

An Introduction to Testing Web Applications with twill and Selenium

by C. Titus Brown, Grig Gheorghiu, and Jason R. Huggins

Copyright © 2007 O'Reilly Media, Inc.

ISBN: 978-0-596-52780-8

Released: June 18, 2007

This Short Cut is an introduction to building automated web tests using two tools, twill and Selenium. twill is a simple web scripting language that can be used to automate web tests, while Selenium is a web testing framework that runs in any browser and can be used to test complex web sites that make extensive use of JavaScript.

The best way to use this Short Cut is to run through the examples. We expect that within an hour you can start writing your own functional tests in either twill or Selenium, and within a day you will understand most, if not all, of the possibilities and the limitations of these tools.

Contents

Introduction	2
Functional Web Testing with twill	4
Functional Web Testing with Selenium	28
Testing a Simple Web Application with twill and Selenium	44



Introduction

Welcome!

This Short Cut is an introduction to building automated web tests using two tools, twill and Selenium. twill is a simple web scripting language that can be used to automate web tests, while Selenium is a web testing framework that runs in any browser and can be used to test complex web sites that make extensive use of JavaScript.

The best way to use this Short Cut is to run through the examples. We expect that within an hour you can start writing your own functional tests in either twill or Selenium, and within a day you will understand most, if not all, of the possibilities and the limitations of these tools.

This book is divided into three parts. First we introduce twill and show how you can browse the Web and test basic web applications with twill. Next we introduce Selenium and discuss using Selenium to test sites that use JavaScript and Ajax technologies. Finally, we build some tests for a simple Django application using both twill and Selenium, and show how to integrate these tests into your development.

What Don't We Cover?

This short cut is not a HOWTO on testing.

That is, this short cut is meant as an introduction to *technical solutions* for web test automation. We, the authors, have found web test automation to be a critical part of designing, writing, and deploying reliable web applications, and we favor the use of agile software development and agile testing strategies. However, a detailed discussion of these strategies is not our plan.

Note

We recommend the following introductory books on testing: **Lessons Learned in Software Testing** by Cem Kaner, et al. (Wiley); **Testing Computer Software** by Cem Kaner, et al. (Wiley); **A Practitioner's Guide to Software Test Design** by Lee Copeland (Artech House Publishers); and **How to Break Software** by James A. Whittaker (Addison-Wesley).

What Are twill and Selenium? And Why Not...

twill is an HTTP protocol driver that mimics a command-line web browser. It "speaks" HTTP and lets the user interactively browse applications over the com-

mand line. twill also lets the user *script* the web access. twill doesn't understand JavaScript, however, and so it can't test all client-side functionality.

Selenium is an in-browser programming system. It lets you drive web browsers using either a simple table-based language or a remote control protocol. Because Selenium works *within* the browser, it can test client-side JavaScript quite easily. Moreover, because it runs within all of the most popular web browsers, it can be used to test your client-side JavaScript functionality in *all* of those browsers, thus ensuring that your site *works* in all browsers.

We hope that both the possibilities and drawbacks of using twill and Selenium will become clear to you after you read this short cut.

Why do we cover only twill and Selenium? There are many, many other web testing tools out there. We are covering these two, and only these two, for several reasons:

1. We wrote them, so we are familiar with them and comfortable with their design, functionality, and limitations.
2. twill and Selenium neatly bracket our testing needs. twill is a lightweight testing solution that is easy to use, easy to automate, easy to execute, simple to extend in Python, platform independent, and capable of testing almost all HTTP functionality out of the box. Selenium is a more heavyweight testing solution that can test almost any web site, in most browsers, on any platform.
3. Both twill and Selenium are open source. By using them you are "buying" into a *community* rather than a company. You will never get locked into these tools for the wrong reasons, and you will spend no money trying them out. We expect that this short cut will let you evaluate both tools in less than a day and make an informed decision as to whether or not they fit your needs.
4. Most other web testing tools are either platform- or browser-specific. Neither twill nor Selenium limit you to a specific platform or (in the case of Selenium) any single browser. Moreover, both tools are language-neutral: twill is written in Python, but can be used with web sites written in any language. Selenium Core implements its own language, based on JavaScript, while Selenium Remote Control works in almost any language, and so Selenium can be used to test a dynamic web site written in any language.

Two other open source web test automation tools that satisfy most of these criteria (except for the first one: we are not their authors!) are **Watir** [<http://wtr.rubyforge.org/>] and **Sahi** [<http://sahi.sourceforge.net/>].

Watir is a Ruby-based system for driving Internet Explorer, and it also works with Firefox. It is a very mature system for web testing, and we urge you to try it out. It cannot be used for cross-browser testing as easily as Selenium, however.

Sahi uses a proxy-based technique to drive browsers using JavaScript. It "competes" directly with Selenium. It is much newer, and it may be as powerful. Currently, we don't know of any strong reason to choose Sahi over Selenium, but we are not very familiar with it, either; it emerged as we were writing this short cut.

Contacting the authors

The authors can be contacted directly at webtesting@idyll.org, or you can ask more general questions about testing on the [testing-in-python](mailto:testing-in-python@lists.idyll.org) mailing list, tip@lists.idyll.org. See <http://webtesting.idyll.org/resources> for more information.

Acknowledgements

We would like to especially thank Jacob Kaplan-Moss for reviewing a draft of this short cut!

Functional Web Testing with twill

A brief history of twill

(by C. Titus Brown)

I first began looking for a way to automate web testing in 2003. I had two unmanageably large web sites being used by hundreds of people, and my default testing strategy for them was to add new features, make them public, and wait for bug reports. This wasn't very satisfactory for either me or the users, needless to say!

Around that time, I ran across Cory Dodt's PBP, a web scripting language built on top of John J. Lee's mechanize package, which let Python "speak" HTTP and understand web pages. (John's mechanize package was a Python port of Perl's WWW::Mechanize, and PBP looked somewhat like Perl's WWW::Mechanize::Shell.) Unfortunately, at that time PBP was in its very early stages and didn't work very well on my web sites.

In late 2004, I decided to investigate PBP again, because I really needed to write some automated web tests. By then PBP had matured considerably, and it got me about 90 percent of the way to a web testing solution for my web sites. I started submitting patches to Cory to make it a 100 percent solution. Eventually, however, we realized that Cory was interested in taking PBP in a different direction. I'd also

become disenchanted with PBP's architecture, which relied on the sprawling Twisted library for certain functions. (PBP was born as a way to write functional tests for `twisted.web`, so this inter-reliance wasn't surprising.)

I set out to rewrite PBP from scratch. My initial rewrite used an IPython-based shell, but I soon discovered that IPython, while excellent for interactive Python execution, wasn't very good at handling *scripts*. In particular, the shell's exception handling *could not* be overridden. So I switched to using `cmd`, a framework for writing command-line style applications that comes with Python itself. This is what `twill` uses today.

Somewhat to my surprise, `twill` was an immediate minor success. A number of people picked it up and wrote to me about it, and soon bug reports and feature requests came flowing in.

Since the rewrite to use `cmd`, the architecture of `twill` has changed very little. `twill` is basically glue code connecting `mechanize` to `cmd`.

Some day soon I even hope to get back to testing the two web sites that motivated me to write `twill` in the first place...

What Is `twill`?

`twill` is a simple language for web testing and web browsing. The `twill` shell mimics the function of a browser as much as possible, but with a command-line interface rather than with a graphical interface. The purpose of `twill` is to allow easy automation of most client-side web interactions, and to provide extension facilities for dealing with more complex needs.

All of our examples below are for testing web sites written in Python. `twill` works just fine with sites written in Java, PHP, Perl, and Ruby—although there are some special tricks you can use to test Python sites with `twill`.

This section starts by introducing the `twill` language and demonstrating some simple web testing techniques. While that's all you'll need to get started with your own web tests, we'll also talk about how to organize your web tests so that running them and maintaining them is easier. After that, we'll introduce the `scotch` recorder, software that sits between your browser and a web site and records web traffic; `scotch` lets you record your path through a web site, and then replay it or save it as a `twill` script.

The second half of the section discusses some more advanced `twill` topics, including the extension functions that are included with `twill`, and how to extend `twill` yourself using Python.

Introducing twill

The easiest way to demonstrate twill is to show it in action. First, install twill (see <http://twill.idyll.org> for instructions). Then run 'twill-sh', and type:

```
go http://webtesting.idyll.org/
```

followed by:

```
show
```

You'll see this output:

```
=> blah
Hello, world!
<p>
<a href="link1">top link</a>
<p>
<a href="link2">another link</a>
<p>
...
```

(You can save the output of `show` into a file with `save_html <filename>.`)

Try the 'showlinks' command: you'll see a list of links on the page:

Links:

```
0. top link ==> link1
1. another link ==> link2
2. JavaScript alert box ==> JavaScript:%20alert_it()
```

Similarly, the 'showforms' command shows a list of the forms on the page:

```
Form #1
## ##  __Name__  __Type__  __ID__  __Value__
1   num1      text      (None)
2   num2      text      (None)
```

These are simple examples of the twill language, showing you how you can load specific web pages and look at the raw HTML, the links on the page, and the forms on the page.

(Note: From here on out, we mark the text to enter into twill by '>>' in the example, followed by example output.)

It's pretty easy to follow the links:

```
>> follow "top link"
==> at http://webtesting.idyll.org/link1
```

You can also go back. This goes to the previous page without actually request it again from the web server; to request the page again, use `reload`:

```
>> back
==> back to http://webtesting.idyll.org/
>> reload
==> reloaded
```

If you want to see where `back` is going to take you, you can look at the history:

```
>> go http://webtesting.idyll.org/
==> at http://webtesting.idyll.org/
>> showhistory
History: (1 pages total)
      1. http://webtesting.idyll.org/
```

You can reset the history by executing `reset_browser`. This also resets the cookie jar and some of the debug options.

`twill` automatically follows redirects, too; for example:

```
>> go thispage
==> at http://webtesting.idyll.org/thatpage
>> echo __url__
http://webtesting.idyll.org/thatpage
```

will show that you actually end up at 'thatpage'.

Relative URLs work as expected:

```
>> go /
==> at http://webtesting.idyll.org/
>> go subpage
==> at http://webtesting.idyll.org/subpage/
>> go subpage2
==> at http://webtesting.idyll.org/subpage/subpage2
```

Bottom line: browsing the web with `twill` should work the same way browsing the Web with your browser works.

Filling out forms

Suppose we have a login form that we want to fill out:

```
>> go /login
==> at http://webtesting.idyll.org/login
>> showforms
Form #1
```

##	##	__Name__	__Type__	__ID__	__Value__
1		username	text	(None)	
2		password	password	(None)	
3	1	login	submit	(None)	log in
4	2	create	submit	(None)	create account

This form has the usual username/password/submit button, as well as a 'create new account' button. Since we don't have an account yet, let's create one by clicking on the 'create' button:

```
>> submit create
Note: submit is using submit button: name="create", value="create account"
>> showforms
Form #1
## ##  __Name_____  __Type___  __ID_____  __Value_____
1   name                text      (None)
2   gender               select   (None)      ['male'] of ['male', 'female']
3   username             text      (None)
4   password             password (None)
5  1  create              submit   (None)      create
```

Now it's asking us for a name, gender (from a pull-down menu), account name, and password. Let's fill out this form, using the 'formvalue' command to fill out the fields displayed by 'showforms':

```
>> formvalue 1 name "Ted Meister"
>> formvalue 1 gender male
>> formvalue 1 username ted
>> formvalue 1 password secret
>> submit
Note: submit is using submit button: name="create", value="create"
>> show
<p>
<form method='POST'>
Username: <input type='text' name='username'> <p>
Password: <input type='password' name='password'> <p>

<input type='submit' name='login' value='log in'>
<input type='submit' name='create' value='create account'>
</form>
```

All right! Our account is created—let's go log in:

```
>> showforms
Form #1
## ##  __Name_____  __Type___  __ID_____  __Value_____
1   username            text      (None)
2   password            password (None)
3  1  login              submit   (None)      log in
4  2  create             submit   (None)      create account
>> fv 1 username ted
>> fv 1 password secret
>> submit
Note: submit is using submit button: name="login", value="log in"
```

Using 'formvalue' is tiresome, so in fine UNIX tradition we're using the two-letter alias 'fv'. Note that the name of the field given to 'formvalue' can be the field name (<input name=...>) or the id (<input id=...>), if a field id is present in the form.

We're logged in; let's see if we can get access to the restricted area:

```
>> go /restricted
==> at http://webtesting.idyll.org/restricted
>> show
Welcome, ted
```

Yep, we can—it all works!

As the section above demonstrates, twill happily fills out forms, follows redirects, and otherwise browses the Web at your command. It's up to you to test that your expectations are met; however, you need to make sure that the page contains what you think it should, and that redirects and access restrictions are properly handled.

Cookie management

Implicit in the last example (accessing the restricted area) is the use of a cookie jar: twill keeps track of cookies for you.

There are a few commands specifically for dealing with cookies. `show_cookies` shows the current list of cookies, while `save_cookies <filename>` saves the current set of cookies into the given filename. `load_cookies <filename>` empties the cookie jar, deleting all cookies, and then loads the cookies in the given filename. `clear_cookies` deletes all of the cookies in the cookie jar.

Note that cookies aren't saved when you exit twill; you need to explicitly issue a `save_cookies` command if you want to keep them.

One nice feature of twill is that you can use multiple identities by loading and saving cookies. For example:

```
>> go /whoami
==> at http://webtesting.idyll.org/whoami
>> show
Username: ted<p>Name: Ted Meister<p>Gender: male<p>
>> save_cookies ted.txt
>> go /login
==> at http://webtesting.idyll.org/login
>> submit create
Note: submit is using submit button: name="create", value="create account"
>> formvalue 1 name "Sally Sitter"
>> formvalue 1 gender female
>> formvalue 1 username sally
>> formvalue 1 password hidden
>> submit
Note: submit is using submit button: name="create", value="create"
>> go /whoami
==> at http://webtesting.idyll.org/whoami
>> show
Username: ted<p>Name: Ted Meister<p>Gender: male<p>
>> save_cookies sally.txt
>> load_cookies ted.txt
```

```
>> go /whoami
==> at http://webtesting.idyll.org/whoami
```

Testing for success

twill was built around the need for testing web sites. Now, in order to *test*, you need to be able to *assert* conditions. That is, you may want to make sure that a page loads correctly, or that a login command succeeds. twill contains a few useful commands for this purpose.

The first is `code <n>`, which just requires that the last HTTP result code is `<n>`. For example:

```
>> go /
==> at http://webtesting.idyll.org/
>> code 200
```

makes sure that the top-level page returns a 'success' code, HTTP code 200. The `code` assertion is particularly useful when testing page loads and access restrictions; for example:

```
>> go /logout
==> at http://webtesting.idyll.org/
>> go /restricted
==> at http://webtesting.idyll.org/restricted
>> code 403
```

will test to make sure that the 'restricted' area is off-limits to people who aren't logged in, by checking for an HTTP code of 403 (unauthorized).

