DAVID N. BLANK-EDELMAN

practical Perl tools: be the browser

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Perl for System Administration*. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

BELIEVE IT OR NOT, MY SYSADMIN-themed column for this issue is about Web crawling, automation, scraping, browsing, circumnavigating—whatever you'd like to call it. Why is this something a sysadmin would want to automate, versus, say, a Web developer? Besides your everyday sysadmin-related surfing (searching for reference material or answers to questions, participating in the community, etc.), you may have noticed the increasing number of Web sites to which you need to connect strictly to get your job done. Maybe you need to work with mailing lists, submit requests to a certificate authority, interact with a trouble-ticket system, or deal with any number of Web applications. Surely it would be pleasant to reduce the amount of menial pointing and clicking you do on a daily basis. This column can help.

One quick caveat before we dive in, mostly for the regular readers of the column. In most of my columns I've tried to present a number of tools to perform a task. I like the idea that my readers can assemble their own tool chest from which to choose the best utensil or approach. Well, in today's column I'm going to hand you a single spanner wrench and say, "Here, here's the best tool."

In this column we're going to focus on using just the module WWW::Mechanize and the modules in its orbit. There are other modules out there for performing tasks like the ones we're going to explore, but I've not found one to date that I've liked better than WWW::Mechanize. I should note that I'm not the only person enamored of this module. Mech, as it is affectionately called, has been reimplemented in several other of your favorite languages: pymechanize (in Python) and WWW::Mechanize (in Ruby). The boundary between Perl and Ruby is actually porous in both directions. We're not going to look at it in this column, but WWW::Mechanize::Plugin::Web::Scraper lets you use the module Web::Scraper from WWW::Mechanize. Web::Scraper takes its design from the excellent Ruby-based scrAPI toolkit.

First Steps with WWW::Mechanize

Almost all WWW::Mechanize scripts start out like this:

```
use WWW::Mechanize;

my $mech = WWW::Mechanize->new();
$mech->get($url); # get can also take a :content_file param to save to a file
```

We initialize a new Mech object and ask it to go fetch some Web page. If we want the contents of the page we just fetched, we call:

```
my $pagecontents = $mech->content();
```

It's not uncommon to hand the results of the content() method off to some other module (for example, we could feed it to HTML::TableExtract, one of my favorite modules) to do more sophisticated parsing. We won't see an example of that handoff in this column, but I wouldn't be surprised if it didn't find its way into a future column.

OK, so far the code has been really simple. So simple LWP::Simple could have basically handled it. Let's take things to the next level:

```
use WWW::Mechanize;

my $mech = WWW::Mechanize->new();

$mech->get( 'http://www.amazon.com' );
$mech->follow_link( text => 'Help' );
print $mech->uri . "\n";

# prints out something like:
# http://www.amazon.com/gp/help/customer/display.html?ie=UTF8&nodeId
# =508510
```

So what happened here? WWW::Mechanize retrieved the home page for Amazon and then found the link on the page with the text 'Help'. It followed the link in the same way you would on a browser and retrieved the contents of the URL specified in the link. If we called `$mech- >content()` at this poinr, we'd get back the contents of the new page found by browsing to the selected link.

If we wanted to, we could use an even cooler feature and write something like:

```
$mech->follow_link ( text_regex => qr/rates.*policies/ );
# or
$mech->follow_link ( url_regex => qr/gourmet.*food/ );
```

The first line of code will find and then follow the first link whose text matches the given regular expression. This means we can follow links in a page without knowing the precise text used (e.g., if each page was generated dynamically and had unique links). The second line of code performs a similar find and follow, this time based on the URL in the link.

follow_link() has a number of other options available. There's a related url => 'http://...' option equivalent to the text => 'text' option that will take a fully specified URL to follow. Though this is more fragile, follow_link can also take an n => option to allow you to choose the *n*th link on the page. All of the options mentioned so far can be compounded. If you want the third 'help' related link on a page with a URL that included the path 'forum' in its name you could write:

```
$mech->follow_link( text => 'help', url_regex => 'forum', n => 3 );
```

If for some reason you want to just find the links on a page without navigating to their target, WWW::Mechanize provides `find_link` and `find_all_links`, which take the same selector arguments as `follow_link`. WWW::Mechanize can also find images on a page via `find_images` and `find_all_images`, which use similar arguments.

WWW::Mechanize Tip #1: mech-dump

Now that we've seen some of the basic stuff, I'd like to show you a couple of tips that will make the more complex WWW::Mechanize work we're going to explore considerably easier.

