Ruby Advent 2008

Ruby Cool-aid for Holiday Season

HTML Scraping with scRUBYt! for Fun and Profit

Tue Dec 23 00:00:00 +0530 2008

Peter Szinek walks us through the process of scraping data from web sites with scRubyt!. Impress your friends (and even your mother!) this christmas with your slick data mining skillz!

Intro

Collecting data from various websites to analyze, compare, aggregate, mash up or otherwise transform and eventually display the data they contain in your own application (possibly augmenting them with your interpretation of the mined data set or other additional information) is a very common activity in (not only) web circles nowadays. Modern (web2.0? {ducks}) web sites are keeping up with the times, offering their API, RSS feed or other means to access their data. Adding as much semantic information as possible to web sites is commonplace these days, and it is becoming more and more important (probably the next big thing(tm)) - microformats is a good example of this direction with the goal to provide their data in an accessible format.

So why do I Need this Scraping Thing then?!?

Acquiring data from these kinds of sites is fairly straightforward - there is ‘only’ one catch: majority of the web sites out there in the wild are typically not offering their API (probably because they do not have one), most of them do not publish a feed and they might be not even well formatted. The reason behind this is pretty straightforward - their creators didn’t aim to enable (or downright try to avoid/block) automated data mining/exchange from their site. All they cared about is whether their site renders nicely in a web browser (preferably in all major variants, adding even more hacks, tricks and other cruft to the page source). Filtering out meaningful data from such a HTML mess is referred to as *web scraping* and in this article I’d like to show you how to do it with scRUBYt!, Ruby’s probably most popular web scraping framework.

Hello, world! err… Google

If you’d like to know more about web scraping in Ruby, check out my earlier, wildly popular article on the topic (even mentioned in “Learning Ruby” from O’Reilly). Otherwise, let’s dive into scRUBYt! straight away.

I recommend you to install scRUBYt! and follow along. You can find the installation instructions here.

So here is our first example - a Google scraper (not very fascinating I know - but it’s easier to explain the basics on a google scraper rather than on something more practical but complicated):

```
require 'rubygems'
require 'scrubyt'

google_data = Scrubyt::Extractor.define do
 fetch 'http://www.google.com/search?hl=en&q=ruby'

 link_title "//a[@class='l']"
end

puts google_data.to_xml
```

The output:

```
   <root>
    <link_title>Ruby Programming Language</link_title>
    <link_title>Ruby (programming language) - Wikipedia, the free encyclopedia</link_title>
    <link_title>Ruby - Wikipedia, the free encyclopedia</link_title>
    <link_title>Kaiser Chiefs - Ruby</link_title>
    <link_title>ruby</link_title>
    <link_title>Ruby - The Inspirational Weight Loss Journey on the Style Network</link_title>
    <link_title>Ruby on Rails</link_title>
    <link_title>Ruby Central</link_title>
    <link_title>Ruby&amp;#39;s Diner - rubys.com</link_title>
    <link_title>Ruby Annotation</link_title>
    <link_title>Welcome! [Ruby-Doc.org: Documenting the Ruby  Language]</link_title>
    <link_title>Bootsy Collins, Jeff Ruby open downtown club</link_title>
  </root>
```

Deciphering the Google example

A scRUBYt! extractor (like the above) has 2 main parts: **navigation** (fetching a new page, entering text into text fields, submitting forms, checking checkboxes etc) and once you arrive at the desired page, the **data scraper** itself.

Navigation is fairly obvious I guess (the other actions besides `fetch` - which should be always present as the first step - are `fill_textfield`, `fill_textarea`, `click_link`, `check_checkbox`, `check_radiobutton`, `select_option`, `submit` and if you can't submit the form automatically for some reason, `click_by_xpath` as the last resort.

Scraping is much more complicated, and therefore I'd like to speak more about it, with lot of examples and explanations. So let's get started.

scRUBYt! Patterns

In the above example, `link_title` is called a "pattern". A scRUBYt! pattern is to an HTML document what a regular expression pattern is to a string: similarly to a regexp, which matches and extracts a portion of the input string, a scRUBYt! pattern matches and extracts portions of the HTML input document (and beyond, as you will see in the next paragraph).

scRUBYt! patterns are hierarchical - let's say we have the following pattern tree:

```
book
 |
 +-+ author
  |   |
  |   +-- first_name
  |
  + price
```

then, the input of the 'book' pattern is the whole HTML document, the input of 'author' and 'price' patterns is whatever 'book' extracts, and in turn, the input of `'first_name'` is the output of 'author'. Sounds confusing? In practice it's not, and you get used to it after writing a scraper or two. Hopefully it will be clear even after finishing this article.

scRUBYt! patterns come in different sizes and shapes. In the above example, you have seen the most basic one, the *XPath pattern* in action. The XPath pattern returns the textual content of the elements matched by the supplied XPath.