You can also use regular expressions to match against the URL, page title, or page text:

```
>> go /sometext
==> at http://webtesting.idyll.org/sometext
>> url text
>> title some
title is 'some title, n*t some other title'.
>> find elmo
```

All of these commands produce no output unless they *fail*, in which case they complain (and abort script execution, if run as part of a script; see the [Scripts](#) section).

Variables

twill has two kinds of variables, those starting with `__` and those starting with `$`. Generally `__` variables are variables set by the system, while `$` variables are variables set by the user.

The four "system" variables that are set by twill are:

- `__url__`, the current URL
- `__input__`, the result of the last call to `getinput`
- `__password__`, the result of the last call to `getpassword`
- `__match__`, the first matching subgroup (or entire matching group) in a call to `find`, `url`, or `title`

User variables are very useful. You can set them with `setglobal` and `setlocal`:

```
>> setglobal test1 "howdy, pardner"
>> setlocal test2 "hello"
>> echo $test1
howdy, pardner
>> echo $test2
hello
```

'global' and 'local' variables are only different when you're running twill scripts (see below); global variables are accessible across all twill scripts, while local variables are only set within a particular script's scope.

You can also interpolate the variables in any string:

```
>> setglobal expected elmo
>> go /sometext
==> at http://webtesting.idyll.org/sometext
>> find "the word ${expected}"
```

`echo` can be a useful way to see exactly what a variable expands to:

```
>> echo "expected is ${expected}"
expected is elmo
```

Using the shell

The twill shell has tab completion; for example, if you type `'show_c'` and then hit tab, it completes the command into `'show_cookies'`. Tab completion also works on form names and form values.

You can also browse through the command history with up and down arrows, as well as the usual 'readline'/emacs-style editing commands like Ctrl-A (beginning of line) and Ctrl-E (end of line).

The twill shell history is saved in (and loaded from) a file `.twill-history` in the current working directory, so your command history is retained across twill shell sessions.

Command reference and help

You can view all of the available commands with `'help'`, and get more information for each command with `help <command>`. For example:

```
>> help go
=====
```

Help for command go:

```
>> go <url>
```

Visit the URL given.

```
=====
```

Help (and tab completion) work with any commands loaded from extension modules, too.

The twill command line

Run `twill-sh -h` to see a list of options. Other than those useful when running scripts the most useful option is the `-u` or `--url` option, which specifies a URL at which to start. This is very useful when you want to run the same script on multiple web sites. For example, if you have a personal developer web site to test, and want to run the same tests on a staging site, you can use `-u` to avoid hardcoding the site URL into the script.

Scripts

All of the commands that we introduced above work in much the same way when placed in a script file. Just open a text file with a text editor and enter the commands; then run the file with `twill-sh <filename>`. The one difference is that when a command *fails*, e.g., due to a failed assertion, or a web page not loading, the script execution is aborted and the remaining commands in the script are not run. (You can override this behavior by specifying `-n` or `--never-fail` on the command line.)

You can run multiple script files by specifying them all on the command line: `twill-sh <file1> <file2> <file3>`. If given multiple files, twill runs all of them and at the end tallies up the ones that finished successfully and lists the ones that failed. This is particularly useful for ensuring that your web sites are up:

```
go /some/web/site
code 200
```

in a file, one file for each web site, and run `twill-sh *`.

It's also possible to pass arguments into twill shell scripts by doing

```
twill-sh <file1> <file2> -- <arg1> <arg2>
```

See the 'argparse' extension module, below, for more information on extracting command-line arguments.

One particularly useful trick in scripts is to make use of the initial URL (the URL specified by `-u`). You can save this URL in a global variable, so that it's available to scripts:

```
# at the top of a script
>> setglobal initial_url __url__

# some time later, return to the initial URL
>> go $initial_url
==> at http://webtesting.idyll.org/sometext
```

More Advanced twill Features

Now that we're past the basics of using twill, here is a look at some of the issues you'll run into as well as some of the more advanced options available to you. Even if you are already familiar with twill, it is probably worth at least skimming this section!

runfile and run: Executing other scripts and Python commands

`runfile` and `run` are two very useful commands.

`runfile` runs a twill script. Among other uses, you can use it to help organize your tests (see the [Organizing your twill tests](#) section) into reusable script files that are run from a master script.

`run` executes an arbitrary Python command using Python's `exec` statement. `run` obviates the need to write extension commands for simple Python one-liners; for example, you can use this to set an environment variable:

```
>> run "import os; os.environ['VAR'] = 'hello';"
```

String and argument handling

As you may have noticed, you only need to quote arguments to twill commands when they include spaces. For example, both:

```
>> fv 1 greeting hi
```

and:

```
>> fv 1 greeting "hi"
```

fill in the 'greeting' form element with the string "hi", but:

```
>> fv 1 greeting hello world
```

will not work, because `fv` takes precisely three arguments: the form, the form element, and the value to place in the form element. Instead, you would need to do:

```
>> fv 1 greeting "hello, world"
```

Note that the "n" character is interpreted as a newline, so:

```
>> fv 1 textbox "hello, \n world"
```

would put a two-line string into textbox.

Debugging options

The `debug` command turns on debugging output for various actions. `debug http 1` turns on HTTP debugging, which is very useful for figuring out exactly what is sent to the server and received back:

```
>> debug http 1
DEBUG: setting http debugging to level 1
>> go /login
connect: (webtesting.idyll.org, 80)
send: 'GET /login HTTP/1.1\r\nAccept-Encoding: identity\r\nHost: webtesting.idyll.org\r\nConnection
reply: 'HTTP/1.1 200 OK\r\n'
header: Date: Sun, 03 Jun 2007 20:47:07 GMT
header: Server: Apache/2.2.3 (Debian) DAV/2 SVN/1.4.2 mod_scgi/1.11
header: Expires: -1
header: Content-Length: 248
header: Connection: close
header: Content-Type: text/html; charset=iso-8859-1
==> at http://webtesting.idyll.org/login
```

`debug commands +` turns on commands debugging. If you place the following in a script 'test':

```
debug commands +
echo "hello, world"
```

then twill prints out each command as it's running:

```
% twill-sh -f test
>> EXECUTING FILE tst
twill: executing cmd 'echo "hello, world"'
hello, world
--
1 of 1 files SUCCEEDED.
```

This can be very useful when running complex scripts.

Another very useful trick for debugging scripts is to use the `-f` or `--fail` option when running scripts. With this turned on, the first script to fail will yield a Python traceback which may help you pinpoint where things went awry. Note that line numbers in twill scripts are printed out in the tracebacks, too: if you put a failing command in a script:

```
run "assert 0"
```

and run the script with `-f`:

```
>> EXECUTING FILE tst
EXCEPTION raised at line 1 of 'tst'
```

```
run "assert 0"
```

Error message: ''

Traceback (most recent call last):

```
File "/System/Library/Frameworks/Python.framework/Versions/2.3/bin/twill-sh", line 7, in ?
    sys.exit(
```

```
...
File "tst:1", line 0, in ?
```

```
...
AssertionError
```

then you'll see the failing test script and line number in the traceback ('tst:1' in the output above).

You can also turn on debugging of META HTTP EQUIV-REFRESH headers; see below.

Output control

twill is intentionally verbose; it's not meant for strong, silent web browsing. However, you can use `redirect_output` to redirect all normal output to a file:

```
>> redirect_output output.log
```

You can also redirect all of the error output:

```
>> redirect_error error.log
```

The commands `reset_output` and `reset_error` resume the normal reporting of output and error messages.

(Note that these commands do not reassign all stdout and stderr output—they only capture output from twill itself.)

Miscellaneous HTTP issues

There are a host of annoying HTTP issues that you may need to deal with. twill can handle many of them.

Browser agent string

Some sites want you to be running a respectable browser, like Mozilla/Firefox or Internet Explorer. By default, twill sends its own agent string, which can confuse these sites. You can use the `agent` command to tell twill to represent itself as another browser. For example:

```
gt;> agent ie5
```

will tell twill to "pretend" to be Internet Explorer 5, and:

```
>> agent opera7
```

See `help agent` for a list of all of the agent string shortcuts built into twill.

You can also write your own agent strings:

```
>> agent testing/using-twill
```

This might be useful when testing a live production server: errors stemming from twill browsing could be logged to a special place by the server.

If blank pages are being returned to twill, or the server is aborting connections in the middle of a transaction, then you should try changing the browser agent string.

HTML parsing

twill is as vulnerable to bad HTML pages as any other program. If HTML parsing isn't working as you would like (or expect), you can try solving it in one of two ways.

First, try installing Tidy (<http://tidy.sf.net/>), a program that cleans up and regularizes "bad" HTML. If Tidy is installed, twill will detect it and pass all HTML pages through it. This can fix badly coded forms and links.

Alternatively, if Tidy is *already* installed, your HTML code could be getting cleaned up in unexpected ways. One example is textarea fields: Tidy will automatically insert a newline into empty textarea fields, for no good reason.

By and large, HTML parsing problems should arise only when using twill on sites that you don't control. (If you're testing the site, you must *control* the site in some sense, right? So you should fix the HTML!)

However, if you insist, you can incrementally turn off tolerant HTML parsing with two config options. (Both of these config parameters are on by default.)

```
# fail on pages that cannot be read by Python's HTMLParser
>> config use_BeautifulSoup -
```

```
# don't fix pages with the tidy preprocessor
>> config use_tidy -
```

You can also use the assertion `tidy_ok` to explicitly check that `tidy` reports no errors on the current page.

To *require* Tidy preprocessing, issue this command:

```
>> config require_tidy 1
```

Normally twill simply doesn't run Tidy when it's not present; this option *requires* Tidy's presence, and page loads will fail if Tidy is not available.

HTTP-EQUIV=REFRESH support

twill has some support for the META HTTP-EQUIV=REFRESH header: refresh headers are followed, but no delays will occur even if a delay is specified. By default, twill will follow refreshes five times; after five reloads of a particular page, however, twill stops reloading that page. (This is to prevent infinite loops in situations where the page would normally be updating with a pause.)

To see exactly what's happening when twill handles an HTTP-EQUIV=REFRESH, run:

```
>> debug equiv-refresh 1
DEBUG: setting equiv-refresh debugging to level 1
```

HTTP authentication

twill supports HTTP basic authentication with the `add_auth` command:

```
>> add_auth realm url user password
Added auth info: realm 'realm' / URI 'url' / user 'user'
```

(You can get the authentication realm by visiting the access-restricted web page with a web browser and looking at the pop-up box that asks for username/password.)

You can also ask twill to use the default realm by executing:

```
>> config with_default_realm 1
```

Organizing your twill tests

Suppose that you have a number of twill tests, and you want to be able to run them quickly and easily—preferably with a single command—and perhaps on a regular basis. How should you organize them?

There are a number of options that we have used on various projects. They all have advantages and disadvantages, of course; the trick is picking which one suits your needs best.

The zeroth option (which we do not recommend!) is to write one long script containing your tests. The problem with this is that the first assertion command to fail causes your entire script to fail, even if most of the tests are independent of each other and should succeed or fail independently. This kind of test is also more difficult to maintain.

If your tests are independent of each other, and do their own setup (preloading the web site with users, for example) and teardown (clearing the users), you can simply place them all in some directory and run `twill-sh <directory>`. twill produces a simple report after executing all of the scripts under that directory, summarizing

how many scripts finished without error and which scripts failed, if any. This scheme works perfectly well, but it's only possible when you have completely independent tests.

If your tests *aren't* independent of each other, and you need to run them in a particular order, you can put them all in a directory in lexical order:

```
01-setup
02-test-login
03-test-user-permissions
04-check-html
05-teardown
```

Then run 'twill-sh' on that directory.

This works well for projects that have a medium-sized collection of functional tests that may need to be run against multiple servers.

Note that in both of the above layouts, you can specify an initial URL with the `-u` option to `twill-sh`. `twill` then loads that URL prior to executing *each script*, so you can point the `twill` scripts at different web sites quite easily. You can also use the 'argparse' extension module, discussed below, to pass arguments like starting URLs into each script.

The problem with both of these layouts is that you're left with a directory full of tests, and that isn't a good way to organize a large number of functional tests.

A third option is to use a 'master' script runner that executes other `twill` scripts in the right order:

```
-- (file 'master-script-runner')
runfile 01-setup
runfile 02-test-login
runfile 03-test-user-permissions
runfile 04-check-html
runfile 05-teardown
--
```

This is more flexible than either of the first two layouts, because you can run tests in an arbitrary order. This lets you reuse scripts; for example, the `01-setup` and `05-teardown` scripts could be run multiple times by putting in multiple `runfile` commands.

A twist on this option is to use a shell script or batch file to run `twill` on the tests in the right order. However, by exiting `twill` you lose all of the global variables and custom configuration settings, so it can be more convenient to run everything from within a single `twill` session.

None of these options work well for projects that have more than a screenful of test files; at that point you probably want to have a directory hierarchy. And thus, our final layout suggestion: using a framework of some kind.

Using a framework to organize and run your twill tests

Our recommended setup for medium-size projects on up is to organize your tests inside of another framework. One option is to use a unit test discovery and execution framework, such as `py.test` or `nose`. Another option is to use `FitNesse`, which is a useful way to present twill tests to customers. The last chapter of this book also contains an demonstration of using `unittest` to automatically start the server and run the tests. Here's a simple example of how you might organize a simple set of tests:

```
def setUp():
    createDatabase()
    startWebServer()

def test1():
    twill.execute_file('test-login', initial_url='http://webtesting.idyll.org')

def test2():
    twill.execute_file('test-roles', initial_url='http://webtesting.idyll.org')

def test1():
    twill.execute_file('test-admin', initial_url='http://webtesting.idyll.org')

def tearDown():
    stopWebServer()
    destroyDatabase()
```

The advantages of using a framework for running twill tests are myriad. First, frameworks usually let you specify set up and tear-down fixtures (as in the example above), which let you set up and finalize resources like web servers and databases. Second, odds are you already have such a framework in place, either for running your unit tests or for running acceptance tests. Third, you have one less framework to add into your continuous integration software. The biggest advantage, however, is probably *standardization*—simply put, a framework lets you specify unambiguously where tests are and (when it matters) what order they're run in.

Recording tests with scotch

There are several reasons why you might want to record from your browser and save into twill scripts. You may want to jump-start your test scripts by recording a web session; you could have many forms to submit; you may simply want to record exploratory testing so that you can reproduce any problems you find.