The first tip involves a utility that ships with WWW::Mechanize and optionally gets installed during the module's installation: mech-dump. mech-dump calls WWW::Mechanize and gives you a little bit of insight into how WWW::Mechanize is parsing a particular page. It offers four choices:

- 1. Display all forms found on a page.
- 2. Display all links found on a page.
- 3. Display all images found on a page.
- 4. Display all of the above.

Let's see it in action:

```
$ mech-dump --links http://www.amazon.com
http://www.amazon.com/access
/
/gp/yourstore/ref=pd_irl_gw?ie=UTF8&signIn=1
/gp/yourstore/home/ref=topnav_ys_gw
  ...
```

I cut that list off quickly, because:

```
$ mech-dump --links http://www.amazon.com|wc -l
  247
```

Finding links can be helpful, but this command really shines when it comes time to interact with forms, something we're going to do in just a moment:

```
$ mech-dump --forms http://www.boingboing.net

GET http://www.google.com/search
  ie=UTF-8                                  (hidden readonly)
  oe=UTF-8                                  (hidden readonly)
  domains=boingboing.net                    (hidden readonly)
  sitesearch=boingboing.net                 (hidden readonly)
  q=                                        (text)
  btnG=Search                               (submit)

POST http://www.feedburner.com/fb/a/emailverify
  email=                                    (text)
  url=http://feeds.feedburner.com/~e?ffid=18399 (hidden readonly)
  title=Boing Boing                         (hidden readonly)
  loc=en_US                                 (hidden readonly)
  <NONAME>=Subscribe                        (submit)
```

The output shows us that each form has a number of fields. Some are hidden fields set in the form by the form's author, but the useful information in the output is the fields that someone sitting at a browser would need to fill and select. For example, the blog BoingBoing has an option to allow people to subscribe via email using a Feedburner service. The output of mech-dump

lets us know that we'll be filling in a field called "email" (versus something else such as "address" or "user_email" or any number of possibilities).

Let's put this utility into practice to help with our WWW::Mechanize programming. I recently had a need to scrape some information from our internal wiki. We use a commercial product that has its own login screen to control access (versus depending on external authentication performed by the Web server). To get past the login screen to the content I needed, my program had to fill in the Web form on the login screen. The first step toward figuring out how to do so was to run a `mech-dump --forms` on that page. (A quick aside: Mech can talk to anything LWP::UserAgent can talk to. This means that pages served over HTTPS are not a problem as long as you have Crypt::SSLeay or IO::Socket::SSL installed.)

`mech-dump --forms` returned something like the following:

```
GET https://wiki.example.edu/dosearchsite.action
where=conf_all (hidden readonly)
queryString= (text)
<NONAME>=Search (submit)
<NONAME>=Search (hidden readonly)

POST https://wiki.example.edu/login.action [loginform]
os_username= (text)
os_password= (password)
os_cookie=<UNDEF> (checkbox) [*<UNDEF>/off|true]
login=Log In (submit)
os_destination= (hidden readonly)
```

The first form on the page is a search box; the second is exactly the one I needed. The output for the second form told me that I needed to provide two fields to log in: `os_username` and `os_password`. In WWW::Mechanize you can use the `submit_form` method to fill in a form, like so:

```
use WWW::Mechanize;

my $mech = WWW::Mechanize->new();
$mech->get( $loginurl );
$mech->submit_form(
form_number => 2,
fields => { os_username => $user, os_password => $pass },
);
```

`submit_form` chooses the form to use, fills in the given fields, and performs the "submit" action (the equivalent of selecting the 'Log In' element on the page). Now the script is "logged in" to the wiki and can proceed to operate on the protected pages.

WWW::Mechanize Tip #2: WWW::Mechanize::Shell

The second tip I want to pass on is about a companion module called WWW::Mechanize::Shell. WWW::Mechanize::Shell calls itself "an interactive shell for WWW::Mechanize." If you type the following slightly unwieldy command:

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

you get an interactive shell with the following commands:

```
(no url)>help
Type 'help command' for more detailed help on a command.
    Commands:
    auth:       Sets basic authentication credentials.
    autofill:   Defines an automatic value.
    back:       Goes back one page in the browser page history.
    browse:     Opens the Web browser with the current page.
    click:      Clicks on the button named NAME.
    comment:    Adds a comment to the script and the history.
    content:    Displays the content for the current page.
    cookies:    Sets the cookie file name.
    ct:         Prints the content type of the most current response.
    dump:       Dumps the values of the current form.
    eval:       Evaluates Perl code and print the result.
    exit:       Exits the program.
    fillout:    Fills out the current form.
    form:       Selects the form named NAME.
    forms:      Displays all forms on the current page.
    get:        Downloads a specific URL.
    headers:    Prints all C<< <H1> >> through C<< <H5> >> strings found in
                the content.
    help:       Prints this screen, or help on 'command'.
    history:    Displays your current session history as the relevant commands.
    links:      Displays all links on a page.
    open:       <open> accepts one argument, which can be a regular
                expression or the number.
    parse:      Dumps the output of HTML::TokeParser of the current content.
    quit:       Exits the program.
    referer:    Alias for referrer.
    referrer:   Sets the value of the Referer: header.
    reload:     Repeats the last request, thus reloading the current page.
    response:   Displays the last server response.
    restart:    Restarts the shell.
    save:       Downloads a link into a file.
    script:     Displays your current session history as a Perl script using
                WWW::Mechanize.
    set:        Sets a shell option.
    source:     Executes a batch of commands from a file.
    submit:     Submits the form without clicking on any button.
    table:      Displays a table described by the columns COLUMNS.
    tables:     Displays a list of tables.
    tick:       Sets checkbox marks.
    timeout:    Sets new timeout value for the agent. Affects all subsequent.
    title:      Displays the current page title as found.
    ua:         Gets/sets the current user agent.
    untick:     Removes checkbox marks.
    value:      Sets a form value.
    versions:   Prints the version numbers of important modules.
```

Seeing the whole list is a little intimidating, so I'll pick out a few choice commands for an example that will get us back into the main discussion of WWW::Mechanize use again. Let's look at a more complex forms example from a real-life problem.

One common Web-based administrative interface is the one used to configure and administer Mailman (http://www.list.org/) mailing lists. This interface mostly makes working with the server easier, but there have been some

issues. One issue that used to exist (it has been fixed in later versions) was its handling of large batches of spam. If a spammer sent a large batch of mail that was intercepted by Mailman and queued for an administrator to triage, that admin had to click the buttons to delete each message individually in a form that looks like that in Figure 1.

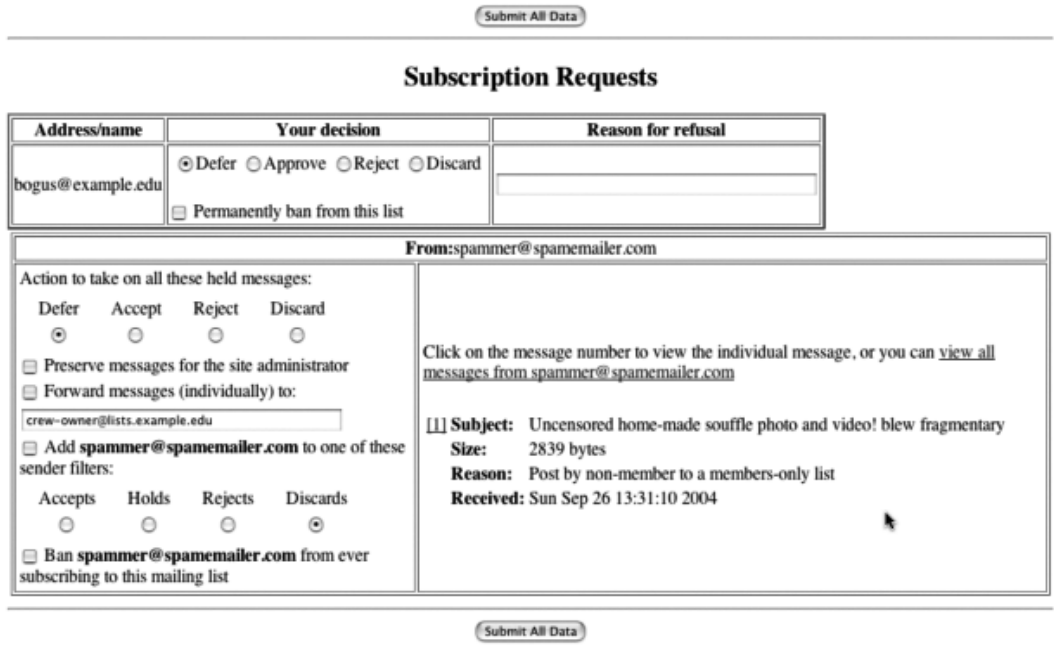

FIGURE 1: MAILMAN FORM

If you get 800 messages in the queue, this gets old quickly. Let's see how we can use WWW::Mechanize::Shell and WWW::Mechanize to make that pain go away.