Meet the Attribute Pattern

scRUBYt! comes with a handful of different pattern types - you have seen the XPath pattern already. Let's say we'd like to extract not just the titles of the search results, but the URLs they are pointing to as well. This is how we could go about it with an *attribute pattern*:

```
google_data = Scrubyt::Extractor.define do
fetch 'http://www.google.com/search?hl=en&q=ruby'

 link_title "//a[@class='l']" do
  url 'href', :type => :atribute
 end
end

puts google_data.to_xml
```

the result:

```
 <root>
  <link_title>
  <url>http://ruby-lang.org/</url>
  </link_title>
  <link_title>
  <url>http://en.wikipedia.org/wiki/Ruby_(programming_language)</url>
  </link_title>
  <link_title>
  <url>http://en.wikipedia.org/wiki/Ruby</url>
  </link_title>
  <link_title>
  <url>http://www.youtube.com/watch?v=JMDcOViViNY</url>
  </link_title>
  <link_title>
  <url>http://www.youtube.com/watch?v=d14511Amd08</url>
  </link_title>
  <link_title>
  <url>http://www.rubyonrails.org/</url>
  </link_title>
  <link_title>
  <url>http://www.mystyle.com/mystyle/shows/ruby/index.jsp</url>
  </link_title>
  <link_title>
  <url>http://www.rubys.com/</url>
  </link_title>
  <link_title>
  <url>http://www.rubycentral.org/</url>
  </link_title>
  <link_title>
  <url>http://www.w3.org/TR/ruby/</url>
  </link_title>
  <link_title>
  <url>http://www.ruby-doc.org/</url>
  </link_title>
  <link_title>
  <url>http://www.bizjournals.com/triad/stories/2008/12/15/daily71.html</url>
  </link_title>
</root>
```

Wait, Where is my Link Title?

By default, scRUBYt! outputs the results for the leaf patterns only, as usually the intermediate patterns are used to dig your way to the data - but sometimes you still want to see the data they extracted as well, as in the above case for example. No worries - scRUBYt! comes with a lot of defaults, which can be tweaked by further parameters. Let's add a `:write_text => true` parameter to the `link_title` pattern and check out the result.

```
google_data = Scrubyt::Extractor.define do
 fetch 'http://www.google.com/search?hl=en&q=ruby'

 link_title "//a[@class='l']", :write_text => true do
  url 'href', :type => :atribute
 end
end

puts google_data.to_xml
```

Now the extractor yields the following XML:

```
 <root>
  <link_title>
  Ruby Programming Language
  <url>http://ruby-lang.org/</url>
  </link_title>
  <link_title>
  Ruby (programming language) - Wikipedia, the free encyclopedia
  <url>http://en.wikipedia.org/wiki/Ruby_(programming_language)</url>
  </link_title>
  <link_title>
  Ruby - Wikipedia, the free encyclopedia
  <url>http://en.wikipedia.org/wiki/Ruby</url>
  </link_title>
  <link_title>
  Kaiser Chiefs - Ruby
  <url>http://www.youtube.com/watch?v=JMDcOViViNY</url>
  </link_title>
  <link_title>
  Kaiser Chiefs - Ruby: Video
  <url>http://www.youtube.com/watch?v=d14511Amd08</url>
  </link_title>
  <link_title>
  Ruby on Rails
  <url>http://www.rubyonrails.org/</url>
  </link_title>
  <link_title>
  Ruby - The Inspirational Weight Loss Journey on the Style Network
  <url>http://www.mystyle.com/mystyle/shows/ruby/index.jsp</url>
  </link_title>
  <link_title>
  Ruby&amp;#39;s Diner - rubys.com
  <url>http://www.rubys.com/</url>
  </link_title>
  <link_title>
  Ruby Central
  <url>http://www.rubycentral.org/</url>
  </link_title>
  <link_title>
  Ruby Annotation
  <url>http://www.w3.org/TR/ruby/</url>
  </link_title>
  <link_title>
  Welcome! [Ruby-Doc.org: Documenting the Ruby  Language]
  <url>http://www.ruby-doc.org/</url>
  </link_title>
  <link_title>
  Ruby Tuesday to close about 70 locations The Business Journal of ...
  <url>http://www.bizjournals.com/triad/stories/2008/12/15/daily71.html</url>
  </link_title>
</root>
```

OK, but XML is so Java in the 1990s…

It's obvious that if you have a complicated scraper, the result will be complicated too - so you might end up parsing the XML result, which would suck! Fortunately, the XML is just for viewing the result (unless you want to pass it on to a function requesting XML, transform it with XSLT etc. - in this case of course you can use the result as an XML) - most probably you want to store the result in a DB (perhaps first creating an ActiveRecord model from it), or in a flat file, or do something with it, and XML is not the best for this, obviously.