Whatever the reason, if you want to record a web session, we recommend using the scotch project's recorder. The scotch recorder establishes an HTTP proxy server that records all of your web traffic, including form submission and cookies. This recording can then be saved into a file, and either replayed directly from that file or converted into a twill script.

You can **download scotch here** [<http://darcs.idyll.org/~t/projects/scotch/doc/README.html>].

The following is a short recording of the login/create process for the example server.

The recorder output displays each client request and the response from the server, excepting only non-form POST content:

```
% run-recording-proxy -p 9000

** scotch proxy server running on 127.0.0.1 port 9000 ...
** RECORDING to filename 'recording.pickle'

REQUEST ==> http://webtesting.idyll.org/login
(post form)
      create: "create account"

++ RESPONSE: 302 Moved Temporarily
++ (80 bytes of content returned)
++ (response is text/plain; charset=iso-8859-1)
(# 3)
REQUEST ==> http://webtesting.idyll.org/create

++ RESPONSE: 200 OK
++ (328 bytes of content returned)
++ (response is text/html; charset=iso-8859-1)
(# 4)
REQUEST ==> http://webtesting.idyll.org/create
(post form)
      name: "Ted Meister"
      gender: "male"
      username: "ted"
      password: "secret"
      create: "create"

++ RESPONSE: 302 Moved Temporarily
++ (79 bytes of content returned)
++ (response is text/plain; charset=iso-8859-1)
(# 5)
REQUEST ==> http://webtesting.idyll.org/login

++ RESPONSE: 200 OK
```

```
++ (249 bytes of content returned)
++ (response is text/html; charset=iso-8859-1)
(# 6)
```

This pickled file can then be translated into a twill script:

```
% bin/translate-recording-to-twill recording.pickle

# record 0
#   referer = http://webtesting.idyll.org/create
go http://webtesting.idyll.org/login

# record 2
#   referer = http://webtesting.idyll.org/login
fv 1 create 'create account'
submit

# record 4
#   referer = http://webtesting.idyll.org/create
fv 1 username 'ted'
fv 1 gender 'male'
fv 1 create 'create'
fv 1 password 'secret'
fv 1 name 'Ted Meister'
submit
```

scotch is probably most useful as a way of recording form values and "paths" through web sites; and, of course, for testing purposes you still need to add in assertions!

scotch is still in an early phase of development. If you find it useful but buggy, please be sure to look at the most recent version mentioned on twill.idyll.org [<http://twill.idyll.org/>], and please report any bugs you find or problems you have.

twill Extensions

twill comes with a number of useful extensions. The source code for these extensions can be found in the `twill/extensions/` directory within the distribution; they are installed along with the rest of twill by default.

check_links: a simple link checker

`check_links` is a simple link checker that automatically visits all of the HTML links on the current page and makes sure that each one can be visited successfully (i.e., with an HTTP result code of 200).

To use `check_links`, do:

```
>> extend_with check_links
Imported extension module 'check_links'.
```

```
(at .../twill/check_links.py)
>> check_links
```

You may not want to check all of the links on a particular page, e.g., if some of them are to external sites that forbid automated browsing. If a regular expression is passed as an argument to `check_links`, it only checks pages that match that regexp. For example,

```
>> check_links webtesting.idyll.org
```

will only check links on *webtesting.idyll.org*.

By default `check_links` raises an assertion on a bad link, which will cause scripts to fail. This may not be desirable in a situation where you're checking links on all the pages you visit and you only want to see the bad links at the very end of the script. There's a configuration option that can be set to alter this behavior:

```
>> config check_links.only_collect_bad_links +
```

Turning this option on prevents `check_links` from reporting bad pages until you issue the command `report_bad_links`. This option is particularly useful when used in conjunction with the `require` extension, below.

require: making assertions about pages

When debugging a web site or writing a twill script, you may frequently forget to verify that the page loaded successfully with a `code 200`. If the majority of pages *should* load successfully, you can use the `require` extension to assert a `code 200` after each page:

```
>> extend_with require
Imported extension module 'require'.
(at .../twill/require.pyc)
>> require success
```

Once this command is executed, twill silently issues a `code 200` after each page load. You can turn off this behavior with the `no_require` command.

Where pages should fail (for example, you're testing restricted pages) you can use the `skip_require` command to ignore the result of the next page load (i.e., `go`, `back`, `reload`, `follow`, and `submit`):

```
>> skip_require
>> go /failure
==> at http://webtesting.idyll.org/failure
```

You can also require that all pages contain good links, with the `links_ok` argument to `require`:

```
>> require links_ok
```

Unlike the `check_links` module, `require_links_ok` only visits links once to check if they can be loaded; a cached list of the already visited links is retained. This speeds up browsing, but may miss errors in particularly complex sites.

You can flush this cache with `flush_visited`.

csv_iterate: Iterating over values in CSV files

The `csv_iterate` command gives you a way to iterate over many values. This can be useful for situations where you need to try out many different combinations of form input parameters, for example.

To use `csv_iterate`, first write a script that fills out the form with values in the CSV file, using variables `$col1`, `$col2`, etc. Then call `csv_iterate`:

```
>> extend_with csv_iterate
    Imported extension module 'csv_iterate'.
    (at /System/Library/Frameworks/Python.framework/Versions/2.3/lib/python2.3/site-packages/twill-0.8
>> csv_iterate csvfile scriptfile
```

The `csv_iterate` command parses the comma-separated values in columns in `csvfile`, and for each row of values, executes `scriptfile` with `$col1` set to the first column in that row, `$col2` set to the second column in that row, and so on.

argparse: Passing command-line arguments into scripts

You may need to pass command line arguments into scripts, e.g., so that you can specify what user to log in as, or what web site URL to start at. The `argparse` extension lets you read command-line arguments into twill variables:

```
% twill-sh -- val1 val2 val3
>> extend_with argparse
>> get_args
>> echo "${arg1} is val1"
```

The `--` separator is used to separate script files to be run by twill from command line arguments, which is ignored unless the `argparse` extension is called.

Note that the `argparse` extension is a more flexible alternative to the `--url` command-line argument. When specifying multiple script files on the command line, using `--url` causes twill to load that URL for *each script*. This is in contrast to using `argparse` to read an initial URL off the command line:

```
% twill-sh setup_script testscript cleanup_script -- http://test-site.com/
```

where the scripts are run in a particular order and only the first script needs to know what URL to use.

dirstack: Changing directories

`dirstack` adds convenience functions for changing and restoring the current working directory (cwd). The command:

```
>> extend_with dirstack
Imported extension module 'dirstack'.
(at .../twill/dirstack.pyc)
>> chdir ..
/Users/t/Desktop/ora-books/intro-web-testing
changed directory to ".."
```

will put the current directory on the directory stack and then change to the parent directory. The command:

```
>> popd
popped back to directory "/Users/t/Desktop/ora-books/intro-web-testing"
```

will then return to the original directory.

formfill: Convenience functions for filling out forms

`formfill` contains three very useful functions for filling out forms. As usual, use `extend_with formfill` to load the extension module.

The command:

```
>> extend_with formfill
Imported extension module 'formfill'.
>> fv_match form1 field.* somevalue
```

will fill in *all* fields on the form `form1` that match `field.*` with `somevalue`. In other words, it works like `formvalue` but it doesn't fail when there are multiple matches to the field name—it just sets all those fields to the given value.

The command:

```
>> fv_multi form1 field1=val1 field2=val2 field3=val3
```

will fill in multiple form fields at once. This is identical to issuing:

```
>> fv form1 field1 val1
>> fv form1 field2 val2
>> fv form1 field3 val3
```

The command `fv_multi_sub` takes the same arguments as `fv_multi` but issues a `submit` after setting all of the values.

Using twill from Python

Warning: this section assumes you're minimally familiar with the Python programming language!

All of the commands available to the twill shell are, in actuality, simple Python functions that take string arguments. For example, in the Python shell you could write the following:

```
>>> from twill.commands import *
>>> go('http://webtesting.idyll.org/')
>>> code(200)
```

There's also a higher-level object-oriented interface that you can use:

```
>>> from twill.browser import Browser
>>> b = Browser()
>>> b.go('http://webtesting.idyll.org/')
>>> assert b.get_code() == 200
```

There are advantages and disadvantages to using the two Python interfaces. The main advantage is that with both interfaces you can use native Python code to construct URLs and manipulate HTML. You can also write loops, conditionals, and so on.

For testing purposes, we tend to recommend the basic scripting language, precisely *because* it doesn't have any loops or conditionals—that way you don't have to test your tests! It's also much easier to read and write the twill language than it is Python, at least for people who aren't primarily programmers. (We find that people who are primarily programmers prefer to write Python code directly, of course!)

If you want to mix Python scripts and twill, there's a good alternative to writing everything in Python using the twill API: you can extend twill with Python directly! This is a simple way of writing what is known as a "domain specific language" for your application. Read on...

Extending twill Yourself

Warning: this section assumes you're minimally familiar with the Python programming language!

All of the basic twill commands and the extension modules we discussed above are implemented in Python—generally, very *simple* Python. To write a twill extension function, just write a Python function that takes only strings as arguments: for example:

```
def hello(person):
    print 'Hello,', person
```

is an extension function that can be called from twill like so:

```
>> hello julie
>> hello "ted meister"
```

To write an extension module that you can load with the `extend_with` command, put the commands in a Python module or package and make sure it's in the Python module path.

Writing a simple extension module

The rules for writing an extension module are really pretty simple. Here's the above "hello" extension function put in its entire context.

(Put the following in a file `test_extend.py` and make sure the directory containing this file is in your `PYTHONPATH`. To test this, run 'python' and try importing the module directly before running this example.)

```
"""
Test extension module.

(instructions on using it go here.)
"""

# a list of all functions exported by this module
__all__ = ['hello']

def hello(person):
    """
    >>> hello <person>

    Say hello to <person>.
    """
    print "hello,", person
```

This is just a standard Python module. The module docstring is printed out on import in the shell (although not when running a script) and each function docstring becomes the help for that function. You can now use `extend_with` to import your extension module:

```
>> extend_with test_extend
>> help hello
>> hello <person>
```

...and that's pretty much all there is to it. Extension modules can use any of the commands available to Python as well as any commands in twill's Python API.

Writing domain-specific languages in twill

One particularly important use for extension modules in twill is to write simple *domain specific languages*, or DSLs, to facilitate testing. twill itself is an example of a DSL for web browsing: it contains commands that map intuitively to most of the common actions performed in browsing the Web. However, by extending twill

with functions specific to your testing problem, you can turn twill into a language better suited for your testing.

Let's consider the account creation form that's part of the example web site. The sequence of tasks we used to create a new account and log in was this:

```
>> go /login
==> at http://webtesting.idyll.org/login
>> submit create
Note: submit is using submit button: name="create", value="create account"
>> formvalue 1 name "Ted Meister"
>> formvalue 1 gender male
>> formvalue 1 username ted
>> formvalue 1 password secret
>> submit
```

If you write a simple twill extension function:

```
def add_account(name, gender, username, password):
    from twill.commands import *
    go('/login')
    submit('create')
    fv(1, 'name', name)
    fv(1, 'gender', gender)
    fv(1, 'username', username)
    fv(1, 'password', password)
    submit()
```

you could use this simpler command instead:

```
>> add_account "Ted Meister" male ted secret
```

This is a fairly trivial example, but the benefits in clarity and reuse of code are already significant. You could test web site access restrictions pretty easily by adding a few simple functions. For example, if you built two more functions, `add_role` and `login_as`, you could write scripts that look like this:

```
>> add_account User1 male user user
>> add_account Admin female admin admin
>> add_role admin administrator

>> login_as user user
>> go /restricted
>> code 403

>> login_as admin admin
>> go /restricted
>> code 200
```

This is considerably more concise than going and visiting all of the web pages right in the twill script. Moreover, if the login page or account addition page forms

change, you can more easily change the functions to match than you can a bunch of different twill scripts that all use the regular twill commands. And, finally, it achieves what programs *should* achieve: a one-to-one mapping between the task to be accomplished and the language in which the program is written.

One specific use for domain specific languages in twill is to test web sites that use Ajax. We'll discuss this more in the Advanced book.

More on DSLs

Because twill can be arbitrarily extended with Python modules, you can add features entirely unrelated to the Web. The `dns_check` extension module is one such example.

`dns_check` is an extension for twill that wraps the [dnspython](http://www.dnspython.org/) [<http://www.dnspython.org/>] library, which is a DNS client library for Python written by Bob Halley. `dns_check` lets you make assertions about name service records from within twill; for example:

```
dns_resolves agile.idyll.org 69.55.232.78
```

ensures that host *agile.idyll.org* resolves to the IP address 69.55.232.78.

The `dns_resolves` function itself is just a simple extension function:

```
def dns_resolves(host, ipaddress, server=None):
    if not is_ip_addr(ipaddress):
        ipaddress = _resolve_name(ipaddress, server)

    for answer in _query(host, 1, server):
        if ipaddress == answer.address:
            return True

    raise TwillAssertionError
```

with nothing web related about it.

`dns_check` contains a number of useful functions for systems administrators. However, here we show it only to demonstrate that twill can be extended quite easily to perform essentially any Python-programmable function or test, whether or not it's web-related.

Functional Web Testing with Selenium

A Brief History of Selenium

(by Jason R. Huggins)

Back in 2004, I was in charge of writing a new time tracking system for ThoughtWorks. We chose Plone, a Python-based content management system for the base

of our system. We customized it quite a bit and spent a lot of time getting the time and expense data entry forms to look and act correctly.

Since you're a web developer, you know all too well that "getting forms to look and act correctly" often means editing raw HTML and JavaScript. And if you can't dictate a browser standard by fiat in your organization, then it also means testing your sites in many web browsers—at the very least, you need to test your site in Internet Explorer, Firefox, and Safari.

And that's how Selenium began. It all started with a simple "add 1 row" button on the time entry form. The default timesheet page had five rows for entering time sheet data. However, some consultants worked on many projects in any given week, so they may need to have more than five rows. We needed an option to "add 1 row", and also needed to *test* it.

Implementing this feature sounded simple enough, but in actuality it wasn't. There are two general ways to do this, and they are very different in implementation style. The first one is a server-side technique: an "add 1 row" button submits the current form to the server, the request for adding 1 row is detected, and a new timesheet page is created this time with one extra blank row. The other technique relies only on client-side JavaScript: a JavaScript trigger clones an existing row, blanks out the values, and appends that row to the form.

At the time (early 2004), the conventional wisdom in web development way to avoid JavaScript. From the developer's perspective, it was better to do everything on the server, where you have full control over the code and what gets displayed, and where you have full-featured and high-quality tools to write, test, and document everything you do. JavaScript was "bad" for several reasons: first, JavaScript engines are implemented slightly differently in each browser, so writing in JavaScript follows the old joke about Java—"write once, test everywhere." This isn't a problem if you have automated tests, but there was simply no way to *write* automated tests for web sites that used JavaScript heavily. Moreover, tools for writing and testing web applications that used client-side JavaScript were either too simple or too expensive.

