Automating website access with screen scrapers

Simple Data Scraper

Sometimes it is just too much trouble to dig down through the structure of a website simply to find a few snippets of information. A Perl module called *WWW::Mechanize::Shell* facilitates writing so-called screen scrapers, tools that act like browsers and automate website access.

BY MICHAEL SCHILLI

In California, you can select your own license plates, for a small charge. This explains why the plates on my 13-year-old Acura Integra read "PERL MAN" (see Figure 1). The department of motor vehicles wears a progressive hat and provides a website where you can check the availability of your customized plates. Sadly, most find access to the site too slow. This article shows how to use a Perl script to emulate web requests.

Let's write a screen scraper: a program acting like a web browser, happily clicking through web pages, but without the manual user interaction like mouse-clicking or typing on the keyboard.

CPAN Modules to the rescue

WWW::Mechanize by Andy Lester provides the framework for coding screen scrapers in Perl. *WWW::Mechanize::Shell*, an addition by Max Maischein, spawns a shell-like command interpreter to control a virtual browser. Developers steer the session interactively, and, on a push of a button, convert the command sequence into a permanent Perl script.

The session description is a logical abstraction (for example "follow the link containing the string 'xxx' on the current page") and ensures that the script will not break, just because the format of the website has been slightly modified.

Figure 1: The magnificent PERL MAN: 13 years old, but that doesn't stop it from drag racing those sports cars at San Francisco's traffic lights.

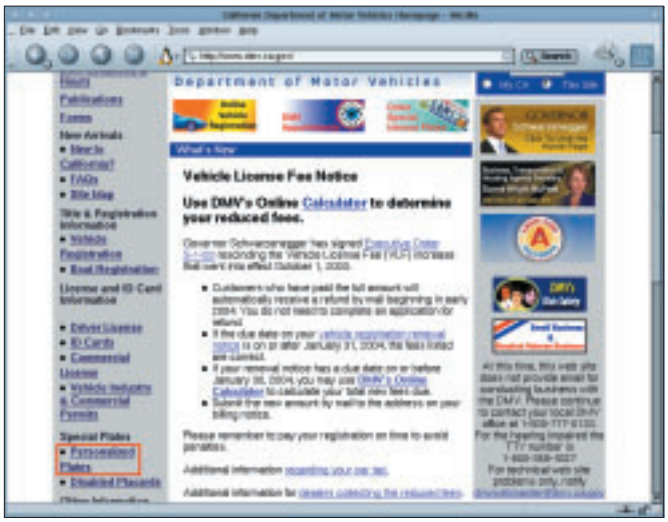

Figure 2: The welcome page of the Californian Department of Motor Vehicles. The highlighted link leads to a page where Californians can create their own personalized license plates.

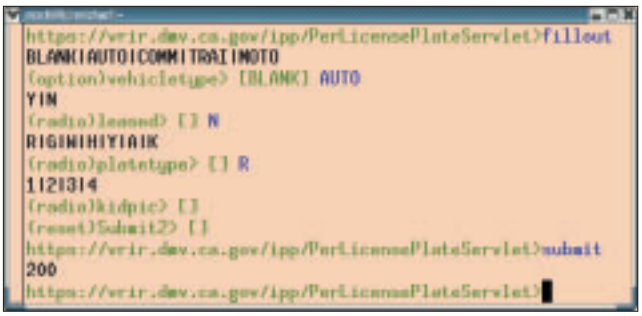

Figure 3: The shell provided by the *WWW::Mechanize::Shell* module acts just like a Web browser. In this example, the user has just filled out a form using a dialog provided by the the *fillout* command.

