Building a Web Spider with Ruby

Spider on the Web

Ruby is a very elegant language, and it's harmonious – the parts work together effectively. Ruby also significantly reduces a developer's burden. We'll show you how to use Ruby to build a quick and simple web spider application. BY DAVID BERUBE

www.spidertim.com

Ruby is a scripting language developed by Yukihiro Matsumoto and released under the GPL. The Ruby language has an excellent set of string manipulation and networking libraries, making it a great choice for writing web spiders. If you are not familiar with web spiders, they are programs designed to automatically traverse the web. Search engines use web spiders to add web pages to their index; companies like Netcraft use spiders to get statistics on web servers.

You can use a web spider to find information automatically from almost any website; in this article, we'll discuss how to use Ruby to retrieve information from LiveJournal, a popular weblog provider. You can extend these techniques to virtually every website that provides public information.

Getting Web Data

Programmers have several ways of retrieving information from a website. Some sites provide information in an easy-to-process format, like XML. A good example of this is what's called RSS, which is an XML format. RSS stands for Really Simple Syndication, and it's a way for news sites to provide up-to-the-minute headlines to other sites and to programs designed to display the information. A large number of sites provide their headlines in RSS format; BBC provides all of its sections in RSS. Many programs deal with RSS headlines; Mozilla Firefox, for example, lets you create "live bookmark" folders that automatically fill with RSS headlines.

Another method for retrieving information is so called "screen scraping" – downloading the HTML page just like a web browser does and then stripping out just the sections that we are interested in. LiveJournal provides some information – a list of a user's posts, for example – in RSS XML format, and they request that users use that format whenever possible. Other information can only be retrieved by screen scraping: a user's friends list, for example.

Because LiveJournal blogs contain a fairly large amount of information, you could mine the blogs for a wide variety of different kinds of data. In this example, we're going to write a program that compiles comments from a single LiveJournal user; that is, the program will download every comment the user has made in any of his friends' journals and display the comments. Of course, it won't find messages left in blogs that are not on the user's friends list, nor will it find private messages.

Web spiders can only find publicly accessible information, so this application will only find information that the target user has already agreed to make public. Of course, that could still be a lot of information in some cases. It's easy to extend this technique for almost any type of public information available on the web.

Spider Plan

The program will take a LiveJournal user's name and download his friends

Listing 1: parse_user.rb

```
require 'net/http'

h = Net::HTTP.new('www.livejournal.com', 80)
friend_arr = []
person = ARGV[0]

resp, data = h.get("http://www.livejournal.com/userinfo.bml?user=#{person}",nil)

print "Friend list for #{person}\n"

data.split("\n").each do |line|
        line.split(",").each do |token|
                if token =~ /userinfo.bml\?user=([^'&]*)\'/
                        friend_arr.push $1
                        print "#$1\n"
                end
        end
end

print "\n"

friend_arr.each do |friend|

  print "Parsing #{friend}'s journal for #{person}'s comments...\n";
  f = File.new("#{person}_#{friend}.txt","w")
  f.puts `ruby parse_journal.rb #{friend} #{person}`
  f.close
end
```

list. The program will then retrieve a list of posts for the user and for all of the user's friends and download all of those posts. It will then parse the HTML files and print the comments made by our target user to STDOUT, separated by three dashes on a line.

The Tools We Have

What tools are available to us? The first is the Ruby net/http library, which comes with the Ruby distribution. The Ruby net/http library lets us access the HTTP protocol through an object oriented interface. In order to download webpages, we must first make a connection to the server by calling the *Net::HTTP.new* method, which takes a server address and a port number and returns a *Net::HTTP* object.

We can then call the *get* method on the object to retrieve a page. The *get* method returns an array; the first element in the array is a *HTTPResponse* object and the second element is a string containing the body of the result – in other words, the contents of the page we asked for. The *HTTPResponse* object contains the headers and also the body of the result. Both methods are included for compatibility reasons, as well as for simplicity – often, you are only interested in the data of a response and not in the HTTP status code.

Other powerful tools we can make use of are the *split* and *each* methods. The *split* method takes a string and splits it into an array, delimited by either a regular expression or a string; it is similar to the *split* function in Perl or PHP. Notice, however, how it can be chained with other methods, as it is in our example. This improves readability.

The *each* method takes an array and passes each element to a block of code. Incidentally, the block syntax is generic, and you can use this syntax in your own functions. The block of code is enclosed by a *do…end* pair, and the parameters are surrounded by pipe symbols. The *each* method is very similar to *foreach* methods in other languages – notice, however, that it is not a language construct, as it is in most languages, but a method, and as such, you could easily construct your own iterator.

Webspidering LiveJournal

We have two scripts involved in parsing a weblog. The first, *parse_user.rb* (see Listing 1), takes an argument on the command line: the user you wish to monitor. It then downloads the list of people that the user has marked as friends, on the theory that he is most likely to comment on the blogs of those users; if you wish, you could search the friends of those friends, and their friends, and so on. It repeatedly calls the second script, *parse_journal.rb* (see Listing 2), once for each friend.