Some developers suggested that we implement the feature on the server (the first style of implementation) and use jWebUnit to test it.

I wasn't happy with that answer.

Even if I wanted to avoid using JavaScript, I didn't like having to dumb down my application to the level of my testing tool. Instead, I wanted to code the feature I wanted, and match the testing tool to the demands of my application.

However, jWebUnit couldn't test JavaScript. Or to be precise, it could only emulate a browser and kinda sorta test JavaScript *logic*. It couldn't verify how JavaScript interacted with a web page in Firefox, IE, or Safari, because it didn't use those browsers to test your code. (Since then jWebUnit has been updated to fix this by embedding Selenium!)

Perhaps I was guilty of a little premature optimization at the time, but I believed that doing an "add 1 row" operation in JavaScript would involve far less code to implement and would also scale better. One reason we were working on the new system was that the previous system was really slow, so I figured it was part of my mandate to fix this. After all, if JavaScript could add a row for me, the server wouldn't get hit as often; fewer unnecessary hits to the server means a faster, more scalable, more responsive app. (This theory held up well in practice, too.)

But even if the theory was good, I still had to convince the developers that every action on a web page shouldn't trigger a page refresh every time. At first, I went with their original suggestion. We implemented "add 1 row button" on the server and tested it with jWebUnit. Well, as I'd guessed, the time it took it process a long form and give back a copy of that form, plus one row, didn't fly too well with users, and it was especially unpleasant for those connecting over phone lines, which some users still do today.

We left the feature implemented on the server for the moment, since it was working, and planned to come back and optimize for speed later. Then feature #2 came: pop-up list for project codes. ThoughtWorks has many client codes, each with many project codes. The list of codes to be presented to the user was quite long. Too long, in fact, to be usable in a drop-down menu, which would have been the first choice. Anne Brent, ThoughtWorks' GUI and graphic designer at the time, suggested that we implement the "project picker" as a pop-up window. This, of course, needed JavaScript. And again, this was something that jWebUnit couldn't test because we needed to test the feature as it worked in a real browser. This time, I couldn't see how to "dumb down" the application to make this all work on the server.

So, I decided to reverse course: I accepted that we would be using JavaScript, and decided to upgrade our testing tools.

Now, why did I pick on poor jWebUnit? jWebUnit represents one class of web testing tools—protocol drivers. You may already be familiar with a protocol driver: that's what twill is. Protocol drivers mimic the client-side aspect of browser-server interaction by speaking HTTP directly to the server, but in general they do not

understand JavaScript. Even those that do cannot speak the different dialects used by the different browsers.

When I searched the web for "web testing tools" almost all the tools I found were protocol drivers, and none could test JavaScript in the browser. We needed an "application driver" instead. Moreover, we needed an application driver that worked with multiple browsers, and all of the existing projects were tightly coupled to IE. Tools from Segue and Mercury could test both Mozilla and IE, but were (and are!) *very* expensive, hence out of range for this internal project.

Once I decided to use JavaScript, we were left exposed without any working functional tests as we tried to build or buy a tool that could test the JavaScript in our application. And during that time without an automated functional testing tool, I discovered why JavaScript got its notorious reputation. It was easy to implement the "add 1 row" button, at first. When a developer thought it was looking good, she ran her unit tests, checked the site (manually) in her favorite browser, and committed the code to the repository. However, as we soon found out, testing in one browser was not enough. In fact, testing in any single version of two browsers was not enough. Each time the Mozilla foundation released a new version of Mozilla or Firefox, there would be subtle changes that broke the add row button and/or the project pop-up window. And a fix for a new version of Mozilla would break the feature in IE. Repeat the flip-flopping of which browsers got tested and which had broken features, and I could see how a developer could only retain her sanity by avoiding JavaScript entirely.

However, despite the testing problems, I believed the usability gains outweighed the pain of testing.

Now, there were a few JavaScript *unit test* libraries available, but what I needed was a functional/UI testing tool. A unit testing library would be good for testing logic, business rules, or the correctness of individual code modules and functions, but we needed a testing tool to test our web application from the user's perspective. We also needed a tool that could drive the different browsers—IE, Mozilla, and Safari—through the application.

So some requirements emerged for our functional testing tool:

- Tests login to logout, from the user's perspective
- Tests pages that use JavaScript, including those that make changes to the document within the browser window.
- Tests pages in the same browsers that users are using, in particular, Internet Explorer and Mozilla/Firefox. (Safari was a later addition.)

Andrew McCormick produced a working prototype that met the first two requirements, but since it was a Mozilla-only extension, it did not meet the third requirement of multi-browser support. We were getting closer, though! Also around this time, we discovered the FIT framework, written by Ward Cunningham.¹ Even with FIT, however, there was no ability to test an application that uses JavaScript. There was some code to functionally test a web app, but FIT used jWebUnit to execute the tests!

Impressed with the FIT framework's overall design and style, especially its emphasis on a people friendly test syntax, I started a quest to implement a FIT-like tool for web browsers using JavaScript.

At first, I tried to stick to the FIT specification, but it was inadequate for testing UIs. So Paul Gross, Tina Wang, and I started tweaking the API. We also wanted to *see* the tests running in the browser, rather than simply get a FIT-style summary of the tests. We also decided to add a step debugger mode.

With that last requirement (in-place step-running of tests), we knew we had an altogether different kind of tool in our hands. It was inexpensive (free!); it could test web applications from within IE 6, Mozilla 1.0+, and Mozilla Firefox .8+; it could test our site from login to logout; it could include all pages that depended on JavaScript; and because it ran on-screen, you could see all the action as each command was executed.

After we had used it for a few weeks, I started showing the tool (at the time called JavaScript Functional Tester [JSFT]) to other developers. They immediately wanted to use it on *their* projects, because then they, too, could extend their web applications to use JavaScript without losing the ability to test them! Within 2 months, Paul Hammant and Bret Pettichord completed the push to open source the software. Bret is the also the co-author of the Watir testing tool. Paul is also the creator of Selenium Driven, the forerunner to Selenium Remote Control, written by Patrick Lightbody. Remote Control (and Driven) allows a developer to "drive" Selenium tests from xUnit-style libraries in C#, Java, Perl, PHP, Python, or Ruby. (How to use Selenium Remote Control is not covered in this Short Cut.)

JSFT wasn't the most exciting name, so we took aim (mostly in jest) at Mercury Interactive's hegemony in the testing tool marketplace. A few health sites suggested that the natural cure for mercury poisoning is Selenium, so we ended up choosing "Selenium" as the name.

¹Ward Cunningham may be better known as the father of the wiki.

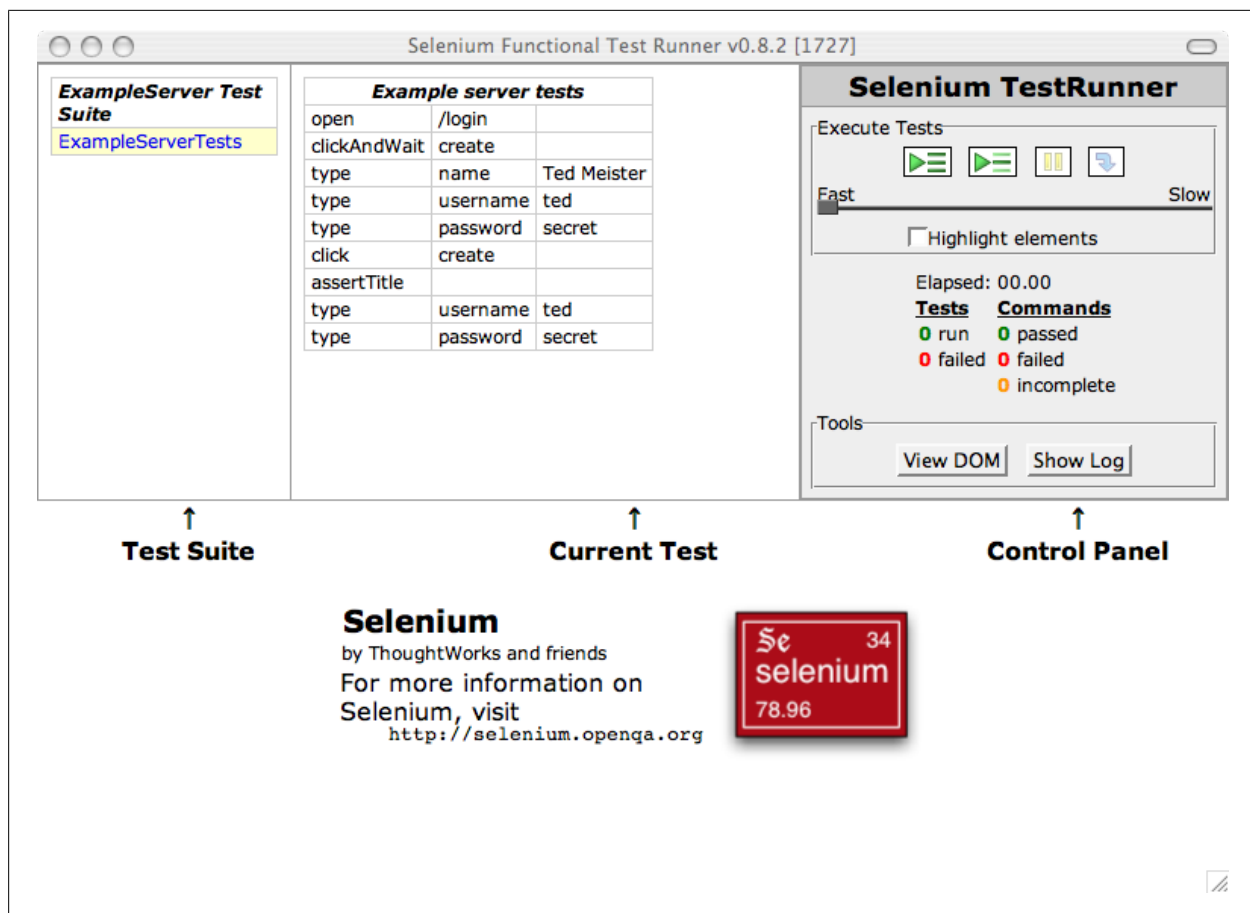


Figure 1.

This was back in 2004. Now, just three years later in 2007, things have changed dramatically. Ajax, Web 2.0, and Rails mean that JavaScript is heavily used in many new web sites. And, as a result, Selenium is enjoying quite a bit of attention—it's a tool in the right place at the right time. We hope you agree!

Introducing Selenium Core

Selenium Core is a tool that uses JavaScript to drive your browser through a program. Rather than describing it in detail, it's simpler to just show you how it works. We have included a very simple Selenium test page for the example server discussed in the twill section, above. Go to <http://webtesting.idyll.org/selenium> and select the "Introductory Selenium test suite."

You should now see a window with four panes: the ExampleServer test suite (on the upper left), a pane in the top center containing a table with the title "Example server tests," a pane in the top right with the title "Selenium TestRunner," and a pane across the bottom half of the screen with a Selenium logo in it (**Figure 1**).

Now, run all of the tests by selecting the left-most green "Play" button in the "Selenium TestRunner" window. This will execute the test suite, which in turn will

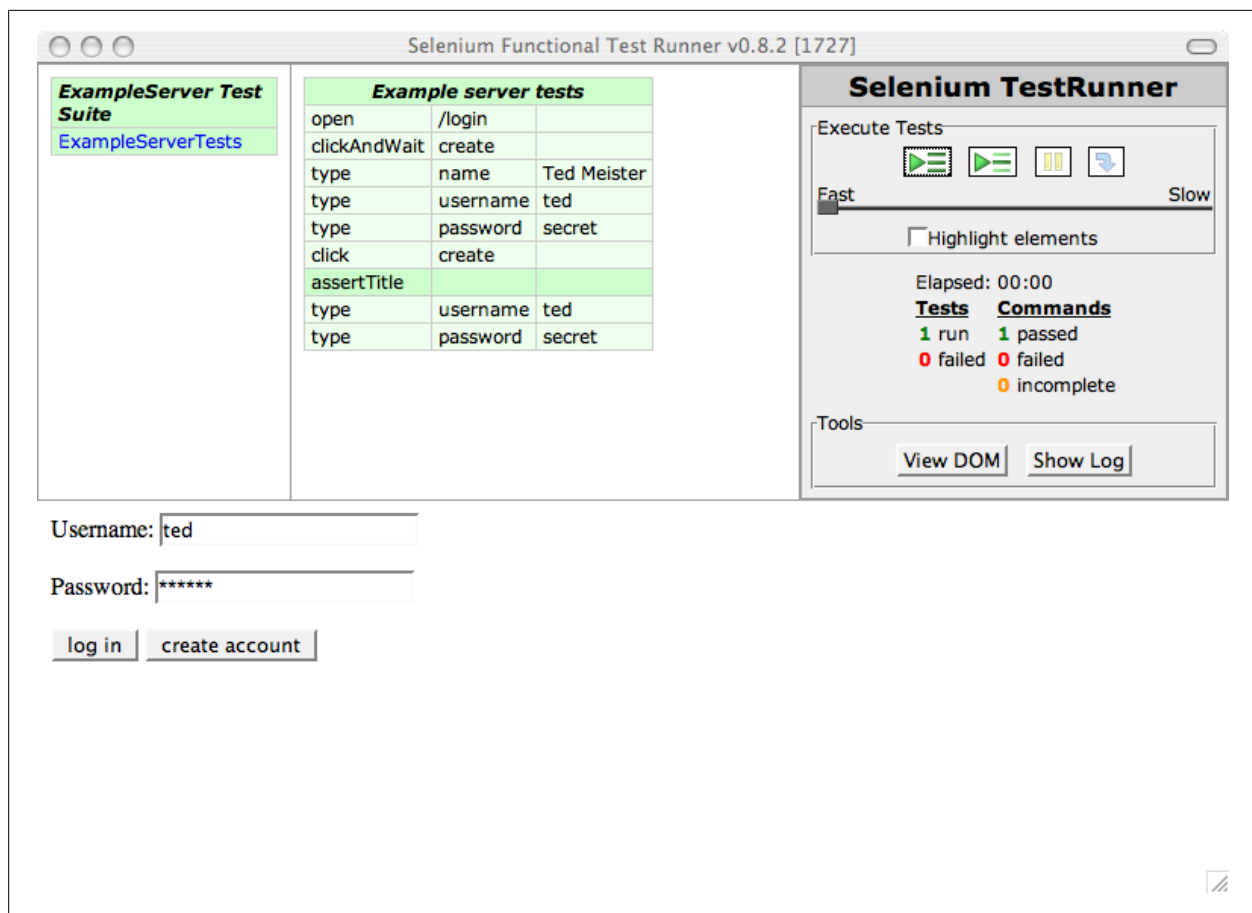


Figure 2.

drive your browser window (the bottom pane of the window) through the site according to the Selenium program on the test page. If the test succeeds, everything should be green in both the test suite window (upper left) and the test window (upper center). (It is all green, right?) You can also see a test summary in the Test Runner window, which will give the elapsed time, the number of test pages run (and the number that failed), and the number of commands that passed or failed or didn't complete (**Figure 2**).

