Scraping for Jobs with Perl and WWW::Mechanize

John Perry
Database Editor
The Oklahoman

The greatest thing about Perl is the abundance of modules that extend its capabilities. But before you can take advantage of their capabilities, you have to install the module, and often figure out the object style syntax that many modules now use.

In this two-part tutorial, we will build a Web spider to find pages on the IRE site with job information, and then scrape the job info from the pages. Along the way we'll also learn about Perl modules, object-oriented syntax and complex data structures.

In this first tutorial, we'll use WWW::Mechanize to collect links from a page and then follow them. WWW::Mechanize is a fairly new module created to automate Web site testing. It understands forms, links and buttons, taking care of all the messy details of creating the proper HTTP request and parsing the response. It's also an object-oriented module.

We'll also use a the Data::Dump module, a traditional procedural module, to peek inside the WWW::Mechanize objects to see what they're about.

Installing Modules: The Windows Version

If you're using ActivePerl on a Windows machine, then you have the Perl Package Manager, or ppm, to make adding modules easy. Open a command window and type ppm. You'll see some introductory text:

```
Entering interactive shell. Using Term::ReadLine::Stub as readline
library.

Profile tracking is not enabled. If you save and restore profiles
manually,
your profile may be out of sync with your computer. See 'help profile'
for
more information.

Type 'help' to get started.
```

Then a prompt like this:

```
ppm>
```

You're now in the ppm shell program. To get a list of the available commands, type help. To get a list of available modules, use the search command:

```
ppm> search WWW-Mechanize
Searching in Active Repositories
```

```
    1. WWW-Mechanize            [0.44] automate interaction with websites
    2. WWW-Mechanize-FormFiller [0.03] framework to automate HTML forms
    3. WWW-Mechanize-Shell      [0.19] A crude shell for WWW::Mechanize
```

With the install command, ppm looks through the various ActivePerl module repositories, downloads the module, compiles it if necessary, installs it in the appropriate directory and also installs any documentation for the module. It may also install other related modules:

```
ppm> install WWW-Mechanize
====================
Install 'WWW-Mechanize' version 0.44 in ActivePerl 5.8.0.805.
====================
Downloaded 18097 bytes.
Extracting 7/7: blib/arch/auto/WWW/Mechanize/.exists
Installing C:\Perl\html\site\lib\WWW\Mechanize.html
Installing C:\Perl\html\site\lib\WWW\Mechanize\Examples.html
Installing C:\Perl\site\lib\WWW\Mechanize.pm
Installing C:\Perl\site\lib\WWW\Mechanize\Examples.pod
Successfully installed WWW-Mechanize version 0.44 in ActivePerl
5.8.0.805.
```

Data::Dump is a core module, so you should not have to install it

Installing Modules: Linux/Unix

In the [Li|U]nix world, you'll use the CPAN shell to install modules. First su to root. Then open the CPAN shell with this shell command: `perl -MCPAN -e 'shell'`. You should see this message and prompt:

```
cpan shell -- CPAN exploration and modules installation (v1.70)
ReadLine support enabled

cpan>
```

For a list of commands, type `help` or `?`. To search for available modules, use the m command, followed by a search word or regular expression. To install a module, use the install command:

```
cpan> install WWW::Mechanize
```

After which you will get a flood of messages telling you which sites are being search, how every step in the make, compile and install is proceeding, the results of any tests of the installation. And at the end, if everything is well, you should see something like:

```
  /usr/bin/make install  -- OK

cpan>
```

An Object Lesson

With WWW::Mechanize installed, you're ready to start writing code. But first, a minimum of object oriented theory may make what follows more understandable.
In the old days of procedural programming, you wrote code that told the computer to do this and then that, and if such and such then do this, or else do that. If the program got very big, you broke the code up into subroutines or functions that did one specific thing, then combined the subroutines to create the larger program.

Object-oriented programming takes a different approach. Instead of procedures, you build objects, which are computer versions of, well, objects in the real world. Like real world objects, computer objects have attributes -- A car can have a color attribute of red; a web server response object can have a status attribute of 401 (unauthorized).

