type="html">tedneward: @kdellison
              Happens to the best of us...</content>
            <id>tag:twitter.com,2007:
              http://twitter.com/tedneward/statuses/1292396349</id>
            <published>2009-03-07T11:07:18+00:00</published>
            <updated>2009-03-07T11:07:18+00:00</updated>
            <link type="text/html" rel="alternate"
              href="http://twitter.com/tedneward/statuses/1292396349"/>
            <link type="image/png" rel="image"
              href="http://s3.amazonaws.com/twitter_production/profile_images/
              55857457/javapolis_normal.png"/>
            <author>
              <name>Ted Neward</name>
              <uri>http://www.tedneward.com</uri>
            </author>
          </entry>
        </feed>

      assertEquals(atom \\ "entry" \\ "title",
        "tedneward: @kdellison Happens to the best of us...")
    }
  }
}]]>

```

For some more details on Scala's XML support, check out "Scala and XML" (see [Resources](#)).

Consuming raw XML by itself is really not a fun exercise. If Scala is designed to make our lives easier, let's let it make our lives easier by creating a class or collection of classes specifically designed to make sending and receiving Scala messages easier. As a goal to go along with this, the library should be easily consumable from a "normal" Java program (which means it'll be easily accessible from anything that understands normal Java semantics, such as Groovy or Clojure).

## API design

Before we get too deeply into the API design for the Scala/Twitter library (which I will call "Scitter" as suggested to me by my fellow ThoughtWorker Neal Ford), there's a

couple of requirements that need to be hashed out.

First, clearly Scitter is going to have some kind of dependency on network access — and by extension, the Twitter servers — which will make it awkward to test.

Second, we're going to need to parse (and test) the various formats that Twitter sends back.

Third, we'll want to hide the differences between the various formats behind the API so that the client doesn't need to worry about the documented Twitter message formats but can instead just work with standard classes.

And last, because Twitter relies on the "authenticated user" for a large number of the APIs, the Scitter library will need to accommodate the difference between "authenticated" and "non-authenticated" APIs as appropriate, without making things too complicated.

The network access will require some form of HTTP communication in order to contact the Twitter servers. While we could use the Java library itself (specifically the `URL` class and its brethren), because the Twitter APIs require a lot of request and response body connection, it'll turn out to be easier to use a different HTTP API, specifically the Apache Commons `HttpClient` library. In order to make it easier to test the client API, the actual communications will be buried behind some APIs internally to the Scitter library so that not only would it be easier to swap out for another HTTP library (not that I can imagine why that would be necessary), but so that it will be easier to mock the actual network communication for easier testing (which is pretty easy to imagine the need for).

As a result, the first test is to simply Scala-fy the `HttpClient` calls to make sure we've got a basic communication pattern in place; note that because `HttpClient` depends on two other Apache libraries (Commons Logging and Commons Codec), both of those libraries will also need to be present during runtime; for those readers looking to develop similar kinds of code, make sure all three libraries are present on the classpath.

Because the easiest Twitter API to use is the test API which is documented to read

*Returns the string "ok" in the requested format with a 200 OK HTTP status code.*

Let's use that as the opening salvo in the Scitter exploration tests. It's documented as living at the URL `http://twitter.com/help/test.format` (where "format" is either "xml" or "json"; we'll choose to use "xml" for now) and the only supported HTTP method is `GET`. The `HttpClient` code thus pretty much writes itself, as in Listing 3:

### Listing 3. Twitter PING!

```
package com.tedneward.scitter.test
{
  class ExplorationTests
  {
    // ...

    import org.apache.commons.httpclient._, methods._, params._, cookie._

    @Test def callTwitterTest =
    {
      val testURL = "http://twitter.com/help/test.xml"

      // HttpClient API 101
      val client = new HttpClient()
      val method = new GetMethod(testURL)