To run the test again, but more slowly, change the speed slider selection bar in the Test Runner window from "Fast" to something intermediate, and then hit the play button again. This will walk through the commands more slowly.

If you have multiple browsers installed, go ahead and run the tests in each one. It should work in Internet Explorer, Firefox, Safari, Opera, and most other browsers.

This simple demonstration shows why Selenium is so powerful. Not only does it let you automate the browser in a way that's indistinguishable to the server from manual browsing, but you can test both the site *and* the browser's interaction with

the site, within multiple browsers. Moreover, as you'll see below, you can test JavaScript and Ajax client-side code as well.

Building a Selenium Test Suite

In this section, we will describe the mechanics of deploying Selenium tests, and give an overview of how to build Selenium tests with the Selenium IDE and the XPath Checker extensions to Firefox. We also describe some of the most frequently used Selenium commands. At the end we discuss how to test JavaScript and Ajax sites with Selenium.

Deploying and running tests

Selenium is, at heart, a set of JavaScript code that interprets tests written in straight HTML. To install Selenium, all you need to do is unpack the Selenium Core zip archive and move the `selenium/` subdirectory into a location that your web server or web application will serve as static pages. You can get the latest version of Selenium Core at <http://www.openqa.org/selenium-core/> (at the time of this writing, the latest version of Selenium Core is 0.8.2).

The next thing you need to do is write a page for each test suite. Each test suite contains multiple Selenium test pages; to write the test suite, just write an HTML table with a single column, and with each row containing the name of a test page enclosed in an `<a href=...>` tag. To get started, just grab the source code for the first test suite above, at <http://webtesting.idyll.org/selenium/suite.html>.

The suite source is pretty simple:

```
<table cellpadding="1" cellspacing="1" border="1">
  <tbody>
    <tr><td><b>ExampleServer Test Suite</b></td></tr>
    <tr><td><a href="ExampleServerTests.html">ExampleServerTests</a></td></tr>
  </tbody>
</table>
```

It's a table that contains a column full of tests; any link in this column is considered a test page. Each test page consists of a Selenium Core "program," which is just an HTML table with the first column containing each Selenium command in the program. Arguments to the commands are provided in additional columns, so the command:

command | argument 1 [| optional argument 2]

would be written in HTML like this:

```
<tr>
  <td>
    command
  </td>
```

```

<td>
    argument 1
</td>
<td>
    argument 2
</td>
</tr>

```

For an actual Selenium program, you can look at the source HTML for the example server tests at <http://webtesting.idyll.org/selenium/ExampleServerTests.html>.

This URL contains the example code you ran above. For example, the first two commands are `open | /login` and `clickAndWait | create`, which go to the URL `/login` and click on the `create` button. The HTML source for these two commands is:

```

<tr>
    <td>open</td>
    <td>/login</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>create</td>
    <td></td>
</tr>

```

So, the test suite consists of links to individual test pages, and each test page contains a program written in HTML, but what actually runs the test suite? The answer is the Selenium Test Runner, a static HTML and JavaScript page that's in charge of loading the test suite and executing the tests; to load a particular test suite, you pass it into the Test Runner page as part of the URL, `selenium/core/TestRunner.html?test=` ***<path to test suite HTML page>***

The path to the test suite HTML page can be either absolute or relative; if it's relative, it will be interpreted as a path relative to the `TestRunner.html`. In the case of the example server, the absolute path to the test suite is `/selenium/suite.html` and the relative path from `TestRunner.html` is `../suite.html`, so the full URL for the tests is: <http://webtesting.idyll.org/selenium/core/TestRunner.html?test=../suite.html>.

Note that in the [Testing a Simple Web Application with twill and Selenium](#) section of this shortcut, we go through the mechanics of deploying a Selenium test suite under a Django application.

Recording tests with the Selenium IDE

Writing GUI tests for web apps is no fun, no matter what your testing tool is. You need to navigate through pages, fill and submit forms, and verify that certain elements are present on the pages. Doing all this manually can quickly become tedious.

If you're writing Selenium tests, these activities are made tolerable due to the Selenium IDE. The Selenium IDE is a Firefox extension that you launch via the Tools menu. It records web browsing actions such as opening a URL, clicking on a link, entering text in a field form, selecting a drop-down menu item, submitting a form, etc. This makes building a basic test suite a simple matter of starting up the Selenium IDE and running through the actions you want to test.

For example, to build the example test suite demonstrated above, do the following:

1. Download and install the Selenium IDE from <http://www.openqa.org/selenium-ide/>. (You will need to have Firefox installed, of course.)
2. Go to <http://webtesting.idyll.org/>. Go to the Tools menu, and select "Selenium IDE". A Selenium IDE window will pop up, ready to record.
3. Click on the "log in" link on the webtesting.idyll.org main page.
4. Click on the "create account" button.
5. Enter a name, gender, username, and password, and click on "create". If you look at the Selenium IDE window, you should see it faithfully recording all of your actions (**Figure 3**).
6. At the login page, enter the username/password pair for the account you just created, and push "log in" (**Figure 4**).

And there you are—a full test of creating an account and logging in! The Selenium IDE window will have the complete test program recorded, and you can either run it (by pushing the green "Play" button) or you can save it (by going to the File menu and saying "Save test"). This will save an HTML test program that can be directly linked to from a test suite (**Figure 5**).

A guide to Selenium commands

Below we discuss a few of the most commonly used Selenium commands. The full list is available on the **Selenium TestRunner reference page** [<http://www.openqa.org/selenium-core/testrunner.html>].

Opening web pages by URL

Web pages can be opened by their URL via the *open* command:

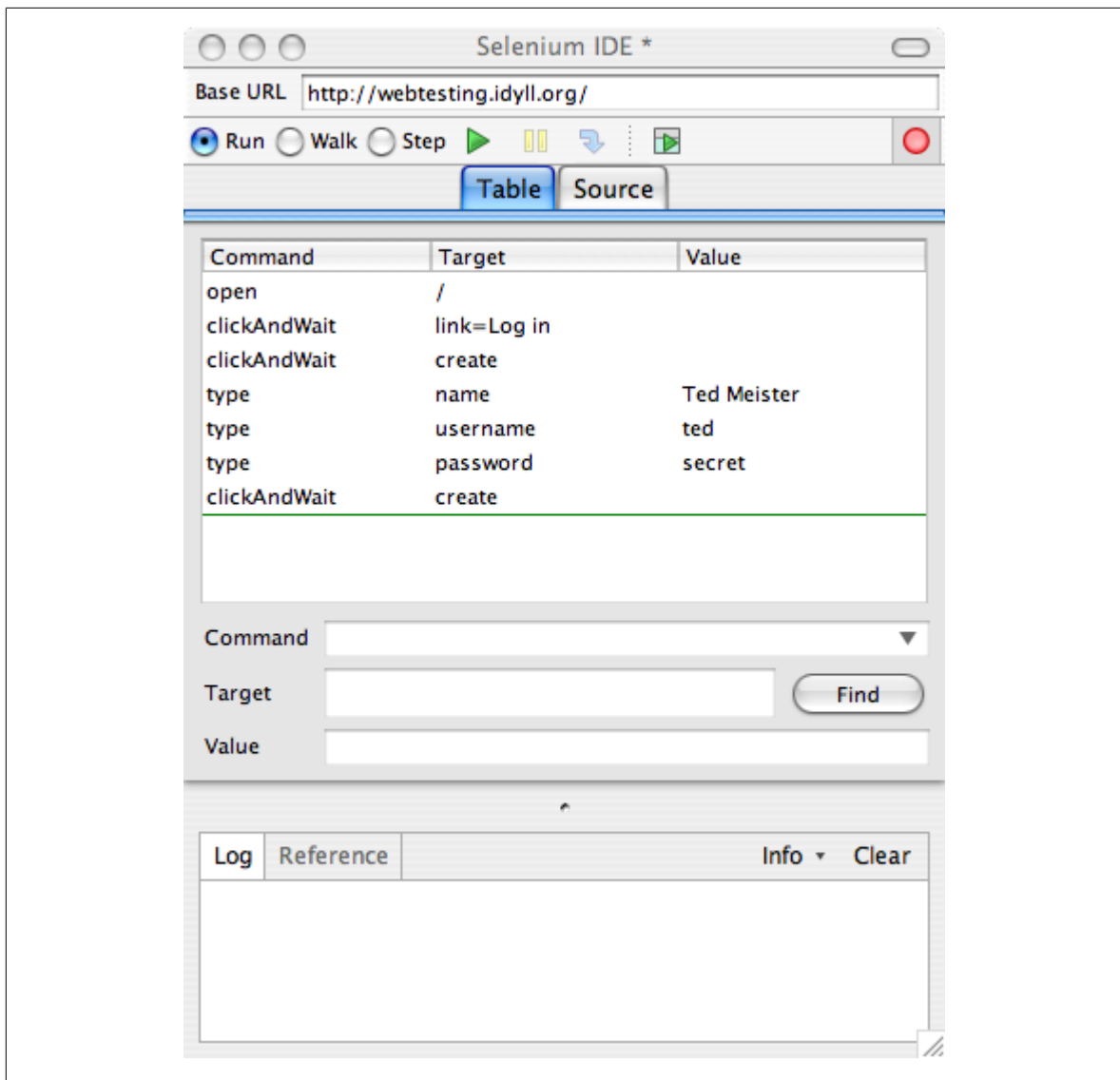


Figure 3.

open | url

Clicking on links

This is one of the trickiest aspects of writing tests in Selenium. In general, HTML elements on the web pages you're trying to test can be referred to in Selenium commands via "element locators", which can be one of the following:

- identifiers: the `id` or `name` attribute of the element
- DOM traversal syntax: `document.forms['myForm'].myDropdown`
- XPath syntax: `//img[@alt='The image alt text']`

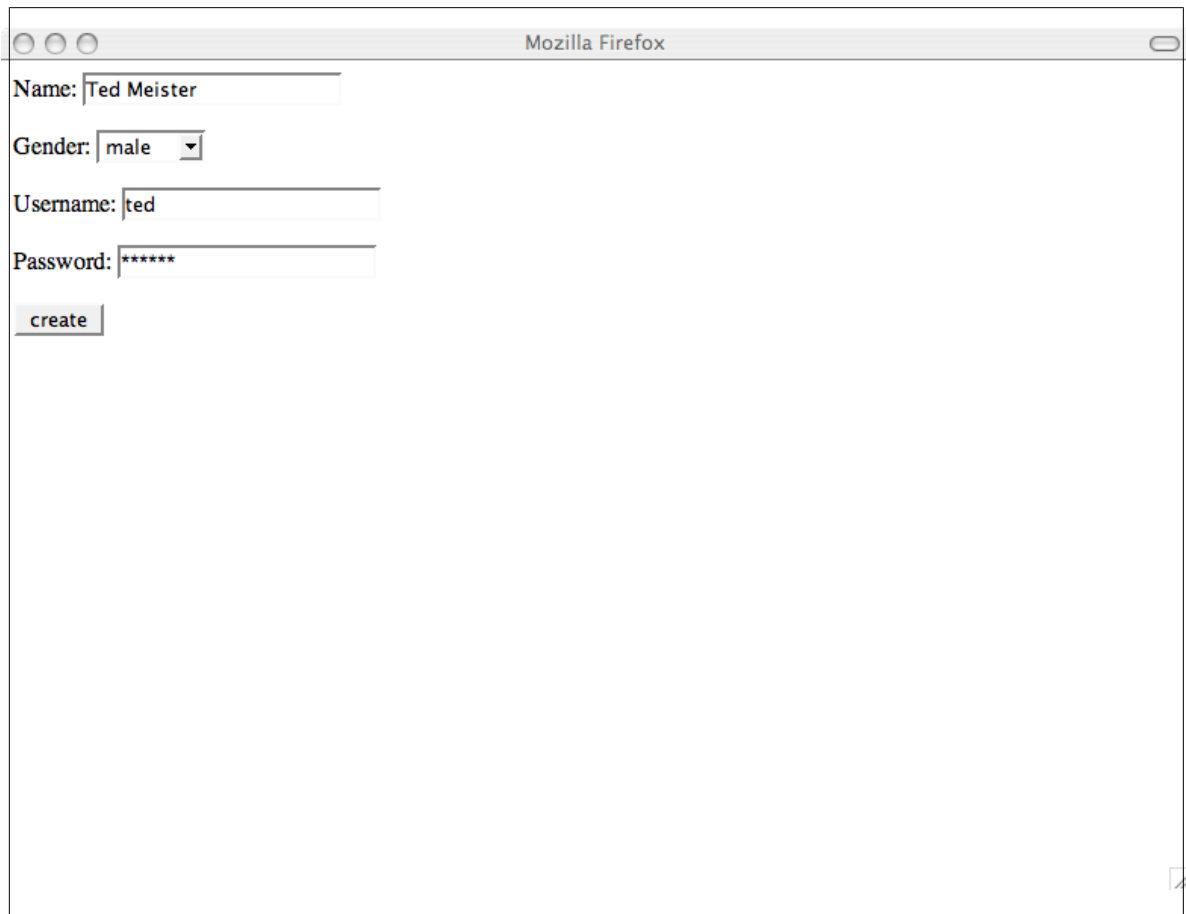


Figure 4.

Life is easy when HTML elements such as links have id attributes such as `id="the_link_id"`. In this case, the command you need to use for clicking on the link is simply:

```
click | the_link_id
```

Life is also pretty easy when links have text that is on the same line with the starting `a` tag and the closing `/a` tag. In this case, you can use the following XPath syntax:

```
click | //a[text() = "the link text"]
```

A recently introduced Selenium command for accessing links by their text is the *link:* command:

```
click | link:the link text
```

Note that you need to follow link by a colon, then immediately by the text of the link with no quotes.

