COLUMNS

Practical Perl Tools

Dance, Browser, Dance!

DAVID N. BLANK-EDELMAN

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 *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. 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. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@pobox.com

In past columns, we've written code together that contacted Web sites that didn't have an API per se and queried information from them. Tools like HTTP::Tiny, LWP::Simple, Mojo::UserAgent, and WWW::Mechanize have made an appearance in this column (some as recently as the previous column). These are all fantastic tools (some of them more fantastic than others), but if you have felt something was lacking, I can't blame you. With all of these modules, we've sidestepped, for better or worse, the Web browser. This has also meant giving up certain functionality found in the browser—the biggest elephant being JavaScript. People have written code to glue JavaScript engines to WWW::Mechanize (e.g., WWW::Mechanize::PhantomJS) or to drive browsers from these kinds of modules, but they haven't been particularly widespread in their implementation or adoption. In this column, we're going to look at how to use Perl with a framework that lots of people use to drive browsers in a whole range of languages.

The framework we'll be exploring in brief today is called Selenium. It originated from work that people have done to create testing frameworks that used real browsers to construct real tests of Web applications. Let's say you built a Web app and you'd like to make sure that your automated test suite (you have a test suite, right?) actually tests the app's functionality using the same browsers humans will be using when you finally make it available on the Web. Enter Selenium (http://www.seleniumhq.org). But, testing is just one thing you could use this for; driving your browsers (both desktop and mobile) using a script could be applied to all sorts of things.

Before we dive into how to set all this up and get it to rock from Perl, there is a piece of Selenium history worth mentioning so that you don't take a wrong turn while learning about this stuff. Once upon a time, as in version 1, Selenium offered something called Selenium Remote Control (or Selenium RC as you will see it written) as one of its main interfaces.

There were a number of Perl modules written for Selenium 1, and we're not going to touch any of them. Selenium 1 was a bit of a hack (basically it injected JavaScript code that manipulated the browser), so at some point Selenium 2 (sometimes called Selenium WebDriver because that was the name of the other project that merged with Selenium) was born. In this column, we are going to be using a Perl module that works with Selenium 2 only. If you want to dive deeper into this subject by searching the Web for more information, be sure to pay attention to which version of Selenium the resources you find are describing.

Wait, Was that Java I Just Saw Zoom By?

Let's talk about how we get set up to start using Selenium. While there are ways to directly talk to a browser using the WebDriver stuff, the Perl module we're going to be using expects to talk to a standalone Selenium server. That server is written in Java. But, besides needing the JDK installed and running one command, you can pretend I never mentioned that language. Actually, let me be a little bit of a tease and mention that there are companies like Sauce Labs (https://saucelabs.com) that actually provide Selenium as a service so that you could connect to their hosted Selenium infrastructure instead of bringing up your own server. But for our purposes, bringing up a standalone server (vs. an industrial-strength service) is pretty trivial.

First step (providing you have the JDK installed): go to http://www.seleniumhq.org and download the latest stable Selenium server release.

Step 2: start it up like so:

```
java -jar {name of jar file}
```

See, that wasn't so bad. The Java app will produce output that will look roughly like this:

```
  21:52:09.373 INFO - Launching a standalone server
  21:52:09.437 INFO - Java: Apple Inc. 20.65-b04-466.1
  21:52:09.437 INFO - OS: Mac OS X 10.10.1 x86\_64
  21:52:09.458 INFO - v2.44.0, with Core v2.44.0. Built from
revision 76d78cf
  21:52:09.574 INFO - Default driver org.openqa.selenium.
ie.InternetExplorerDriver registration is skipped:
registration capabilities Capabilities \[{platform=WINDOWS,
ensureCleanSession=true, browserName=internet explorer,
version=}] does not match with current platform: MAC
  21:52:09.643 INFO - RemoteWebDriver instances should connect
to: http://127.0.0.1:4444/wd/hub
  21:52:09.644 INFO - Version Jetty/5.1.x
  21:52:09.645 INFO - Started HttpContext\[/selenium-server
/driver,/selenium-server/driver]
  21:52:09.646 INFO - Started HttpContext\[/selenium-server
,/selenium-server]
  21:52:09.646 INFO - Started HttpContext\[/,/]
  21:52:09.717 INFO - Started org.openqa.jetty.jetty.servlet
.ServletHandler@3b6f0be8
  21:52:09.717 INFO - Started HttpContext\[/wd,/wd]
  21:52:09.727 INFO - Started SocketListener on 0.0.0.0:4444
  21:52:09.728 INFO - Started 657576922 org.openqa.jetty.jetty
.Server@7a3570b0
```