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpMethodRetryHandler(3, false))

      client.executeMethod(method)

      val statusLine = method.getStatusLine()

      assertEquals(statusLine.getStatusCode(), 200)
      assertEquals(statusLine.getReasonPhrase(), "OK")
    }
  }
}
```

The lions' share of this code is the `HttpClient` boilerplate — interested readers should check out the `HttpClient` API docs for details. Assuming that a network connection to the public Internet is available when run (and that Twitter hasn't changed its public API), this test should pass with flying colors.

Given that, let's hash out the first part of the Scitter client. This means we're smack up against a design point: How to structure the Scitter clients to handle authenticated-vs-non-authenticated calls. For the moment, I'll do it in a classic Scala fashion, assuming authentication to be a "per-object" quality and thus putting the authentication-required calls into a class definition and the non-authenticating calls into an object definition:

#### Listing 4. Scitter.test

```
package com.tedneward.scitter
{
  /**
   * Object for consuming "non-specific" Twitter feeds, such as the public timeline.
   * Use this to do non-authenticated requests of Twitter feeds.
   */
  object Scitter
  {
    import org.apache.commons.httpclient._, methods._, params._, cookie._

    /**
     * Ping the server to see if it's up and running.
     *
     * Twitter docs say:
     * test
     * Returns the string "ok" in the requested format with a 200 OK HTTP status code.
     * URL: http://twitter.com/help/test.format
     */
  }
}
```

```

    * Formats: xml, json
    * Method(s): GET
    */
    def test : Boolean =
    {
        val client = new HttpClient()

        val method = new GetMethod("http://twitter.com/help/test.xml")

        method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
            new DefaultHttpClientRetryHandler(3, false))

        client.executeMethod(method)

        val statusLine = method.getStatusLine()
        statusLine.getStatusCode() == 200
    }
}
/**
 * Class for consuming "authenticated user" Twitter APIs. Each instance is
 * thus "tied" to a particular authenticated user on Twitter, and will
 * behave accordingly (according to the Twitter API documentation).
 */
class Scitter(username : String, password : String)
{
}
}

```

For now, let's leave the network-abstraction alone — that will be a refinement added later when offline testing becomes more important. It'll also help avoid "overabstracting" the network communication away when we get a better idea of exactly how we're using the HttpClient class(es).

Because we've made the distinction between authenticated and non-authenticated Twitter clients, let's quickly create an authenticated method as well. It turns out that Twitter has an API for verifying the login credentials of a user which is about as close to an authenticated ping as we're going to get. Again, the HttpClient code will be similar to the previous code with the exception that now we have to pass the username and password in to the Twitter API as well.

This brings us to the notion of how Twitter authenticates users. After some quick research on the Twitter API page, it turns out that Twitter uses a stock HTTP authentication approach, same as any authenticated resource does within HTTP. This means the HttpClient code must provide username and password as part of the HTTP request, not as a POSTed body as might be expected, shown in Listing 5:

### Listing 5. Hey Twitter, it's me!

```

package com.tedneward.scitter.test
{
    class ExplorationTests
    {
        def testUser = "TwitterUser"
        def testPassword = "TwitterPassword"

        @Test def verifyCreds =
        {

```

```

    val client = new HttpClient()

    val verifyCredsURL = "http://twitter.com/account/verify_credentials.xml"
    val method = new GetMethod(verifyCredsURL)

    method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpClientRetryHandler(3, false))

    client.getParams().setAuthenticationPreemptive(true)
    val defaultcreds = new UsernamePasswordCredentials(testUser, testPassword)
    client.getState().setCredentials(new AuthScope("twitter.com", 80,
        AuthScope.ANY_REALM), defaultcreds)

    client.executeMethod(method)

    val statusLine = method.getStatusLine()

    assertEquals(200, statusLine.getStatusCode())
    assertEquals("OK", statusLine.getReasonPhrase())
  }
}
}

```

Note that in order for this test to pass, the username and password fields will need to be populated with something that Twitter accepts — I used my own Twitter username and password during development, but obviously you'll have to use one of your own. Registering for a new Twitter account is pretty simple, so I'll assume you either have one or can figure out how to get one. (It's OK ... I'll wait while you do it.)

Once that works, it's pretty easy to see how this will map into the Scitter class itself using the username and password constructor parameters, shown in Listing 6:

### Listing 6. Scitter.verifyCredentials

```

package com.tedneward.scitter
{
  import org.apache.commons.httpclient._, auth._, methods._, params._

  // ...