Computer objects, like real world objects, also have tasks they can perform – a deck of cards can perform a deal-card-off-bottom task; a database handle object can perform a select-next-row-and-save-as-array task. These tasks are called methods.

The advantage of using objects to write computer programs is that, if you know how to set or read its attributes and how to access methods methods, you don't need to know anything about an object's inner workings. This is called encapsulation. Another advantage is that objects can be used as a base to create other objects. This is called inheritance. Families of related objects, parents and siblings, are called classes.

True to its TMTOWTDI philosophy, Perl lets you write code in either procedural or object-oriented style. In fact, Perl simply reuses procedural coding ideas to build object coding capabilities. In Perl, at the simplest level, a class is a module, a method is a subroutine and an object is a reference.

A reference is a variable that, instead of holding information such as a number or character string, holds a memory location where the information can be found. References let you can store the complex data structure of an object in a single scalar variable.

A Hash With No Name

Now, finally, to our intended task of scraping job announcements from the IRE site. It's always good practice to start a Perl script with these lines:

```
#!/usr/bin/perl

use warnings;
use strict;
```

The first line, the shebang, tells a Linux/Unix computer where to find the Perl interpretor. For cross-platform portability, it's a good idea to include it even if you're working in a Windows environment.

The next two lines include pragmas in your code. Pragmas are modules that give special instructions to the Perl compiler. The warnings pragma tells the compiler to point out code that, while legal, looks not quite right. The strict pragma gives an error if you use an undeclared variable. This helps find typos in variable names. Another helpful pragma is diagnostics, which translates warnings and errors into something resembling English. Now we will add the WWW::Mechanize and Data::Dump modules:

```
use WWW::Mechanize;
use Data::Dump qw( dump );
```

The qw(dump), the equivalent of ('dump'), exports the dump function for use in your code.

Next we'll create a WWW::Mechanize object and assign it to a scalar variable, `$browser`, which we have to declare (since we're using strict) with the `my` statement:

```
my $browser = WWW::Mechanize->new;
```

So here is our first example of Perl's object-oriented syntax. The arrow operator (->) is used to call the method, 'new', on the WWW::Mechanize class. New is a method that all classes must have. It is used to create, or instantiate in object talk, a new object from the class – in this case a new browser object. The object is saved in a scalar variable. What the variable actually holds is a memory location that's been set aside for the object. So if we print `$browser` with a statement such as this:

```
print "$browser";
```

What we get back is an obscure looking message that tells us that $browser is a reference to a hash stored at a particular memory location:

```
WWW::Mechanize=HASH(0x8631048)
```

To see what's being held in that memory location, we can use Data::Dump's dump function:

```
dump $browser;
```

And here's the result:

```
bless({
  agent => "WWW-Mechanize/0.44",
  cookie_jar => bless({ COOKIES => {} }, "HTTP::Cookies"),
  from => undef,
  max_size => undef,
  no_proxy => [],
  page_stack => [],
  parse_head => 1,
  protocols_allowed => undef,
  protocols_forbidden => undef,
  proxy => undef,
  quiet => 0,
  "requests_redirectable" => ["GET", "HEAD", "POST"],
  timeout => 180,
  use_eval => 1,
}, "WWW::Mechanize")
```

In the dump syntax, everything inside curly brackets, {}, is a hash. Items in the hash are listed as, key=>value, key=>value, ... Anything listed within square brackets, [], is an array. As you can see above, the hash key 'requests_redirectable' actually identifies and array, ["GET", "HEAD", "PORT"]. This is an example of how, using references, you can build complex data structures of hashes and arrays.

So the dump of `$browser` shows us that it is a reference to an anonymous hash – anonymous because it has no name of its own. It is only the referent -- or target -- of a reference that we stored in $browser.

The function `bless()` declares that a referent is an object of a class. So the statement, `bless( { ... }, WWW::Mechanize )`, declares the anonymous hash reference to be an object of the WWW::Mechanize class.

