ADAM TUROFF

practical Perl

PROGRAMS TO WRITE PROGRAMS

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

■ *ziggy@panix.com*

Testing Web sites can be quite tedious, but it doesn't need to be. In this column I describe how I wrote a few small programs to generate hundreds or thousands of regression tests for a Web site.

One of my current projects involves reimplementing a large, dynamic Web site. Back in the go-go dot-com days, this kind of project was rather common. The preferred technique was slash-and-burn: Throw out all of the old code and implement the new Web site from scratch using whatever language, library, framework, or platform is popular this week. Along the way, the Web site would get a facelift and be upgraded to use the latest and greatest Web design techniques. The new Web site would look wonderful, garner praise, and win awards, up until the point that the cycle repeated, a few short months later.

Thankfully, the Web development scene has settled down considerably in recent years. Throwing *everything* out and reimplementing a Web site from scratch has come to be seen as not only foolish, but in many cases as not even feasible. The current Web site I am working with is the result of years of requirements gathering, design, development, testing, and debugging. It represents a slow evolution to match the application requirements with the capabilities and quirks of the Web browsers our customers use.

Throwing out years of work because some popular new system can generate Web pages "easier" or "better" is interesting but ultimately irrelevant. While the current Web site architecture may be aging and brittle, any new implementation needs to faithfully reproduce the HTML interface in use today. Customers have come to expect the site's current interface and behaviors. Gratuitous changes made for the sole benefit of the development team would negatively impact customers, and that could easily impact the bottom line. Yet some kind of change is necessary whenever the current Web site code becomes hard to maintain and difficult to extend.

Testing to the Rescue

In some respects, this problem seems like the classic "unstoppable force meets immovable object." The existing HTML must be preserved, and the easiest way to generate the existing HTML is to keep the existing Web site, however old and brittle it may be. The only way to move forward is to replace the existing Web site, but only in a manner that will faithfully reimplement the existing HTML, bug for bug. Seen this way, reimplementing this Web site is a software

project like any other, with one additional constraint, which we can check empirically as we move forward. (Fixing HTML bugs and updating HTML designs are discussions for another time and place.)

This constraint sounds cumbersome and tedious, and indeed it is. Each dynamically generated HTML page from the new site must be checked against a corresponding page from the old Web site to check that all expected content, layout, and structure is present, and *only* that material is present. Repeat this process for each of the dozens of pages to be tested. Because pages may appear differently for different users, check each page multiple times, one time for each account to be tested.

Other factors also require consideration. Like many Web sites today, this site is a front end for a very large database. The database is constantly being updated, so a Web page that passed a test yesterday or this morning might fail this afternoon because the underlying data has changed. But that kind of failure is a "false negative," since that is the expected behavior. So tests need to be updated periodically in order to ignore normal data changes, and focus on the HTML interface elements that surround that data.

As my friend brian d foy likes to point out, we're working with computers, and computers are built for doing boring, repetitive work over and over again. On this project, I need hundreds of long test scripts in order to make sure that the new Web site faithfully re-creates the output of the existing one. Rather than writing those boring, tedious test scripts by hand, I decided to write three interesting programs instead:

- find-links: Finds Web pages on the old site to examine
- build-test: Examines a single Web page on the old site and builds a test script
- MyHTMLAnalysis.pm: A module that analyzes HTML input when building and running a test script

By running two programs, I can generate a few hundred test scripts in the time it takes to get a cup of coffee. If the underlying data changes, I can just delete some tests and rebuild them while I get more coffee.

Test Setup

In order to easily compare old and new Web sites, I needed to spend a little time building an environment to support this testing activity. I started out with two identical copies of the Web site checked out from CVS, Subversion, or another source control system. The two copies of the Web site must be configured identically and run side by side on two different Web servers running on two different ports, or on two different systems. One copy will be the "baseline," running the existing code unchanged. The other copy will be the development server, where all of the changes will be made.

Having access to both versions of the code at all times is important. Whenever a test failure occurs on the development server, the baseline server should be checked with the same test script to determine whether the failure is a bug or a false negative due to normal data changes. If a test failure is in fact a false negative, the baseline can be reexamined to produce a fresh test to find real bugs in development.

With the baseline Web server in place, the first task is to find the pages to profile, the task automated by the find-links script. I could have maintained a text file of links to examine, but it was just as easy to write a program to find links for me. In the spirit of automating tedious tasks, this program emits a Makefile fragment that will build the tests.