The second script takes two arguments: which blog to parse, and which user's comments we are looking for. The second script can also be called by itself, should you only want to search a single user's blog for comments by another user.

The first script retrieves a list of friends by downloading the LiveJournal "user information" page for that user; the user information page is a webpage that contains a user's profile, and, importantly for us, a link to the user information page of each of his friends.

The script searches for those links, takes the user name for each of his friends, and adds the friends to an array. When it's done, it then calls the second script for each of those friends, telling the second script to retrieve every comment by the target user in the friend's blog. It also takes the output of the second script and writes it to a file for easy access.

The second script first downloads the RSS newsfeed for the user passed to it – conveniently, this contains, among other things, a link to each of the user's posts. It retrieves all of these links and places them in an array. It then iterates over the array, grabbing the page for each one, and then iterating over each line of each page.

The line processing is split into two parts. First, the script waits until it finds a post by the appropriate user, signaled by a link to use his information page. At that point, the next line resembling a post is presumed to be a comment by that user; the line is detected, then the comment and the link are extracted.

The comment is transformed to be more user friendly; the sequence *<br />* is transformed into newlines, and HTML tags, and other unimportant information is stripped out. After this, the comment is printed out, and the script starts looking for another comment by the target user.

As you can imagine, this whole process is extremely sensitive to the exact format of the page, and this illustrates one of the basic problems with screen scraping; since the format of the data can and does change, you occasionally have to update your spider to keep it working. In this case, changes to the format of the LiveJournal can require changes to the spider for it to properly process a blog. Notice, however, that the spider is generally easy to modify when changes occur.

When Things Go Wrong

Errors can occur in a number of places while developing and using a spider. Most often, though, errors crop up in two places: one, the networking code, and two, the screen scraping code. The networking code can have problems, since the Internet is inherently unreliable: pages move, machines crash, requests time out, and so forth. In order to catch these types of errors, you can get a wealth of information from the *HTTPResponse* object returned by the *Net:HTTP.get* function.

Since our example only retrieves a limited set of URLs, errors such as moved pages are less likely. If they were to occur, they would likely signal a change in LiveJournal's hierarchy and require a rewrite. However, for many other spiders, this is not true, and it would be wise to plan for contingencies. What should your spider do, for example, if the server returns an *HTTPRequestTimeOut* error? You may wish to try downloading again; alternatively, you might wish to ignore the error and take some other action.

The screen scraping code frequently causes problems, both during development and after develolpment. During development, it is wise to download a copy of the page you are attempting to parse and work on it from your local machine. This increases the speed of the development and allows you to focus on retrieving the appropriate pieces of the code you are interested in. It is also wise to use the least restrictive parsing possible; include as little information as you can in your regular expression and still be sure to get the information you seek. That way, your code will break less often because of changes to the target site.

Listing 2: parse_journal.rb

```
require 'net/http'

h = Net::HTTP.new('www.livejournal.com', 80)

url_array = []
person = ARGV[0]
watch_for = ARGV[1]

resp, data = h.get("http://www.livejournal.com/users/#{person}/data/rss", nil)

data.split("\n").each do |line|
  if line =~ /<comments>http:\/\/www.livejournal.com([^<]*)<\/comments>/
        url_array.push $1
  end
end

url_array.each { |url|
  response, data = h.get(url, nil)
  logging = false
  data.split("\n").each do |line|
        if line =~ /<a [^>]*href='http:\/\/www.livejournal.com\/users\/([^"]*)\/'><b>/
                if $1 == watch_for

                        print "\n---\n"

                        logging= true
                  else

                        logging = false
                  end

        end
        if line =~ /<a href='(http:\/\/www.livejournal.com\/users\/[^\/]*\/[^']*thread[^']*)'>.*<\/td><\/tr><tr><td>(.*)<p style='margin:/ and logging
                url = $1
                comment = $2

                comment.gsub!('<br />',"\n")

                comment.gsub!(/<\/*[^>]*>/,'')

                comment.gsub!('(Reply to this) (Parent)','')
                print "#{url}\n#{comment}"


                logging=false
        end
  end
}
```

Conclusion

Spiders are powerful, and Ruby is a great language to write them in. Of course, like most powerful tools, spiders must be used with care. Stories of badly behaved spiders overloading web servers with requests are common. You should ensure your spider does not cause problems for site administrators. In some cases, they may have posted instructions on how automated services may use their site; if so, take care to follow them. They are providing you with a free service, and they are not required to continue. ■

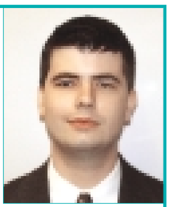
INFO

[1] Ruby Documentation *http://www.ruby-doc.org/*

[2] *Programming Ruby* – a free ruby eBook *http://www.rubycentral.com/book/*

[3] A well-documented web spider tutorial *http://www.searchlores.org/phpregexspider.htm*

THE AUTHOR



David Berube is a self-employed software developer, writer, and speaker. He typically develops using a blend of client-side Microsoft and server-side open source technologies – like Ruby. His article, "Databases and Dynamic Ruby Classes," appeared in the in the December 2004 Dr Dobb's Journal. You can read more of his articles at http://berubeconsulting.com/.