  /**
   * Class for consuming "authenticated user" Twitter APIs. Each instance is
   * thus "tied" to a particular authenticated user on Twitter, and will
   * behave accordingly (according to the Twitter API documentation).
   */
  class Scitter(username : String, password : String)
  {
    /**
     * Verify the user credentials against Twitter.
     *
     * Twitter docs say:
     * verify_credentials
     * Returns an HTTP 200 OK response code and a representation of the
     * requesting user if authentication was successful; returns a 401 status
     * code and an error message if not. Use this method to test if supplied
     * user credentials are valid.
     * URL: http://twitter.com/account/verify_credentials.format
     * Formats: xml, json
     * Method(s): GET
     */
    def verifyCredentials : Boolean =

```

```

    {
      val client = new HttpClient()

      val method = new GetMethod("http://twitter.com/help/test.xml")

      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpClientRetryHandler(3, false))

      client.getParams().setAuthenticationPreemptive(true)
      val creds = new UsernamePasswordCredentials(username, password)
      client.getState().setCredentials(
        new AuthScope("twitter.com", 80, AuthScope.ANY_REALM), creds)

      client.executeMethod(method)

      val statusLine = method.getStatusLine()
      statusLine.getStatusCode() == 200
    }
  }
}

```

And the corresponding Scitter class test in Listing 7 is pretty simple too:

### Listing 7. Testing Scitter.verifyCredentials

```

package com.tedneward.scitter.test
{
  class ScitterTests
  {
    import org.junit._, Assert._
    import com.tedneward.scitter._

    def testUser = "TwitterUsername"
    def testPassword = "TwitterPassword"

    // ...

    @Test def verifyCreds =
    {
      val scitter = new Scitter(testUser, testPassword)
      val result = scitter.verifyCredentials
      assertTrue(result)
    }
  }
}

```

Not bad, not bad. The basic structure of the library is beginning to take shape, though clearly there's a fairly long way to go, particularly because nothing done so far has really been all that Scala-specific — building the library has been more an exercise in object-oriented design than anything else. So let's start consuming some XML and handing it back in a more palatable form.

## From XML to objects

The easiest API to add at the moment is the `public_timeline` one, which collects the most recent  $n$  updates that Twitter has received across all users and hands them back for use. Unlike the other two APIs seen so far, the `public_timeline` API hands

back a response body (rather than relying just on status codes), so we'll need to crack the resulting XML/RSS/ATOM/whatever apart before handing it back to the Scitter client.

In keeping with the progression so far, let's write an exploration test that hits the public feed and dumps the results to `stdout` for examination, shown in Listing 8:

### Listing 8. What's everybody up to?

```
package com.tedneward.scitter.test
{
  class ExplorationTests
  {
    // ...

    @Test def callTwitterPublicTimeline =
    {
      val publicFeedURL = "http://twitter.com/statuses/public_timeline.xml"

      // HttpClient API 101
      val client = new HttpClient()
      val method = new GetMethod(publicFeedURL)
      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpClientRetryHandler(3, false))

      client.executeMethod(method)

      val statusLine = method.getStatusLine()
      assertEquals(statusLine.getStatusCode(), 200)
      assertEquals(statusLine.getReasonPhrase(), "OK")

      val responseBody = method.getResponseBodyAsString()
      System.out.println("callTwitterPublicTimeline got... ")
      System.out.println(responseBody)
    }
  }
}
```

When run, the results will differ every time thanks to the huge number of users on the public Twitter servers, but it will generally look something like Listing 9 in the JUnit text file dump:

### Listing 9. These are the Tweets of Our Lives

```
<statuses type="array">
  <status>
    <created_at>Tue Mar 10 03:14:54 +0000 2009</created_at>
    <id>1303777336</id>
    <text>She really is. http://tinyurl.com/d65hmj</text>
    <source><a href="http://iconfactory.com/software/twiterrific">twiterrific</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>
    <user>
      <id>18729101</id>
      <name>Brittanie</name>
      <screen_name>brittaniemarie</screen_name>
```