Below is the find-links script that I used to crawl the baseline site and find all pages linked from the home page. Of course, each Web site is different, so the

rules for what to profile will likely vary from site to site. For my project, it was sufficient to look at the home page and look at any link into the site from the home page. For other sites, it might be necessary to examine a small, predetermined list of links, to perform an exhaustive traversal of every link in a site, or something in between. Note that links to other Web sites are ignored, since they are beyond the scope of what is to be tested here.

```
#!/usr/bin/perl -w
use strict;
use WWW::Mechanize;
my $usage = "Usage: $0 <baseurl> <urlpath> <testpath> [cookiejar]\n";
my $baseurl = shift(@ARGV);
my $urlpath = shift(@ARGV);
my $testpath = shift(@ARGV);
my $cookies = shift(@ARGV) || "";
die $usage unless $baseurl;
die $usage unless $urlpath;
die $usage unless $testpath;
## Specifying a cookie jar is optional
## Find URLs
my %seen;
my $number = "000";
my $mech = WWW::Mechanize->new (
            cookie_jar => {file => $cookies}
          );
my $base = "$baseurl$urlpath";
$mech->get($base);
foreach my $url ($mech->links()) {
    ## Normalize this URL:
    ## convert into an absolute URL
    ## and remove the internal anchor (if present)
    $url = $url->url_abs()->as_string();
    $url =~ s/#.*$//;
    ## Focus on links within this site, and
    ## make the URL relative to the base
    next unless $url =~ s/^$base//;
    ## Test each URL once and only once
    next if $seen{$url}++;
    ## Write out another entry in the Makefile
    my $file = "$testpath/$number.t";
    print "$file:\n\tbuild-test $baseurl $url $cookies > $file\n\n";
    $number++;
}
```

This script is invoked with four parameters: the location of the baseline server (http://localhost:8080), the URL path to the page to profile (/start), a path to deposit test scripts, and an optional file containing the cookies to use for user authentication. The location of the baseline server must be split out from the URL to process, so that build-test can create a test that assesses either the baseline or the development server.

By using this program I can create multiple test suites from the baseline Web site quickly and easily:

```
$ find-links http://localhost:8080 \
    /start ./admin admin.conf > Makefile.admin
$ find-links http://localhost:8080 \
    /start ./user user.conf > Makefile.user
$ find-links http://localhost:8080 \
    /start ./guest guest.conf > Makefile.guest
...
```

Analyzing HTML

Once find-links has run, the next step requires profiling the baseline server to build test scripts. Because these scripts will be used to check both the baseline and the development server, the location of the Web server to test should not be specified in these test scripts. I have found that the best way to specify which server to test is to place that information in environment variables, either in a Makefile or on the command line. Switching or overriding an environment variable makes it quick and easy to check the baseline server to see whether a test failure on the development server is a true bug or a false negative.

Ideally, the best way to test the output of the development server against the baseline server is to use a simple string comparison. If the output of the development server does not precisely match the expected output, the test fails. In practice, that level of rigor is simply impractical. If the output from the development server does not exactly match the output from the baseline server, it could be because one character changed or because 1000 characters changed. Also, locating where and how the two pages differ can be difficult, especially when dealing with very large HTML pages.

Furthermore, there are many textual changes that have no semantic or structural impact in HTML. The tags below are all equivalent in HTML, but fail a simple textual comparison:

```
<img src="button.gif" width="5" height="10">
<img height="10" src="button.gif" width="5">
<IMG HEIGHT='10" SRC="button.gif" WIDTH="5">
<img src='button.gif' height='10' width='5'>
<img src="button.gif"
width="5" height="10" >
```

For any meaningful comparison of baseline against development Web servers, some measure of scanning or parsing HTML output is necessary. If you are profiling an XHTML site or other XML data, you can use any of the many Perl modules for processing XML to aid your analysis. If not, then regular expressions and some of the many HTML parsing modules on CPAN can help you along.

To ease HTML profiling, it's best to put the code to analyze output from the baseline and development servers into a module used by both the build-test script and the test scripts it generates. This module contains code to do things such as find links, images, and JavaScript blocks and produce data structures that are easy to examine when building and running tests.

Purists will note that this setup adds a measure of uncertainty to the testing process. Although this is true, pre-processing HTML before testing it helpsto factor out meaningless differences and focus on the more meaningful changes between versions. Because HTML is such a troublesome format, using an analysis module provides one central place to catalog all of the differences you consider meaningless in your application.

For example, JavaScript <script> blocks *should* have a type="text/javascript" attribute. That attribute may or may not be present. The deprecated language="javascript" attribute may be present. If neither is present, browsers will assume that the content of the <script> block will be JavaScript.