The easiest way to start is to poke at the Mailman server with WWW::Mechanize::Shell:

```
$ perl -MWWW::Mechanize::Shell -eshell
    > get https://lists.example.edu/bin/admindb/mylist
    > form
    Form [1] (<no name>)
    POST https://lists.example.edu/bin/admindb/mylist
            adminpw= (password)
            admlogin=Let me in... (submit)
```

(where the output has been slightly edited for readability). OK, so this means that we will need to fill in adminpw to get access to the administration page. We set that value and submit the form. Let's do so:

```
    > value adminpw listpw
    > click admlogin
```

If we were to use the `form` command now we'd see a reasonably complex form dump that included lines such as:

```
senderaction-spammer%40spamemailer.com=0 (radio) [*0|1|2|3]
senderpreserve-spammer%40spamemailer.com=<UNDEF> (checkbox)
    [*<UNDEF>/off|1/?Preserve messages for the site administrator]
senderforward-spammer%40spamemailer.com=<UNDEF> (checkbox)
    [*<UNDEF>/off|1/?Forward messages (individually) to:]
senderforwardto-spammer%40spamemailer.com=mylist-owner@
lists.example.edu (text)
```

```
senderfilter-spammer%40spamemailer.com=3 (radio) [6|7|2|*3]
senderbanp-spammer%40spamemailer.com=<UNDEF> (checkbox)
[*<UNDEF>
```

These are all form elements we'd like to set without a major point-and-click fest. The tricky thing is that those elements are all message-sender specific. It says "senderaction-{some email address}" not just "senderaction." Here's where WWW::Mechanize-related options that take regular expressions come to our rescue. We can tell WWW::Mechanize::Shell to set elements that match a regular expression. Let's set all of those fields:

```
> autofill /senderfilter/ Keep
> autofill /senderfilterp/ Keep
> autofill /senderbanp/ Keep
> autofill /senderpreserve/ Keep
> autofill /senderforward/ Keep
> autofill /senderforwardto/ Keep
> autofill /senderaction/ Fixed 3
```

That last one might look a bit strange. It comes from the senderaction form definition. If you look carefully at the HTML source for the Mailman page (not reproduced here), you'll see that "3" in that case is the "Discard" option. In case you are curious about why we're setting all of the options in the form and not just the senderaction one, it's because the next command we're going to use will interactively prompt for any elements that don't have values set already:

```
> fillout
> click submit
```

Congrats! You've just avoided clicking on many hundreds of form elements, because they were set programmatically and the form has been submitted.

Now, here's an even cooler trick: If you type `script` at this point, it will spit out a Perl script that essentially reproduces your interactive session. If we were to run this command after typing in the session above (and edit it to remove some of the initial probing of the forms), we'd get output that looks like this:

```
#!perl -w
use strict;
use WWW::Mechanize;
use WWW::Mechanize::FormFiller;
use URI::URL;

my $agent = WWW::Mechanize->new( autocheck => 1 );
my $formfiller = WWW::Mechanize::FormFiller->new();
$agent->env_proxy();

$agent->get('https://lists.example.edu/bin/admindb/mylist');
$agent->form(1) if $agent->forms and scalar @{$agent->forms};
{ local $^W; $agent->current_form->value('adminpw', 'listpw'); };
$agent->click('admlogin');
$formfiller->add_filler( qr((?-xism:senderfilter)) => "Keep" => );
$formfiller->add_filler( qr((?-xism:senderfilterp)) => "Keep" => );
$formfiller->add_filler( qr((?-xism:senderbanp)) => "Keep" => );
$formfiller->add_filler( qr((?-xism:senderpreserve)) => "Keep" => );
$formfiller->add_filler( qr((?-xism:senderforward)) => "Keep" => );
$formfiller->add_filler( qr((?-xism:senderforwardto)) => "Keep" => );
$formfiller->add_filler( qr((?-xism:senderaction)) => "Fixed" => '3');
$formfiller->fill_form($agent->current_form);
$agent->click('submit');
```

This code has some components (e.g., the use of the WWW::Mechanize::FormFiller module) that we haven't explored in this column, but hopefully you get the basic idea that an interactive session can be the basis for automatically generated code that can be further customized.

In this column we've gotten a taste of some of the major features of WWW::Mechanize. But that's only the core of this subject. There's a whole ecosystem of additional add-on modules that has grown up around WWW::Mechanize that I'd encourage you to explore. These modules add features such as timed requests and human-like requests with delays between them (should you want to test your Web site), and experimental JavaScript support. Web work with WWW::Mechanize can be both time-saving and fun, so enjoy! Take care, and I'll see you next time.