```

    <description>I'm a bright character. I suppose.</description>
    <location>Atlanta or Philly.</location>
    <profile_image_url>http://s3.amazonaws.com/twitter_production/profile_images/
      81636505/goodish_normal.jpg</profile_image_url>
    <url>http://writeitdowntakeapicture.blogspot.com</url>
    <protected>>false</protected>
    <followers_count>61</followers_count>
  </user>
</status>
<status>
  <created_at>Tue Mar 10 03:14:57 +0000 2009</created_at>
  <id>1303777334</id>
  <text>Number 2 of my four life principles. "Life is fun and rewarding"</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>
  <user>
    <id>21465465</id>
    <name>Dale Greenwood</name>
    <screen_name>Greendale</screen_name>
    <description>Vegetarian. Eat and use only organics.
      Love helping people become prosperous</description>
    <location>Melbourne Australia</location>
    <profile_image_url>http://s3.amazonaws.com/twitter_production/profile_images/
      90659576/Dock_normal.jpg</profile_image_url>
    <url>http://www.4abundance.mionegroup.com</url>
    <protected>>false</protected>
    <followers_count>15</followers_count>
  </user>
</status>
  (A lot more have been snipped)
</statuses>

```

It's fairly obvious from examining both the results and the Twitter docs that the result of the call is a mostly-flat collection of "status" messages with a consistent message structure. Picking the results apart using Scala's XML support is a pretty simple exercise, but we'll refine it as soon as we get the basic tests to pass, as in Listing 10:

### Listing 10. What's everybody up to?

```

package com.tedneward.scitter.test
{
  class ExplorationTests
  {
    // ...

    @Test def simplePublicFeedPullAndParse =
    {
      val publicFeedURL = "http://twitter.com/statuses/public_timeline.xml"

      // HttpClient API 101
      val client = new HttpClient()
      val method = new GetMethod(publicFeedURL)
      method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
        new DefaultHttpClientRetryHandler(3, false))
      val statusCode = client.executeMethod(method)
      val responseBody = new String(method.getResponseBody())

      val responseXML = scala.xml.XML.loadString(responseBody)
      val statuses = responseXML \\ "status"
    }
  }
}

```

```

for (n <- statuses.elements)
{
  n match
  {
    case <status>{ contents @ _* }</status> =>
    {
      System.out.println("Status: ")
      contents.foreach((c) =>
      {
        c match
        {
          case <text>{ t @ _* }</text> =>
            System.out.println("\tText: " + t.text.trim)
          case <user>{ contents2 @ _* }</user> =>
            {
              contents2.foreach((c2) =>
              {
                c2 match
                {
                  case <screen_name>{ u }</screen_name> =>
                    System.out.println("\tUser: " + u.text.trim)
                  case _ =>
                    ()
                }
              }
            )
          }
        }
      }
    }
    case _ =>
      () // or, if you prefer, System.out.println("Unrecognized element!")
  }
}
}
}
}
}

```

As example code patterns go, this is hardly encouraging — it feels rather DOM-like, navigating to each child element one at a time, extracting the text, then navigating down another node. We could simply do two XPath-style queries, something like Listing 11:

### Listing 11. Alternative parsing approach

```

for (n <- statuses.elements)
{
  val text = (n \\ "text").text
  val screenName = (n \\ "user" \ "screen_name").text
}

```

Which would certainly be shorter, but two basic problems arise from this:

- We could be forcing Scala's XML library to traverse the graph once for each element or sub-element desired, which could get slow over time.
- We're still wrestling with the structure of the XML message directly. This is the far more important of the two.

This isn't going to scale, so to speak — assuming we eventually will have interest in every element in the Twitter status message, we'll have to extract each element from each status individually.

Which leads to another issue, that of the individual formats themselves. Remember, Twitter has four different formats they like to use and we don't want Scitter clients to have to know the difference between any of them, so Scitter needs an intermediate structure that we can deliver back to the client for further use, something along the lines of Listing 12:

### Listing 12. Breaker, breaker, what's your status?