Method `to_hash` to the rescue. Replacing the very last line of the above example with `p google_data.to_hash` prints the result below:

```
[{:link_title=>"Ruby Programming Language", :url=>"http://ruby-lang.org/"},
{:link_title=>
```

```
   "Ruby (programming language) - Wikipedia, the free encyclopedia",
   :url=>"http://en.wikipedia.org/wiki/Ruby_(programming_language)"},
  {:link_title=>"Ruby - Wikipedia, the free encyclopedia",
   :url=>"http://en.wikipedia.org/wiki/Ruby"},
  {:link_title=>"Kaiser Chiefs - Ruby",
   :url=>"http://www.youtube.com/watch?v=JMDcOViViNY"},
  {:link_title=>"Kaiser Chiefs - Ruby: Video",
   :url=>"http://www.youtube.com/watch?v=d14511Amd08"},
  {:link_title=>"Ruby on Rails", :url=>"http://www.rubyonrails.org/"},
  {:link_title=>
   "Ruby - The Inspirational Weight Loss Journey on the Style Network",
   :url=>"http://www.mystyle.com/mystyle/shows/ruby/index.jsp"},
  {:link_title=>"Ruby&#39;s Diner - rubys.com", :url=>"http://www.rubys.com/"},
  {:link_title=>"Ruby Central", :url=>"http://www.rubycentral.org/"},
  {:link_title=>"Ruby Annotation", :url=>"http://www.w3.org/TR/ruby/"},
  {:link_title=>"Welcome! [Ruby-Doc.org: Documenting the Ruby  Language]",
   :url=>"http://www.ruby-doc.org/"},
  {:link_title=>
   "Ruby Tuesday to close about 70 locations The Business Journal of ...",
   :url=>"http://www.bizjournals.com/triad/stories/2008/12/15/daily71.html"}]
```

i.e. you get an array of hashes, where every hash contains the result of the patterns.

The Regexp pattern

Sometimes you'd like to match and extract a portion of the input with a regexp - let's say you'd like to get just the hosts and not the full URLs in the above example. We add a regexp pattern to our scraper tree like so:

```
google_data = Scrubyt::Extractor.define do
 fetch 'http://www.google.com/search?hl=en&q=ruby'

 link_title "//a[@class='l']" do
  url "href", :type => :attribute do
    host /\/\/.+\.(.+?\..+?)\.?\//
  end
 end
end

pp google_data.to_hash
```

and the result:

```
[{:host=>"ruby-lang.org"},
{:host=>"wikipedia.org"},
{:host=>"wikipedia.org"},
{:host=>"youtube.com"},
{:host=>"youtube.com"},
{:host=>"rubyonrails.org"},
{:host=>"mystyle.com"},
{:host=>"rubys.com"},
{:host=>"rubycentral.org"},
{:host=>"w3.org"},
{:host=>"ruby-doc.org"},
{:host=>"bizjournals.com"}]
```

Raw Ruby Power inside a scRUBYt! extractor

Let me introduce another very useful pattern type - the *script pattern*. The script pattern executes an arbitrary Ruby code on the input - let's continue with our google example for a demonstration: (let's say you want to have the host names in upper case)

```
google_data = Scrubyt::Extractor.define do
 fetch 'http://www.google.com/search?hl=en&q=ruby'

 link_title "//a[@class='l']" do
```

```
    url "href", :type => :attribute do
      host /\/\/.*\.(.+?\..+?)\.?\// do
        big_host lambda {|x| x.upcase}, :type => :script
      end
    end
  end
end
```

As you would expect, the above example prints a similar result than the one before, but this time with capital letters.

Further Filtering the Result

Let's say you don't want all the results (a typical scenario with tables - usually the first row/record is some kind of header, and likewise, the last record might be a footer etc.). You might tweak the scraped results with `select_indices` like this:

```
google_data = Scrubyt::Extractor.define do
 fetch 'http://www.google.com/search?hl=en&q=ruby'

 link_title "//a[@class='l']" do
  url "href", :type => :attribute do
    host /\/\/.*\.(.+?\..+?)\.?\// do
      big_host lambda {|x| x.upcase}, :type => :script
    end
  end
 end.select_indices(1..3)
end
```

As you would expect, this extracts just the 2nd, 3rd and 4th google search result host rather than all of them.

There are other methods of filtering and further tweaking the results (most notably constraints) - unfortunately these are out of the scope of the article, since it already grew too long ;-).

A real life example

Here is an example I have been hacking on yesterday evening. It downloads some files to a directory specified by the scraper and renames them based on the file description and original file name. You can observer yet another pattern type, the *download pattern*. Also note that not everything fits into scRUBYt!'s API, and in this case you might need some post-processing in pure Ruby.

```
require 'rubygems'
require 'scrubyt'

Scrubyt.logger = Scrubyt::Logger.new
data = Scrubyt::Extractor.define do
 fetch 'http://www.us.kohler.com/tech/cutout_result.jsp?module=Sink'

 content_div "//div[@id='content-section']" do
  table "/table[2]" do
    row "//tr" do
      desc "/td[1]"
      file_name "/td[2]" do
        link lambda {|x| "http://www.us.kohler.com/tech/dxffile/" + x}, :type => :script do
          dl_file 'downloaded_files', :type => :download
        end
      end
    end.select_indices :all_but_first
  end
 end
end

data.to_flat_hash.each do |record|
 `mv downloaded_files/#{record[:dl_file]} 'downloaded_files/#{record[:desc]}-#{record[:dl_file]}'`
end
```