Within a JavaScript block, whitespace characters are (mostly) meaningless. If two JavaScript blocks differ only in indentation, they should be considered identical. JavaScript blocks can also be wrapped with optional HTML comments. If the only difference between two such blocks is the presence/absence of HTML comments, the two blocks should be considered equivalent.

Finally, if two JavaScript blocks really do differ, it doesn't matter where they differ, just that they differ. To simplify test output, I find it useful to pre-process

JavaScript blocks and convert them into MD5 checksums. If two JavaScript blocks differ after all meaningless differences have been factored out, their checksums will differ.

Here is the function in my analysis module that cleans up JavaScript blocks for easy comparison. The analysis sounds complex, but the code is actually rather straightforward:

```
package MyHTMLAnalysis;
use MD5;
sub process_javascript {
    my $html = shift;
    ## Grab JavaScript code. Ignore attributes on the <script> tag
    my @javascript = $html =~ m{<script.*?>(.*?)</script>}sig;
    ## Normalize whitespace
    @javascript = map {s/\s+/ /; s/^\s//; s/\s$//; $_} @javascript;
    ## Remove the leading/trailing comments, if found
    @javascript = map {s{^<!—\s*(.*?)\s*//\s*—>$}{$1}s; $_} @javascript;
    ## Convert it to MD5 checksums
    @javascript = {MD5->hexhash($_)} @javascript;
    return @javascript;
}
```

Analysis functions for other portions of the HTML input are generally simple and easy to write and test on their own. HTML testing requirements generally vary from site to site, so be sure to identify what portions of the HTML input you need to analyze, and what meaningless changes you want to factor out from your tests.

Building Tests

With an HTML analysis module in place, it was time to build and run the scripts that would profile the baseline Web site and test the development Web site.

The process of building a test script was pretty simple. Each analysis function that build-test calls is mirrored with a corresponding call in the test script being generated. All of the results available to build-test are copied into the test script as test assertions using Test::More. Here is the portion of build-test that handles building JavaScript tests:

```
#!/usr/bin/perl -w
use strict;
use MyHTMLAnalysis;
use WWW::Mechanize;

my $usage = "Usage: $0 <base> <url> [cookie jar]\n";

my $base = shift(@ARGV) or die $usage;
my $url = shift(@ARGV) or die $usage;
my $cookies = shift(@ARGV) || ""; ## Cookies are optional
my $mech = WWW::Mechanize->new (
            cookie_jar => {file => $cookies}
          );
$mech->get("$base$url");
my $html = $mech->content();
print preamble($url, $cookies);
print test_javascript($html);
##...create more tests
sub preamble {
    my $url = shift;
    my $cookies = shift;
    return <<EOF;
#!/usr/bin/perl -w
use strict;
```

```
use Test::More qw(no_plan);
use MyHTMLAnalysis;
use WWW::Mechanize;
my \$mech = WWW::Mechanize->new (
          cookie_jar => {file => $cookies}
        );
\$mech->get("\$ENV{TEST_SERVER}$url");
my \$html = \$mech->content();
my \@data;
EOF
}
sub test_javascript {
    my $html = shift;
    my @data = MyHTMLAnalysis::process_javascript($html);
    my @tests;
    push (@tests, q/@data =
MyHTMLAnalysis::process_javascript($html)/);
    ## Test that all of the expected JavaScript blocks match
    foreach (@data) {
        push (@tests, qq/is(shift(\@data), q{$_});/);
    }
    ## Make sure there are no other JavaScript blocks
    pushd (@tests, <<EOT);
foreach (@data) {
    fail("Unexpected Javascript block: $_");
}
EOT
    return join("\n", @tests);
}
```

The snippet above shows how to test JavaScript blocks in a Web page. The process can easily be repeated to test more components on a Web page by adding more analysis functions to the shared analysis module, calling them in this script, and embedding the results of that analysis into the test scripts generated.

Note that this program is actually producing a Perl program (a test script), so it is important to get the quoting correct: Some variables need to be escaped because they are variables in the test script being generated. Other variables are unescaped because they are variables in build-test, where the values are being copied into the test script. The resulting program can be run using standard testing tools like Test::Harness and prove.

Finally, keep in mind that these scripts must be able to examine either the baseline or the development server. The location of the server to test is expected to be in the TEST_SERVER environment variable, and that will generally point to the development server (e.g., http://localhost:8081). When checking for changes in the database, this value would be reset to point to the baseline server (e.g., http://localhost:8080).

Conclusion

Testing Web sites is a notoriously difficult and error-prone task, but with a little advance planning and analysis, Web site testing can be a breeze. Just write a few programs to profile your Web site, and let Perl generate your test scripts for you.