```
abstract class Status
{
  val createdAt : String
  val id : Long
  val text : String
  val source : String
  val truncated : Boolean
  val inReplyToStatusId : Option[Long]
  val inReplyToUserId : Option[Long]
  val favorited : Boolean
  val user : User
}
```

... and a similar-looking one for `User`, which I'll not repeat here for reasons of brevity. Note that the `User` child element has an interesting caveat to it — while there is a Twitter user type, it has an optional "most recent status" nested inside of it. Status messages also have a user nested inside of them. In this case, in order to help avoid some of the potential recursion problems, I'm choosing to create a `User` type that's nested inside of the `Status` type to reflect the `User` data that appears there and vice versa for `Status` nested inside of `User`, so as to specifically avoid that problem. (At least, that's the plan until I discover a problem with it.)

Now, having created the object type representing the Twitter messages, we can follow a common Scala pattern of XML deserialization: create a corresponding object definition that contains a `fromXml` method to take an XML node and rip it apart into a object instance, shown in Listing 13:

### Listing 13. Cracking XML

```
/**
 * Object wrapper for transforming (format) into Status instances.
 */
object Status
{
  def fromXml(node : scala.xml.Node) : Status =
  {
    new Status {
      val createdAt = (node \ "created_at").text
      val id = (node \ "id").text.toLong
      val text = (node \ "text").text
    }
  }
}
```

```

    val source = (node \ "source").text
    val truncated = (node \ "truncated").text.toBoolean
    val inReplyToStatusId =
      if ((node \ "in_reply_to_status_id").text != "")
        Some((node \ "in_reply_to_status_id").text.toLong)
      else
        None
    val inReplyToUserId =
      if ((node \ "in_reply_to_user_id").text != "")
        Some((node \ "in_reply_to_user_id").text.toLong)
      else
        None
    val favorited = (node \ "favorited").text.toBoolean
    val user = User.fromXml((node \ "user")(0))
  }
}
}

```

The beautiful thing about this particular idiom is that it can be extended for any of the other formats Twitter supports — either the `fromXml` method itself could check to see if the node holds an XML, RSS, or Atom type of content before picking it apart, or `Status` can have `fromXml`, `fromRss`, `fromAtom`, and `fromJson` methods. The latter approach is actually my preference because then it treats the XML-based and JSON (text-based) formats equally.

Curious and observant readers will notice that in the `fromXml` methods of both `Status` and its nested `User` I'm using the XPath-style decomposition approach instead of walking through the nested elements as I suggested earlier. For now, the XPath-style approach seems to read easier, but fortunately should I change my mind later, good ole' encapsulation remains my friend — I can change it over later without anybody outside of Scitter knowing or caring.

Notice how two of the members inside `Status` use the `Option[T]` type; this is because those elements are frequently left out of the `Status` message and while the elements themselves will appear, they will appear empty (like in `<in_reply_to_user_id></in_reply_to_user_id>`). This is precisely what `Option[T]` was made to represent, so when empty, they will get "None" as a value. (This means that for Java-based compatibility, it'll be just a touch more difficult to access, but only by calling `get()` on the resulting `Option` instance, which isn't too onerous and neatly deals with the "nulls-or-0" issue that would otherwise arise.)

So now, consuming the public timeline becomes a simple matter of:

#### Listing 14. Cracking the public timeline

```

@Test def simplePublicFeedPullAndDeserialize =
{
  val publicFeedURL = "http://twitter.com/statuses/public_timeline.xml"

  // HttpClient API 101
  val client = new HttpClient()
  val method = new GetMethod(publicFeedURL)
  method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,

```

```

    new DefaultHttpMethodRetryHandler(3, false))
    val statusCode = client.executeMethod(method)
    val responseBody = new String(method.getResponseBody())

    val responseXML = scala.xml.XML.loadString(responseBody)
    val statuses = responseXML \\ "status"

    for (n <- statuses.elements)
    {
        val s = Status.fromXml(n)
        System.out.println("\t'" + s.user.screenName + "' wrote " + s.text)
    }
}

```

... which, quite frankly, looks a lot cleaner and easier to use.

Putting this all together in the Scitter singleton is a simple matter of doing the query, parsing the individual `Status` elements out, and appending them together into a `List[Status]` instance to hand back, shown in Listing 15:

### Listing 15. Scitter.publicTimeline

```

package com.tedneward.scitter
{
    import org.apache.commons.httpclient._, auth._, methods._, params._
    import scala.xml._

    object Scitter
    {
        // ...