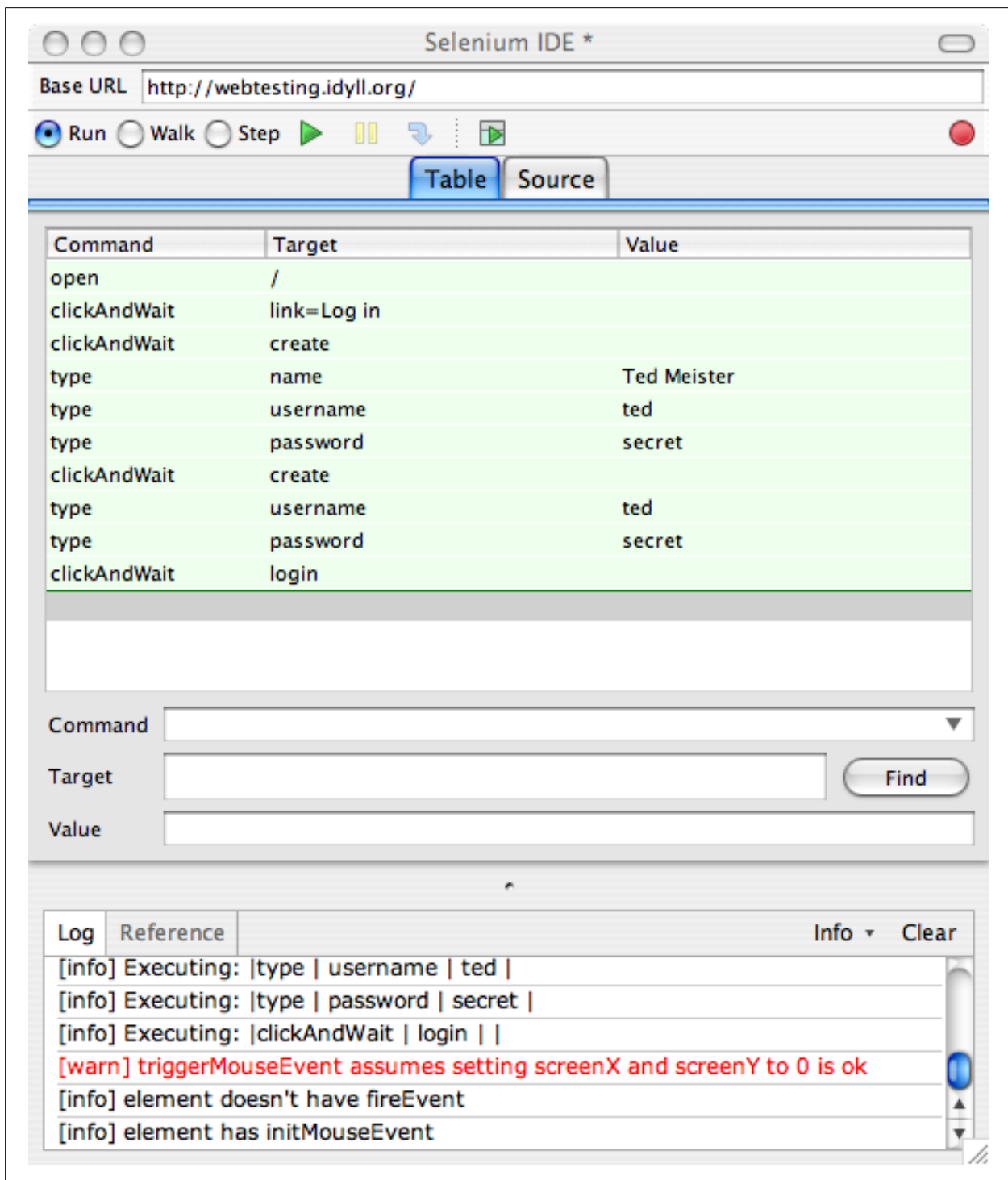


Figure 5.

In other cases, especially when the link text is on a line by itself or spans multiple lines, the text method will not work and you will need to identify the link by some other attributes.

Clicking on submit buttons

The command you need to use for clicking on form submit buttons is *click*. The target of the command is the element locator for the button. This can be either the button's *name* attribute or its *value* attribute.

```
click | form.button.Register
```

Here is an example of clicking a submit button by its value, using an XPath expression:

```
click | //input[@value='Log in']
```

Clicking on checkboxes

The *click* command is also used for clicking on checkboxes. You need to indicate the name of the checkbox field as the target of the command. The click command, when applied to a checkbox, toggles the value of that box.

Note that the *click* command can also be used for clicking on radio buttons.

Entering text in input fields

If you need to fill in input fields in forms, use the *type* command, which takes two arguments: the name of the input field (you'll need to figure what that name is by inspecting the HTML source) and the value you need to type in that field.

Selecting values in drop-down lists

This is accomplished via the *select* command, which also takes two arguments: the name of the drop-down list field and the value you need to select in that list.

Verifying the state of the application

So far we have shown how Selenium can drive an embedded browser via "Selenese" commands. This is only one aspect of testing a web application, since it indirectly verifies that the HTML elements it expects to click or open are indeed present. For direct verifications, Selenium provides a variety of "verify" commands that check the values of the different elements under test. Perhaps the simplest check is *verifyTextPresent*, which makes sure that a given snippet of text is present in a web page.

Values for elements of a form (input fields, drop-down lists, checkboxes) can be verified via the *verifyValue* command:

```
verifyValue | wysiwyg_editor | Epoz  
verifyValue | checkbox | off
```

(For a checkbox such as "listed," the value that we compare with is either *off* or *on*.)

To verify that a web page contains a specific URL, use either *verifyAbsoluteLocation* (which checks that the URL of the page equals the given string) or *verifyLocation* (which checks that the URL of the page ends with the given string).

There are many other types of check commands available on the **Selenium TestRunner reference page** [<http://www.openqa.org/selenium-core/testrunner.html>].

Using variables

One very convenient feature of Selenium is the ability to use variables directly in the test tables. In the official documentation, the only way to deal with variables is via separate HTML pages where these variables are set. In the TestNewUser table, I'm using the `setVariable` command to set three variables: `base_url`, `join_url`, and `logout_url`. The syntax for setting a variable is:

```
setVariable | <variable name> | <variable value>
```

If the value is a string, it needs to be enclosed in quotes.

The syntax for getting the value of a variable is `${var_name}`.

An important use of a variable, especially when testing web sites that provide log in functionality, is to set a random user ID that will be used in the test. For example,

```
setVariable | random_user | 'user' + (new Date()).getTime()'
```

In this case the value for the `random_user` variable is obtained by concatenating a string ('user') with the value returned by a JavaScript function (the `getTime()` method for a `Date` object). So, you can just use JavaScript code in your variable assignments.

Locating page elements with XPath Checker

Several of the commands in Selenium can take XPath locators as arguments; for example, in `click | //a[text() = "the link text"]` the argument is an XPath locator specifying the link to click on. Locating specific bits of text on the page manually can be quite annoying. The **XPath Checker** [http://slesinsky.org/brian/code/xpath_checker.html] is a sanity-saving Firefox extension to help with this chore.³ After installing it, you can highlight any text on a web page and get an XPath expression that uniquely identifies this text. To do this, right-click on the page and choose "View XPath." A Firefox window will pop up with an entry field containing an XPath expression, as identified by the tool, and with the text matching that expression. This window allows you to interactively change the

³You can also try out **XPather** [<https://addons.mozilla.org/firefox/1192/>], another Firefox extension that some people prefer to the XPath Checker extension.

XPath expression and see the text that it matches, which is great for learning XPath. (There's also a good **XPath tutorial** [<http://www.zvon.org/xxl/XPathTutorial/General/examples.html>] available for people unfamiliar with XPath.)

Note that XPath expressions don't work that well in Internet Explorer. If browser compatibility is high on your list, then you will need to identify HTML elements via their DOM locations. For that, you can use the DOM Locator extension which ships with Firefox, or you can also try to build an equivalent DOM expression based on the XPath expression generated by the XPath Checker.

Selenium and JavaScript

Selenium deals with client-side JavaScript very well: it simply handles it, with no special requirements.

To see how it actually works, go to the front page of the example server. Here we have two common examples of client-side JavaScript: first, an alert box, triggered by clicking on the "JavaScript alert box" link; and second, form entry assertions checked using the `onChange` element of input boxes. Try clicking the alert link as well as entering some numbers above 200 into the form boxes to see how they work.

The Selenium tests for these features are located in `suite2.html`; you can run them by going to this page, <http://webtesting.idyll.org/selenium/core/TestRunner.html?test=../suite2.html> in the example server. Try running them all. (They should all work!)

The only thing new about these tests is that they make use of two new Selenium commands, `assertAlert` and `assertValue`. These commands test for the contents of the alert box and the form value, respectively. In particular, there is nothing special that needs to be done to test the `onChange` based JavaScript—simply enter the values, and see what becomes of them.

So, that's it for testing client-side JavaScript! As we mentioned above, Selenium is particularly nice because you can run your tests in any browser—and thereby test your web site, including your JavaScript code, in any browser you have available.

Asynchronous HTTP requests and Ajax

It is also possible to test web sites that use asynchronous HTTP requests with Selenium. Asynchronous HTTP requests are a common component of Ajax functionality, and they enable a very interactive web experience; because of this, Ajax has become very popular.

The problem with Ajax testing is that the HTML page under test is modified asynchronously, so a plain Selenium `assert` or `verify` command might very well fail

because the element being tested has not yet been created by the Ajax call. A quick and dirty solution is to put a `pause` command before the assert. This is error-prone, since the pause might be not sufficient for a slow server, while being unnecessarily slow on a faster one.

A better solution is to use Dan Fabulich's **waitForCondition** [<http://wiki.openqa.org/display/SEL/waitForCondition>] extension, which allows you to wait until a specified JavaScript expression evaluates "True." Selenium will not advance until the condition becomes true, and the test will fail if the condition is not met within a specified timeout.

Here is an example of a Selenium test table row that uses `waitForCondition`:

```
waitForCondition | var value = selenium.getText("//textarea[@name='comment']"); \
value == "" | 10000
```

This command waits for 10 seconds (the second argument to `waitForCondition`) until the JavaScript command passed in as the *first* argument succeeds. Here, the command we passed in checks for the presence of an (empty) textarea named `comment`, which is added to the page dynamically through an Ajax call.

Note that to locate the element, we used the special variable `selenium`, which makes the Selenium API directly available to JavaScript. All of the methods available in the file `selenium-api.js` (which comes with Selenium Core) are available via this variable. In this case, we used `getText`, which takes a locator as its argument and returns the contents of the located element.

Extensions like `waitForCondition` allow you to add new commands to Selenium Core, and there are a number of **user-contributed extensions** [<http://wiki.openqa.org/pages/viewpage.action?pageId=282>] available on the Selenium web site. Installing extensions is very easy, just add them to the `core/scripts/user-extensions.js` file, which is automatically loaded by Selenium.⁴

Testing a Simple Web Application with twill and Selenium

The Django "poll" Application

In order to provide you with an example of how to use twill and Selenium to test an application, we're going to take advantage of the Django "poll" application that you build in the Django tutorial, at <http://www.djangoproject.com/documentation/tutorial1/>.

⁴We will show you how to write your own extensions in our book on Advanced Web Testing, the second book in this series.

You don't actually need to go through the Django tutorial to read this section, you can download the source code [here](#) [[django-poll-site.zip](#)]. You don't need to know anything at all about Django, either; all you need to do is *install* Django, install the application, and then follow along below.

Before we get started, honesty compels us to admit that Django has a pretty nice built-in testing framework, `django.test`. However, that framework is specific to Django—the testing techniques that we demonstrate below apply to *any* framework in *any* language, not just Django apps or web sites written in Python.

Installing and running the poll application

1. Install Django (see <http://www.djangoproject.com/documentation/install/>).
2. Download the demonstration app, at <http://webtesting.idyll.org/resources/django-poll.zip>.
3. Unpack `django-poll.zip` (it will unpack into a subdirectory named `django-poll`) and change into the `mysite` sub-directory.
4. Run `python manage.py runserver`. It will start a web server on `http://localhost:8000`; to make sure that it is running, visit `http://localhost:8000/polls`.

You're done!

The First Step: Charting a Path to Test

After you have the Django application up and running, visit <http://localhost:8000/polls/>.

You will see a single question, "Is India a continent?" This is a link; select it. Now you will see a set of possible answers: "yes," "no," or "maybe." There is also a "Vote!" button. Select one of the answers and push "Vote!" The final page you will see is one that contains a "1 vote" next to the answer you selected; if you go through and do this a few times, you will see a running tally of the number of selections you've made.

This procedure (select question, select answer, observe tally) is a functional *path* through the program. In this case it's one of the two main functional paths in this application (the other involves the admin interface; more about that later). And this is one of the main priorities for testing: the path that you expect the user to take through your web app.

Again, this path is:

1. Select question
2. Select answer
3. Observe tally

There's really only one main question to test here: do answers get tallied appropriately? But other issues include: What happens if you go to a question that doesn't exist? Does submitting a nonsense answer ID cause the application to break inappropriately? Are there corner cases that only malicious users can access (by editing the URL, for example) and hence there are more *security* issues than functionality issues?

The Second Step: Building a Test Script

Let's start by testing exactly the process we described above. Put the following script in a file (`test-vote.twill`):

```
go /polls
showlinks
```

Now run it (pointing it at the right top-level URL) and look at the output:

```
% twill-sh -u http://localhost:8000/ test-vote.twill
```

```
...
Links:
```

```
0. Is India a continent? ==> 1
...
```

OK, the link we want to follow is "Is India a continent?" The 'follow' command will pattern match using regular expressions, so you can just add

```
follow India
```

to the script. (Also add

```
showforms
```

because you know that the next thing coming up is a form.) Run the script again; now you should see

```
% twill-sh -u http://localhost:8000/ test-vote.twill
```

```
...
Form #1
## ##  _Name_____  _Type_  _ID_____  _Value_____
1   choice_____  radio   choice1   [] of ['1', '2', '3']
2  1  None         submit   (None)    Vote
...
```

This is the form on the poll page. For concision, twill only displays the numerical values associated with the polls, but you can also select the values by name. To do

so, you need to know the name or ID of the form element, though; these are 'choice' and 'choice1', respectively.

So, let's select one of the values, and then 'submit' the form. Add the following to your `test-vote.twill` script:

```
formvalue 1 choice yes
submit
```

```
show
```

This will select a form value, submit the form, and show the resulting HTML. Try running it.

```
% twill-sh -u http://localhost:8000/ test-vote.twill
...
Note: submit is using submit button: name="None", value="Vote"
<h1>Is India a continent?</h1>

<ul>
  <li>yes -- 2 votes</li>
  <li>no -- 1 vote</li>
  <li>maybe -- 0 votes</li>
</ul>
...
```

(Your actual vote count will vary depending on what you've pressed in the past, of course!)

Try running the script a few times and verify by eye that the "yes" count increases by one each time.

Congratulations! You've got your first semi-automated test! Unfortunately, you have also run smack into a big problem. Read on...