So now we have our browser object., let's do something with it:

```
$browser->get( 'http://www.ire.org/jobs/look.html' );
```

WWW::Mechanize's get method sends an http request to a web server and returns the response. Taking another look inside our browser object with the dump function, we now see pages of data that starts out like this:

```
my $a = bless({
    agent      => "WWW-Mechanize/0.44",
    base       => bless(do{\(my $o ="http://www.ire.org/jobs/")},
"URI::http"),
    content    => "<!DOCTYPE HTML PUBLIC \"-//IETF//DTD HTML
3.2//EN\">\r\n<html>\r\n<head>\r\n<base
href=\"http://www.ire.org/jobs/\">\r\n<title>\r\nIRE Job
Postings\r\n</title>\r\n</head>\r\n<!--This is the start of the IRE SSI--
>\r\n\r\n<link rel=stylesheet HREF=\"http://www.ire.org/ire.css\"
TYPE=\"text/css\">\r\n</head>\r\n<body
background=\"http://www.ire.org/images/background-testire.gif\"
BGCOLOR=\"#FFFFFF\" TEXT=\"#000000\" LINK=\"#000099\" ...
```

... And on and on for several pages.

Now the HTML content of the page you pointed to is saved as the value of the hash key 'content.' Your object also contains information from the http response header, lists of information about links and forms on the page, the URL you pointed to, and much more.

To access this information, WWW::Mechanize has several status methods. You can apply the `content()` method to your object using the arrow operator to get the page's content:

```
print $browser->content();
```

To get a list of links on the page, the links method returns an array of link objects, each containing the URL it points to, the text between the start and end tags of the link, the link name, if one is set in the link tag, and the tag type.

Job Search

OK, enough theory. Now its time for some Web spidering. The IRE site posts its list of jobs openings here: http://www.ire.org/jobs/look.html. Each job listing has a link to a page with more information about the job. To get to that information, we first must collect all the job links from the first page:

```
$browser->get( 'http://www.ire.org/jobs/look.html' );
dump $browser->links;
```

And we get a list of object representing all the links on the page that look like this:

```
bless(["http://www.ire.org/", "[IMG]", undef, "a"],
"WWW::Mechanize::Link"),
```

This first link object represents the IRE logo graphic at the top left of the page that links back to the IRE home page. The list also includes all the navigation links from the page's left hand column. Clearly these are links we don't want to follow. We want to filter them out and keep only links to jobs. And WWW::Mechanize has a method to do just that. The find_all_links method lets you use a text string or a regular expression to match a link's text, name or url.

Here is an example of a job link that we do want to follow:

```
bless(["12atlanta1.html", "Sunday Editor", undef, "a"],
"WWW::Mechanize::Link")
```

The URL is relative – it gives only the name of a file in the same directory and doesn't include the http://www.ire.org. Also, the page name starts with two digits followed by some other characters, usually a city name, followed by the .html extension. So we can build a regular expression that looks for some digits at the beginning, followed by some stuff, and ending with ".html" and then use it in the find_all_url method and save only the matching links in an array:

```
my @links = $browser->find_all_links( url_regex => qr/^\d+.+\.html/ );
```

Now we have a list of job link objects. All we have to do is add a loop to step through the list, follow each link URL, grab the contents, then return to the main job page.

```
for ( @links ) {
    $browser->follow_link( text => $_->text );
    print "-------------------------------------------------------\n";
    print $browser->content, "\n";
    $browser->back;
}
```

The for statement steps through our array of link objects, storing each one in turn in Perl's special `$_` variable. But the key statement is on the second line. We access the text for each link by applying the text method to the `$_` variable (`$_->text`). Then, applying the follow_link method to `$browser`, we tell our browser object to follow the link on the page with text that matches the text given text string. And after printing the content of the new page, we return to the main job page with the back method.

So now we have our job information scraped from the job detail pages. It's still mixed in with all the HTML code. But Perl has a collection of modules that can help us with the next step of parsing the information we want out of the HTML. And that will be the topic of the scraping for jobs tutorial, part 2.