This output will be primarily useful to us if we want to check some of the values in use (e.g., what port it is listening on). A number of values can be set on start; to see what is supported, run the following:

```
java -jar {name of jar file} -help
```

Back to Cool, Refreshing Perl

Once you have a Selenium standalone server running, it is time to bring Perl into the picture. The module we are going use is called Selenium::Remote::Driver. It can be a little dependency heavy (48 other modules if installing into a fresh Perl instance—I checked), but with the help of the cpanm command mentioned here in a past column, it is installed with a single command (cpanm Selenium::Remote::Driver) and a bit of thumb-twiddling.

Let's start with a simple script that uses it to tell a browser to fetch a Web page:

```
use Selenium::Remote::Driver;

my $driver = new Selenium::Remote::Driver;
$driver->get('http://www.usenix.org');
print $driver->get_title(),"\n";
$driver->quit();
```

This script can be so bare bones because it is using all of the defaults; we'll talk about modifying them shortly. When we run this script, it is a little creepy because Firefox pops open, loads this page, quits, and then the script prints:

```
Home | USENIX
```

We can see what is going on because the window where we started the standalone server is providing some play-by-play debug output:

```
  11:36:19.871 INFO - Executing: [new session:
  Capabilities [{acceptSslCerts=true,
  browserName=firefox, javascriptEnabled=true, version=,
  platform=ANY}]])
  11:36:19.894 INFO - Creating a new session for Capabilities
  [{acceptSslCerts=true, browserName=firefox,
  javascriptEnabled=true, version=, platform=ANY}]
  11:36:26.008 INFO - Done: [new session: Capabilities
  [{acceptSslCerts=true, browserName=firefox,
  javascriptEnabled=true, version=, platform=ANY}]]
  11:36:26.018 INFO - Executing: [get: http://www.usenix.org])
  11:36:28.186 INFO - Done: [get: http://www.usenix.org]
  11:36:28.192 INFO - Executing: [get title])
  11:36:28.536 INFO - Done: [get title]
  11:36:28.542 INFO - Executing: [delete session: 1abb3d91
-ce4a-426d-8096-b4853cf94197])
  11:36:28.646 INFO - Done: [delete session: 1abb3d91-ce4a
-426d-8096-b4853cf94197]
```

You Can Seek, But First You Have to Find First

Retrieving the title of the page you opened in the browser via Selenium magic is probably not the most useful thing you will want to do (although it can be helpful as part of a larger test suite to make sure the rest of the code's assumptions about which page you're on are correct). Most of the time, you will want to be working with elements on that page, either retrieving them or interacting with them (e.g., filling in forms, performing some kind of navigation).

More often than not, the very first thing you have to do is grab hold of part of the page using one of these two find_ commands:

```
find_element
find_elements
```

There are other similar commands (e.g., get_active_element, which returns the element that has focus), but I find almost all of my scripts include one of those two as the first action after pulling up the page.

Here's where things get a little interesting and where one of the defaults mentioned before comes into play. find_element(s) gives you three different "strategies" (that's the term from the docs) for locating elements:

1. HTML specification (my term). This lets you find an element by id (*id = something* in the source)), class (*class = something*), link, etc.
2. CSS specification. This lets you find an element using the standard CSS selectors as the browser implements them. So, for example, you could specify "div#feature3".
3. XPath specification. Faithful readers of this column know I ♥ XPath for its concision and eloquence. They will also recall that we spent an entire column looking at the XPath syntax and like. So that we don't have to do an entire context swap-in of that info, I'm going to simply say that one can use XPath expressions as another way of selecting elements on a page but not provide other examples of this that need to be explained.