The Third Step: Controlling the Application Environment

Now that you have a test that runs through the web app and clicks the appropriate buttons, you have another problem: how do you verify that the votes are getting counted properly? Because the application is starting out in a potentially arbitrary state, i.e., where the number of votes for each choice reflects the history of the application, you cannot simply assume that "yes" has 1 vote and "no" and "maybe" have 0 votes.

How should you deal with this uncontrolled application state? One way is to write a script that keeps track of the number of votes. You can have a program that:

1. Views the "votes" total
2. Extracts the number of votes for each choice and records them

3. Runs through the script (and clicks on a specific choice)
4. Verifies that the number of votes for the correct choice has been incremented

Another way to deal with this is to actually *control* the application state by introducing a **"test fixture"** [http://en.wikipedia.org/wiki/Test_fixture]. Test fixtures are environments containing known data that allow you to make certain assumptions. In this case, one nice test fixture would be one that starts out with zero votes. For example, consider a test program that:

1. Zeroes the counts for each option
2. Runs through the script above (and clicks on a specific choice)
3. Asserts that the number of votes for the correct choice is 1

For functional testing, we highly recommend going with the second option. Less code is usually involved: in the first option, you have to write code that parses out the vote total from the HTML, whereas in the second option you just have to zero out the database. It's also a more predictable option: if you zero out the votes in the database, the votes are *guaranteed* to be zero, and you don't need to worry about e.g. number overflow. And, conveniently, the test fixtures can be reused for multiple tests: once you have the code written to zero out the options, you can simply reuse that code to make sure that the web app starts out in the same condition for every test.

Let's try out this second option. There's a program `reset-counts` in the `mysite/` directory of the Django poll app; try running it.

```
% reset-counts
% twill-sh -u http://localhost:8000/ test-vote.twill
...
<h1>Is India a continent?</h1>

<ul>
  <li>yes -- 1 votes</li>
  <li>no -- 0 vote</li>
  <li>maybe -- 0 votes</li>
</ul>
...
```

Yep, the votes were properly zeroed!

To make this a fully automated test, all we need to do is add a few assertions to the end of the twill script:

```
go /polls
showlinks

follow India
```

```
showforms

formvalue 1 choice yes
submit

show
find "yes -- 1 vote"
find "no -- 0 votes"
find "maybe -- 0 votes"
```

And there we are! A fully automated test! Try running it, both with and without running `reset-counts` first; it should fail unless you run `reset-counts`, of course.

A few notes:

- You don't need to worry about fixtures if you're testing static aspects of a web site, like links on the main page. Most testers will want to test more than this, but `twill` can easily be used to just ping a web site, or check that all the links on a certain page still work.
- Nothing above relies on you, the tester, writing Python code. In fact, as long as the necessary fixtures are made available to the tester, the tester can simply write `twill` scripts.
(It can be convenient to call `twill` from Python, however. In the next section, we'll show how to integrate `twill` with `unittest`.)
- Nothing above requires that you be testing a Django application, much less a Python application.
(There are some nice tricks you can use if you *are* testing a Python application, but you can use `twill` with any web site.)
- The application we're testing *doesn't use JavaScript*. We'll deal with JavaScript later on, when we add Selenium tests.

Integrating Fixtures and Tests with `unittest`

One problem with our automated test is that the fixture setup is separate from the test itself. You, the user, must remember to run the `reset-counts` script manually each time. This isn't terribly inconvenient when the test suite contains exactly one test, but it is a potential source of errors. With multiple tests it can be a much bigger problem, especially if each of the tests requires a different fixture.

This is where unit testing frameworks provide some value, by providing a framework within which to establish fixtures and tests. Below, we show you how to use the `unittest` [<http://docs.python.org/lib/module-unittest.html>] module that comes with Python to fully automate our simple `twill` test.

First, let's outline an empty unit test:

```
class TestPollSite(unittest.TestCase):
    def setUp(self):
        pass

    def test_vote(self):
        pass
```

In `unittest`, the `setUp` function is used to establish the fixture, and it will be run before *each* function whose name starts with `test`. Here, the `test_vote` function is the only test function, so the order of execution will be:

```
test_case = TestPollSite()
test_case.setUp()
test_case.test_vote()
```

(This order of calls is established by the `unittest.main()` function, which is what executes the test suite.)

Now, what do we want to put into each of these functions?

`test_vote` is easy; we can just take advantage of the `twill.execute_file` function to run the `twill` script that we already wrote.

```
def test_vote(self):
    twill.execute_file('test-vote.twill', initial_url='http://localhost:8000/')
```

The `setUp` function is a bit more complicated, because we have to reset the database counts. This means putting the `reset-counts` code into a Python function:

```
from pysqlite2 import dbapi2 as sqlite
def _reset_counts():
    conn = sqlite.connect("mysite/db.sqlite")
    cursor = conn.cursor()

    cursor.execute("update polls_choice set votes=0")
    conn.commit()
```

Then `setUp` just calls this function, and waits for the Django server to reset:

```
def setUp(self):
    _reset_counts()
    time.sleep(2)
```

All in all, with comments and the proper imports, the `unittest` script looks like this:

```
import unittest
import twill
import time
from pysqlite2 import dbapi2 as sqlite

twill.set_output(open('twill.out', 'w'))

def _reset_counts():
```

```

"""
Reset the counts for all choices in the polling database.
"""
conn = sqlite.connect("mysite/db.sqlite")
cursor = conn.cursor()

cursor.execute("update polls_choice set votes=0")
conn.commit()

class TestPollSite(unittest.TestCase):
    def setUp(self):
        """
        Before each test function, reset the database.
        """
        _reset_counts()
        time.sleep(2)                # Wait for the server to reload

    def test_vote(self):
        """
        Run the twill script 'test-vote.twill'.
        """
        twill.execute_file('test-vote.twill',
                           initial_url='http://localhost:8000/')

unittest.main()

```

Try running it! (It's in `poll-tests.py`, so you can just run `python poll-tests.py`.)

One nice result of using this kind of framework is that *adding* tests that depend on existing fixtures is pretty easy. For example, if you duplicate the `test_vote` function by copying it and naming it `test_vote2`, it will work as expected: this is because the `setUp` function is run before *each* `test*` function.

Note that something this code does *not* do is run the Django server. Since Django is written in Python, it's possible to run it directly from your unit test code. However, it's easiest to just execute it via the shell, either manually with `python manage.py runserver`, or with the `subprocess` module.

Now you know how to create and run twill tests, and we've shown you how to integrate them into a fully automated unit testing framework. Next, we'll discuss using Selenium to test the Django admin interface.

The admin Interface

Start the Django example application, open up a web browser and go to <http://localhost:8000/admin/>. You should see a user login box; this is the entryway to the automatically generated Django site administration system.

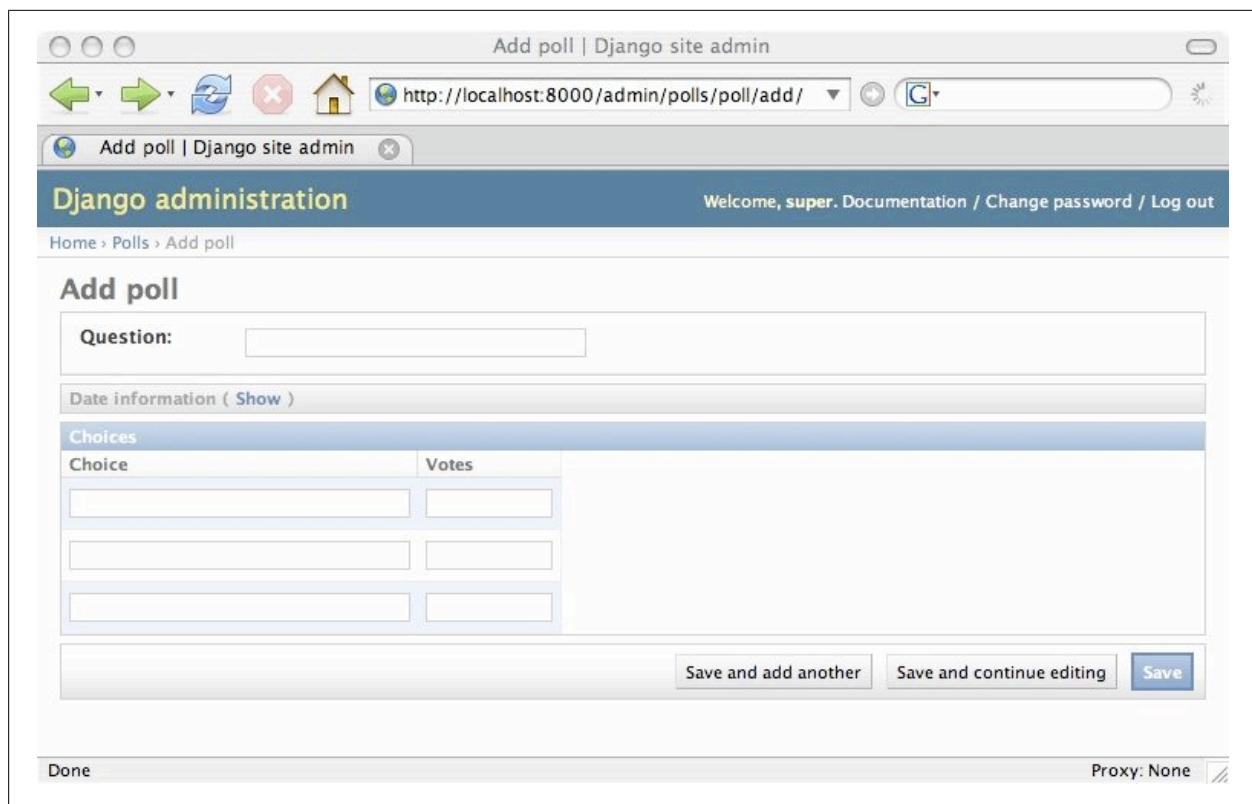


Figure 6.

Enter the username 'super', password 'user'. This is the administrator account we set up for the application.

You should now be at the main "Django administration" page, which lists several different site administration functions. We're going to familiarize you with the process of adding a poll, so that we can record an automated test for it in the next section.

Start by selecting the "Add" button on the line with "Polls" next to it (**Figure 6**).

Now you're at the "Add Poll" page. Make up a question—let's use "What is the best web programming language?"—and type it into the "Question" field.

For choices, let's enter "Python," "Ruby," and "Java." Make sure to set the initial number of votes to 0—we want this to be fair (**Figure 7**). (You'll get an error if you leave any of the fields blank, so be sure to put something in, even if it's "PHP" and not "Ruby.")

Now try clicking on 'Save', lower right.

It should come back with a request to "Please correct the errors below;" but what errors? Click on the "Show" link next to the Date information. Yep, those fields

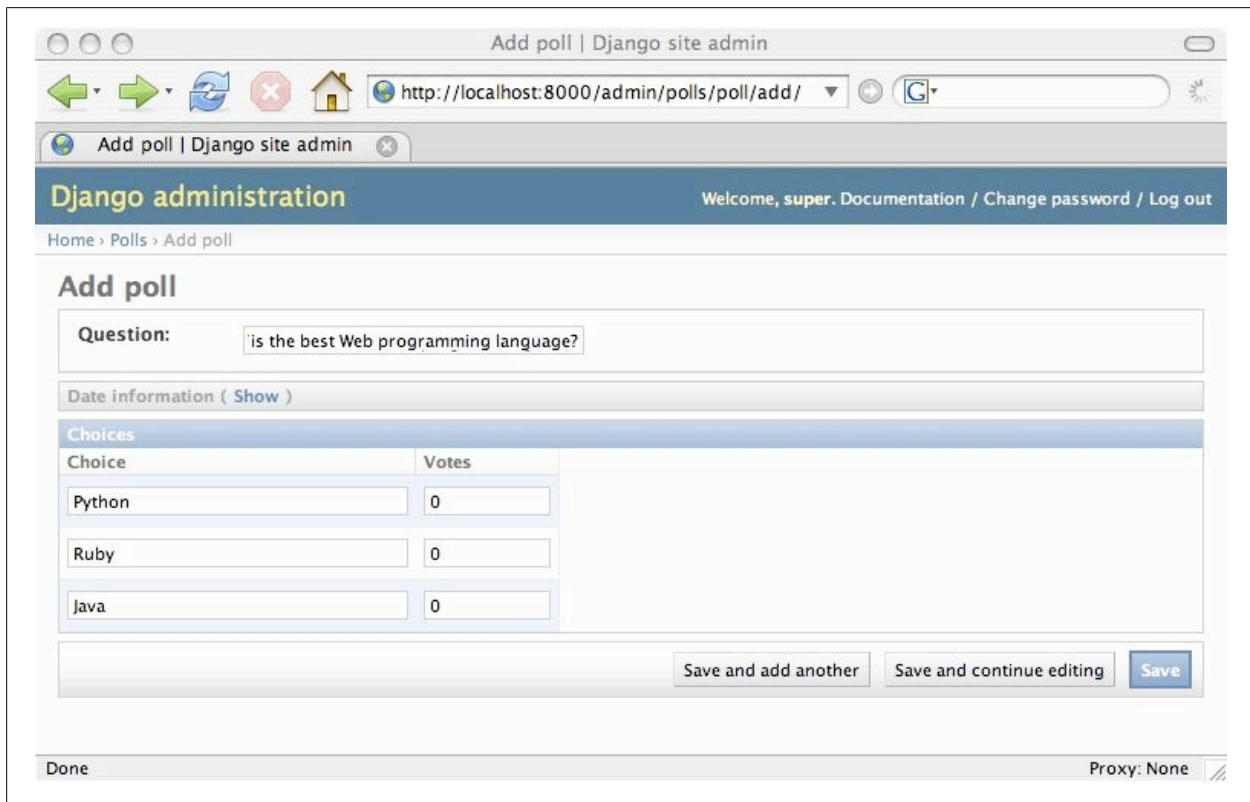


Figure 7.

need to be filled in too! Click on "Today" and "Now" to enter the defaults, and click "Save" again (**Figure 8**).

This time it should add the poll, and send you to the "Django administration" page where you'll see the new poll listed (**Figure 9**).

Let's try it out! Select "Log out" (upper right) and then go to <http://localhost:8000/polls/>. You will see a link, "What is the best programming language?", and if you click on that link you'll see your poll. Select one and vote on it; the results page will reflect your vote (**Figure 10**).

This process—logging into the admin pages, adding a poll, setting the poll name and answers, saving it, and then making sure that the poll was added and works—is another functional path that we want to automated for testing purposes. The main difference between this path and the simpler process of voting that we automated with twill, above, is that now we need to contend with the more complicated admin interface, which uses JavaScript to alter the appearance of the forms. While we can still test the process of adding a poll with twill (it's still just filling out forms) we *can't* test the user interface part of the process with twill, because of the JavaScript code.