Most conveniently, you can use the CPAN shell to install the *WWW::Mechanize* and *WWW::Mechanize::Shell* modules. Both require additional modules, which are easy to install thanks to automatically resolved dependencies. To allow the screen scraper to handle HTTPS pages, the *IO::Socket::SSL* module is also required. The browser emulation shell is called as follows:

```
perl -MWWW::Mechanize::➥
Shell -eshell
```

This displays a prompt to the user, waiting for input. To load the Californian Department of Motor Vehicles' Web page, you can then type *get http://www.dmv.ca.gov*. The shell will respond with *Retrieving http://www.dmv.ca.gov(200)*, to indicate that the script has successfully loaded the initial page (HTTP code 200).

The browser rendering of the page in Figure 2 shows the link to "Personalized Plates", which is what we are looking for. We can now enter the *links* command to find out which links the shell has discovered on this page, and display a list:

```
>links
...
[14] Vehicle Industry ➥
& Commercial Permits
[15] Personalized Plates
[16] Disabled Placards
...
```

Regular Expressions for Links

The user could now follow link number 15 by typing *open 15*, but the numerical value would soon lose its validity, as the number and order of links on a page that is maintained regularly will tend to change. In other words, this process should be more abstract, to avoid breaking the script, if the authority adds new links to the page. The next command searches for a link that contains the text string *Personalized*, and then follows that link:

```
>open /Personalized/
```

If the regular expression matches multiple links on the page, the shell will display a selection menu. In our case, there is only one match; this causes the shell to display the following

```
Found 15
(200)
```

This moves the virtual browser on to the next page which includes a link containing the text string *order Special Interest and Personalized license plates*. Searching for this string, *open "/order Special Interest/"*, leads to another page that contains a number of HTML forms. Note that the regular expression in the *open* command just referred to has to be enclosed in double quotes, as it contains blanks. Unquoted, this would confuse the shell, which uses blanks to split a command and its arguments. The user can now display the forms on the current page using the *forms* command:

```
>forms
...
```

Figure 4: If you are registering a vehicle in California, you can create your own personalized license plate, including a stylish background, on the Department of Motor Vehicles website.

```
Form [2]
POST https://vrir.dmv.ca.gov/➥
ipp/ PerLicensePlateServlet [➥
personalized]
  page=Select          (hidden)
  Submit2=Order Personalized ➥
(submit)
...
```

Form #2, the one labelled *personalized*, is what we are looking for. The form can be selected using the *form "personalized"* command and then typing *submit*. On the next page, the user can then access another form with dropdown menus and radio buttons to specify the vehicle type, the plate design and other things. The shell's *fillout* command comes in useful for complex forms like this, allowing the user interactively to define values for the individual fields in the form. The dialog shown in Figure 3 illustrates this. To select the default option, like to disable the *kidpic* radio button, just press the Enter key. After filling out the form, enter the *submit* command to transmit the form to the server.

Your own License Plate

The next page finally allows the user to select a personalized plate. As the browser rendering in Figure 4 shows, an HTML form is again used for this purpose. The form uses individual (!) selection boxes to collect the letters for the personalized plate. Again, *fillout* will provide the required dialog box, and *submit* will transmit the data.

A Script at the Push of a Button

This is where things start getting interesting. All we need to do, to transform the shell session so far into a reusable Perl script, is to enter the *script* command. Immediatley, the ready-to-use script will show up on standard output. Snap it up via cut-and-paste, or use *script filename,* to store it in a file on disk.

This is how I created the *dmv* script in Listing 1. There are a few modifications. The user now enters the license plate

details on the command line, as in *dmv* PERLMAN. Line 13 exits, if this parameter is missing. The constructor of the *WWW::Mechanize* object has the *autocheck* option enabled, and runs the browser simulator in a mode that immediately terminates the program if the website cannot be found.

Line 26 points the simulator to the welcome page. The *follow()* method in line 28 uses a regular expression to locate a link containing the *Personalized* text string. Line 30 performs another regex-controlled jump. Line 32 selects the form with the *personalized* label, and the *submit()* method that follows activates the *Submit* button of the form, which is otherwise empty.