By default, find_element will use #3, XPath. To change that default, each find_element can take a second argument specifying the strategy, or better yet, we can change the default:

```
my $driver =
  Selenium::Remote::Driver->new('default_finder' => 'css');
```

The docs recommend using HTML selectors (#1) by default, as in:

```
my $Webelement = find_element('search-bar','id');
```

because it is the most efficient kind of search, but that assumes you are dealing with Web pages that have well-structured code. I tend to hope for that but expect to have to use one of the other kinds of finders.

Now we know ways to find things, but what happens when we succeed? find_element() will return a WebElement object (or more precisely, a Selenium::Remote::WebElement object) representing the first thing it finds, and find_elements returns an array of them for all of the matches. With this object, we can do a number of things (documented in the Selenium::Remote::WebElement module documentation). Here's some code that will display the names of the main tabs on the page:

```
my (@elements) =
 $driver->find_elements('ul#main-menu-links li a','css');
foreach my $element (@elements){
 print $element->get_text(),"\n";
}
```

It queries for all of the elements that match a particular CSS selector (finds all of the links in the list items of the unordered list with the ID of "main-menu-links") and then displays the text associated with each.

Let's Do Stuff

Selenium has launched a browser for us, so let's start doing browse-y things. First off, we might want to start navigating around the page and clicking on stuff. One thing we could do would be to click on all of the main menu tabs and retrieve the page title for each page we land on. Let's start with code that does not work, because it will illustrate an important point:

```
# this does not work!
my (@elements) =
  $driver->findelements('ul#main-menu-links li a','css');

# the first link is to the current page, skip it
shift @elements;
foreach my $element (@elements){
 $element->click();
 print $driver->gettitle(),"\n";
 $driver->goback();
}
```

This would seem to be the right thing. Find all of the links, click on a link, hit the back button, click on the next link, easy, right? Here's what happens when we run the code:

```
About USENIX | USENIX

   Error while executing command: An element command failed
because the referenced element is no longer attached to the
DOM.: Element not found in the cache - perhaps the page has
changed since it was looked up
   ...
```

It gets the first click/title print right, but bites the dust on the second one. Why is that? In this case, we've clicked to another page before coming back to the home page. When we return to the home page, there's no guarantee that the structure of the page (the DOM to be precise) we return to is exactly the same as the way we left it. Lots of stuff could happen—the source of the page could have been changed, JavaScript could have altered the structure, and so on. Selenium knows we are dealing with essentially a new page, so the references to parts of the old page aren't viable anymore. The best we can do is rerun the find_elements() and pick the next item in a list whose index we retain. Here's code that does work:

```
my (@elements) =
 $driver->find_elements('ul#main-menu-links li a','css');
```

```
for (my $tab = 1; $tab <= $#elements;$tab++){
 $elements[$tab]->click();
 print $driver->get_title(),"\n";
 $driver->goback();
 @elements =
$driver->find_elements('ul#main-menu-links li a','css');
}
```

If we run it, we get the following output:

```
About USENIX | USENIX
Conferences | USENIX
Publications | USENIX
LISA Special Interest Group for Sysadmins | USENIX
Membership & Services | USENIX
Student Programs | USENIX
USENIX | The Advanced Computing Systems Association
```

Basically, we do another find each time we return to the home page and then click on the next tab in the sequence. Long-time programmers are probably reaching for their pitchforks because they can smell a race condition when they see one, so let me cop to it right now. Yup, this code could potentially lead to a race condition. As in the vaudeville skit where the patient says, "Doctor, Doctor, please help me, it hurts when I move my arm like this," the response is "Don't move your arm like that."

Go Forth and Do Cool Stuff

As you can probably guess, Selenium has lots of other actions you can take on a page. You can select elements, you can send key presses, drag and drop, move the mouse around, select different windows, and so on. In addition to the documentation, there are two good tutorials at http://www.slideshare.net/vroom/testing-your-Website-with-selenium-perl and http://desmoines.pm.org/meetings/selenium_july2013.html worth checking out. Enjoy, and we'll see you next time.