        /**
         * Query the public timeline for the most recent statuses.
         *
         * Twitter docs say:
         * public_timeline
         * Returns the 20 most recent statuses from non-protected users who have set
         * a custom user icon. Does not require authentication. Note that the
         * public timeline is cached for 60 seconds so requesting it more often than
         * that is a waste of resources.
         * URL: http://twitter.com/statuses/public_timeline.format
         * Formats: xml, json, rss, atom
         * Method(s): GET
         * API limit: Not applicable
         * Returns: list of status elements
         */
        def publicTimeline : List[Status] =
        {
            import scala.collection.mutable.ListBuffer

            val client = new HttpClient()

            val method =
                new GetMethod("http://twitter.com/statuses/public_timeline.xml")

            method.getParams().setParameter(HttpMethodParams.RETRY_HANDLER,
                new DefaultHttpMethodRetryHandler(3, false))

            client.executeMethod(method)

            val statusLine = method.getStatusLine()
            if (statusLine.getStatusCode() == 200)
            {

```

```
val responseXML =
  XML.loadString(method.getResponseBodyAsString())

val statusListBuffer = new ListBuffer[Status]

for (n <- (responseXML \\ "status").elements)
  statusListBuffer += (Status.fromXml(n))

  statusListBuffer.toList
}
else
{
  Nil
}
}
}
```

Not bad for an hour's worth of work. We obviously have some distance to go before we have a fully-functional Twitter client, but so far, the basic behaviors look good.

## Conclusion

We're well on our way to building the Scitter library; so far, things are simple and pretty clean from the outside as evidenced by the relatively easy Scitter test implementations, particularly when compared against the exploration tests that gave birth to the Scitter API itself. Outside users don't have to worry about the intricacies of the Twitter API or its various formats and although right now the Scitter library is a bit touchy to test (depending on the network is not a good recipe for unit tests), we'll fix that in time.

Note that I've deliberately kept an object-oriented feel to the Twitter API, keeping with the spirit of Scala — just because Scala supports a number of functional features doesn't mean we have to give up on the object design approaches that the Java structure embraces. We'll take in the functional features where it makes sense but keep the "old ways" alive where they seem to fit.

Which isn't to say that the design I'm putting in place here is the best way to approach the problem, only that this is the way I've decided to design it; and because I'm the one writing the article, we do it my way. Don't like it, write your own library and article (and send me the URL so I can put in a shout-out to you in a future column). In fact, in a future article, we'll package all this up into a Scala "sbaz" package and put it out on the Web someplace for easy downloading.

For now, it's time for us to part ways again. Next month we'll add some more interesting features to the Scitter library and start thinking about ways to make it more easy to test and use.

# Resources

## Learn

- ["The busy Java developer's guide to Scala"](#) (Ted Neward, developerWorks): Read the complete series.
- ["Scala and XML"](#) (Michael Galpin, developerWorks, April 2008) shows how Scala can make working with XML a joy.
- ["Functional programming in the Java language"](#) (Abhijit Belapurkar, developerWorks, July 2004): Explains the benefits and uses of functional programming from a Java developer's perspective.
- ["Scala by Example"](#) (Martin Odersky, December 2007): A short, code-driven introduction to Scala (in PDF).
- ["Programming in Scala"](#) (Martin Odersky, Lex Spoon, and Bill Venner; Artima, December 2007): The first book-length introduction to Scala.
- [Bjarne Stroustrup](#): Designed and implemented C++, which he has described as "a better C."
- ["Java Puzzlers: Traps, Pitfalls, and Corner Cases"](#) (Addison-Wesley Professional, July 2005) reveals oddities of the Java programming language through entertaining and thought-provoking programming puzzles.
- The [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

## Get products and technologies

- The [Jakarta \(Apache\) Commons HttpClient component](#) provides an efficient, up-to-date, feature-rich package implementing the client side of the most recent HTTP standards and recommendations. You may also want the [Commons Logging](#) and [Commons Codec](#) components.
- [Download Scala](#): Start learning it with this series!
- [SUnit](#): Part of the standard Scala distribution, in the *scala.testing* package.

## Discuss

- [developerWorks blogs](#): Get involved in the [developerWorks community](#).

## About the author

Ted Neward

Ted Neward is a Principal Consultant at ThoughtWorks, a global consultancy firm,

where he consults, mentors, and presents on Java, .NET, XML services, and other platforms. He resides near Seattle, Washington.

## Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.