The *WWW::Mechanize::FormFiller* object created in line 22 takes care of filling out the form that then appears. The *add_filler()* method specifies the field name, the input method and the value:

```
$fi->add_filler('leased' ➥
=> Fixed => 'N' );
```

Fixed assigns some hard coded values, while *Interactive* tells the form filler code to prompt the user for input at runtime. After collecting all the values for the form variables, the form filler can get to work. The *fill_form* method is run against the *WWW::Mechanize* agent's current *HTML::Form* object to fill out the form.

The script accesses the DMV welcome page, navigates the links and forms, and finally uses a for loop starting in line 55 to enter the supplied strings into the selection boxes. If the license plate contains less than 7 letters, lines 59-61 add some padding. The submit in line 75 uses SSL to transmit the data to the server, and the shell accepts the HTML formatted results page.

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads*

[2] Californian Department of Motor Vehicles: *http://www.dmv.ca.gov*

There Can Only Be One

If the results page says something like *not available*, you can assume that the combination is already in use, or that the language is unacceptable to the department of motor vehicles. The script issues a warning in this case.

Assuming that the license plate has been okayed, an order form appears. The script checks for the *Complete Order Form* text string, and responds by displaying XXX*: available*. The response, *PERLMAN: not available*, confirms that *PERL MAN* is already in use, and will be so for quite a while in the future. There can only be one! ■

THE AUTHOR

Michael Schilli works as a Web engineer for AOL/Netscape in Mountain View, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at mschilli@perlmeister.com. His homepage is at http://perlmeister.com.

Listing 1: dmv

```
001 #!/usr/bin/perl
002 ##################################
003 # dmv -- Check CA plates
004 # Mike Schilli, 2003
005 # (m@perlmeister.com)
006 ##################################
007 use strict;
008 use warnings;
009
010 use WWW::Mechanize;
011 use
    WWW::Mechanize::FormFiller;
012
013 die "usage: $0 XXXXXXX"
014    unless defined $ARGV[0];
015
016 $ARGV[0] =~ s/\s+//g;
017
018 my $agent =
019   WWW::Mechanize->new(
020     autocheck => 1);
021
022 my $fi =
023  WWW::Mechanize::FormFiller->
024  new();
025
026 $agent->get(
027     'http://www.dmv.ca.gov');
028 $agent->follow(
029            qr(Personalized));
030 $agent->follow(
031  qr(order Special Interest));
032 $agent->form("personalized");
033 $agent->submit();
034
035 $fi->add_filler(
036   'vehicletype' =>
037     Fixed => 'AUTO' );
038 $fi->add_filler(
039   'leased' =>
040     Fixed => 'N' );
041 $fi->add_filler(
042   'platetype' =>
043     Fixed => 'R' );
044 $fi->add_filler(
045   'kidpic' =>
046     Fixed => '' );
047 $fi->add_filler(
048   'Submit2' =>
049     Fixed => '' );
050
051 $fi->fill_form(
052   $agent->current_form);
053 $agent->submit();
054
055 for(0..6) {
056   $fi->add_filler(
057     "LicPltCharAry$_" =>
058       Fixed =>
059       $_ > length $ARGV[0] ?
060         "" : substr($ARGV[0],
061                     $_, 1));
062 }
063
064 for(0..6) {
065   $fi->add_filler(
066     "HalfSpace$_" =>
067       Fixed => '');
068 }
069
070 $fi->add_filler(
071   'Submit2' => Fixed => '');
072 $fi->fill_form(
073   $agent->current_form);
074
075 $agent->submit();
076
077 if($agent->content() =~
078            /not available/) {
079  print "$ARGV[0]: " .
080        "not available\n";
081 } elsif($agent->content() =~
082      /Complete Order Form/) {
083     print "$ARGV[0]: " .
084           "available\n";
085 } else {
086     print "Unexpected: ",
087           $agent->content(),
088           "\n";
089 }
```