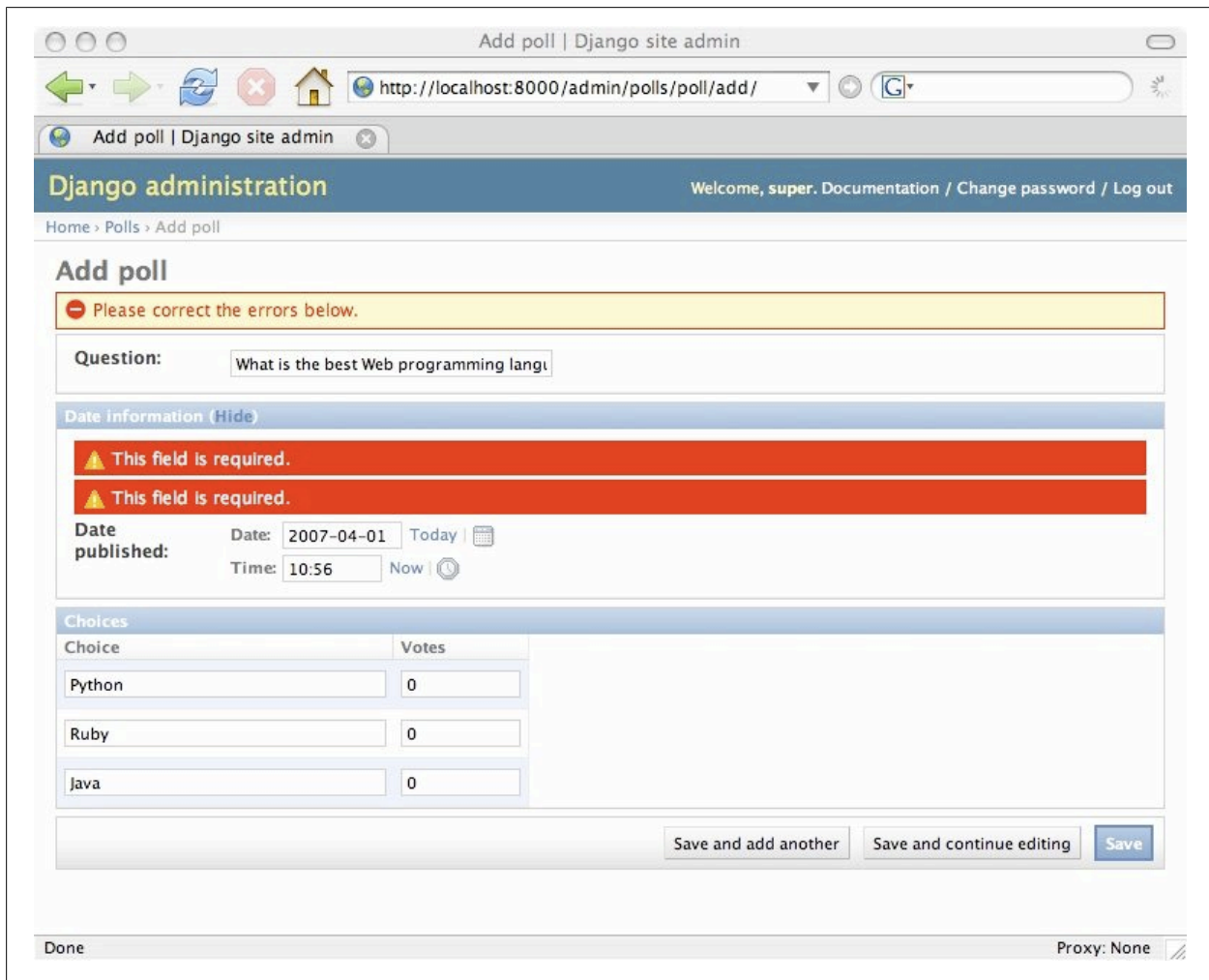


Figure 8.

Recording Some Simple Selenium Tests

One easy way to start building Selenium tests is to use the Selenium IDE to record actions. The Selenium IDE is a Firefox plugin; you will need to have Firefox (<http://www.getfirefox.com/>) and the Selenium IDE (<http://www.openqa.org/selenium-ide/>) installed to follow along with this section.

Run the Django poll site and open <http://localhost:8000/admin/> in Firefox. Then go to the Tools menu and select "Selenium IDE." You will see a blank Selenium IDE window open up. At this point, it is ready to record.

The Selenium IDE is a great way to *start* building a test, because it can record a set of actions and pages. It cannot, however, figure out what you want to actually test on each page. So first we will record a base set of actions and then go through each one and add some individual assertions.

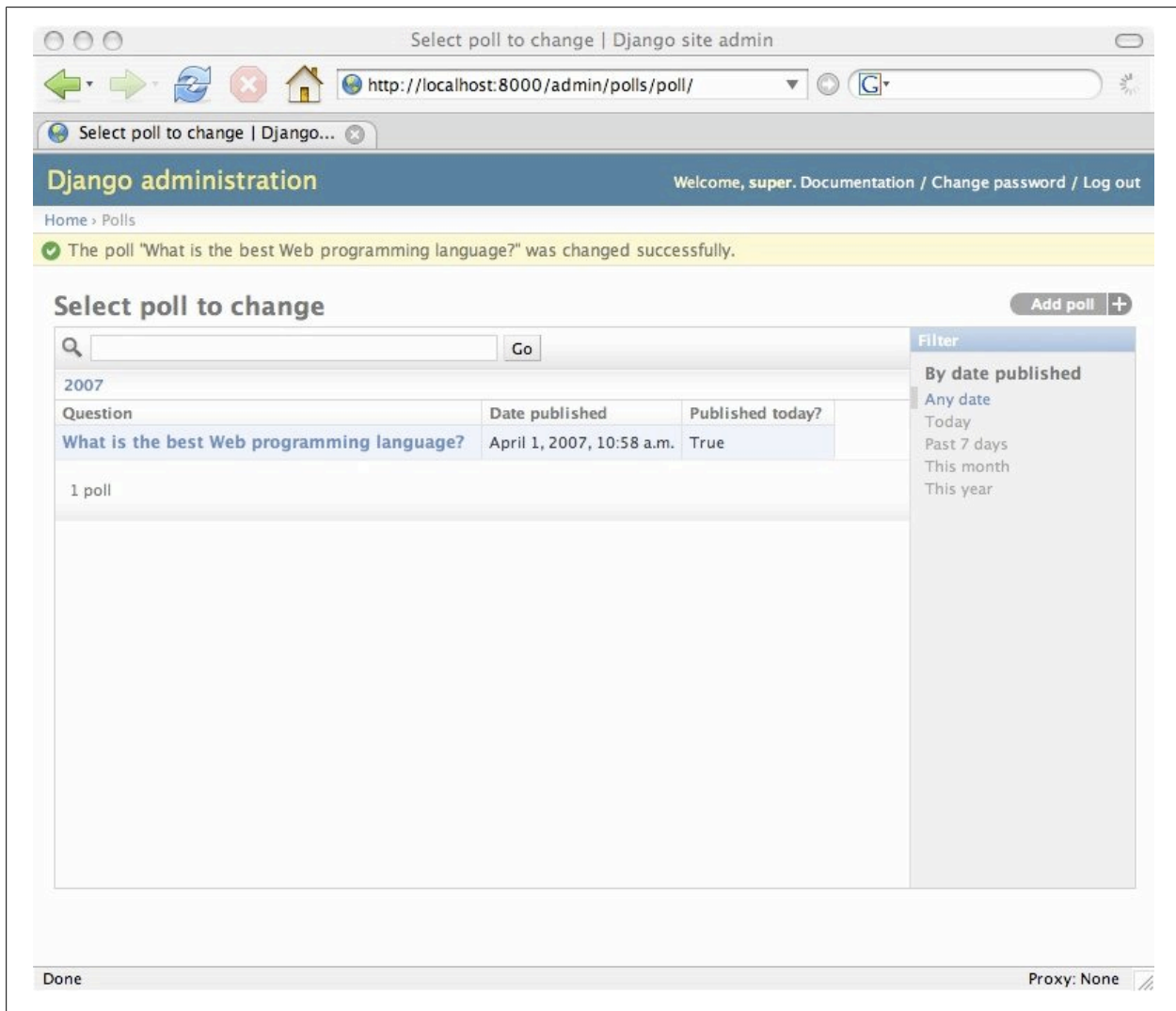


Figure 9.

Let's start by running through all of the actions we performed above (although we'll use a different question, so that we can distinguish between the question we entered earlier and the question we will add now). Briefly:

1. Go to <http://localhost:8000/admin/>.
2. Enter username 'super,' password 'user,' and click on "Log in."
3. Select "Add" next to "Polls."
4. Enter the question "What is the best programming language?", with choices "C," "Python," and "Java." Be sure to set the "Votes" count to 0.
5. Click on "Show" next to "Date information", and select "Today" and "Now."
6. Select "Save" on the lower right, and verify that your new poll question shows up on the list of polls.



Figure 10.

7. Select "Log out."
8. Go to <http://localhost:8000/polls/>
9. Click on "What is the best programming language?" to get to the voting form.
10. Select "Python" and submit the form.
11. Verify by eye that the vote next to your choice has been incremented.

Once you've done all of this, go to the Selenium IDE window (under the "Window" menu) and stop recording by unchecking the little red button on the upper right of the window. You should see something like **Figure 11**. This is the recorded list of all of the actions you just performed in Firefox, in the language of Selenium Core!

Let's try playing it back. Click on the green "Play" button in the middle of the Selenium IDE tool bar. You will see the browser go through all of the actions again, adding another poll (with the same question). It should stop with an error ("Element link=exact:What is the best programming language? not found"; see **Figure 11**) because it is not at the /polls page. This is because the Selenium IDE didn't record the action when you typed in '<http://localhost:8000/polls/>'. To put it in, right-click on the failed command (it's highlighted in pink) and select "Insert command," enter the command "open," with the target "/polls/". Then go back to the top, select the first command, and hit the green "play" button again (**Figure 12**). This time the script should work!

Now, before we work on the script any more, let's integrate it into the Django application itself. To do this, we need to save it into a directory that will be served from the Django application: in this case, we've established a static file directory under `django-poll/selenium-tests/`. Go up to the Firefox File menu and save it as

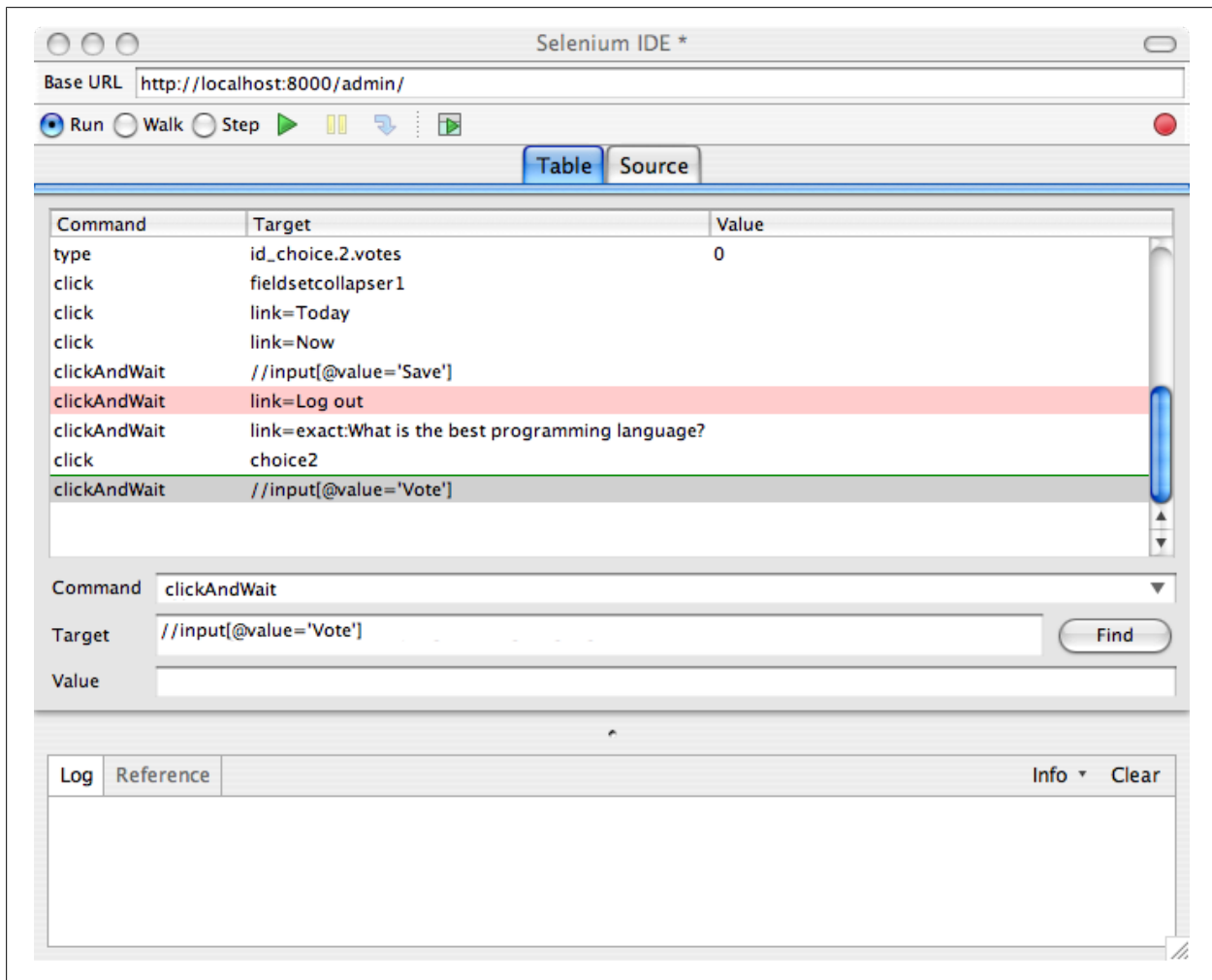


Figure 11.

reader.html in this directory (overwrite the existing empty file). This file is now accessible via <http://localhost:8000/selenium/reader.html>; if you load it, you can see that it's just standard table HTML.

To run this test from inside your application, go to <http://localhost:8000/selenium/index.html> and select "Selenium Tests." The Selenium Functional Test Runner will come up, with two tests, "Admin tests" and "Reader-defined tests" on the upper left. (This list of test names is defined in <http://localhost:8000/selenium/TestSuite.html>, and the link in index.html just points Selenium at this file.) Select the "reader-defined tests" to load in the tests you've just saved, and hit the second green "Play" button in the Selenium TestRunner frame. You will see Selenium run through the Django application in the bottom window, performing all of the actions you recorded.

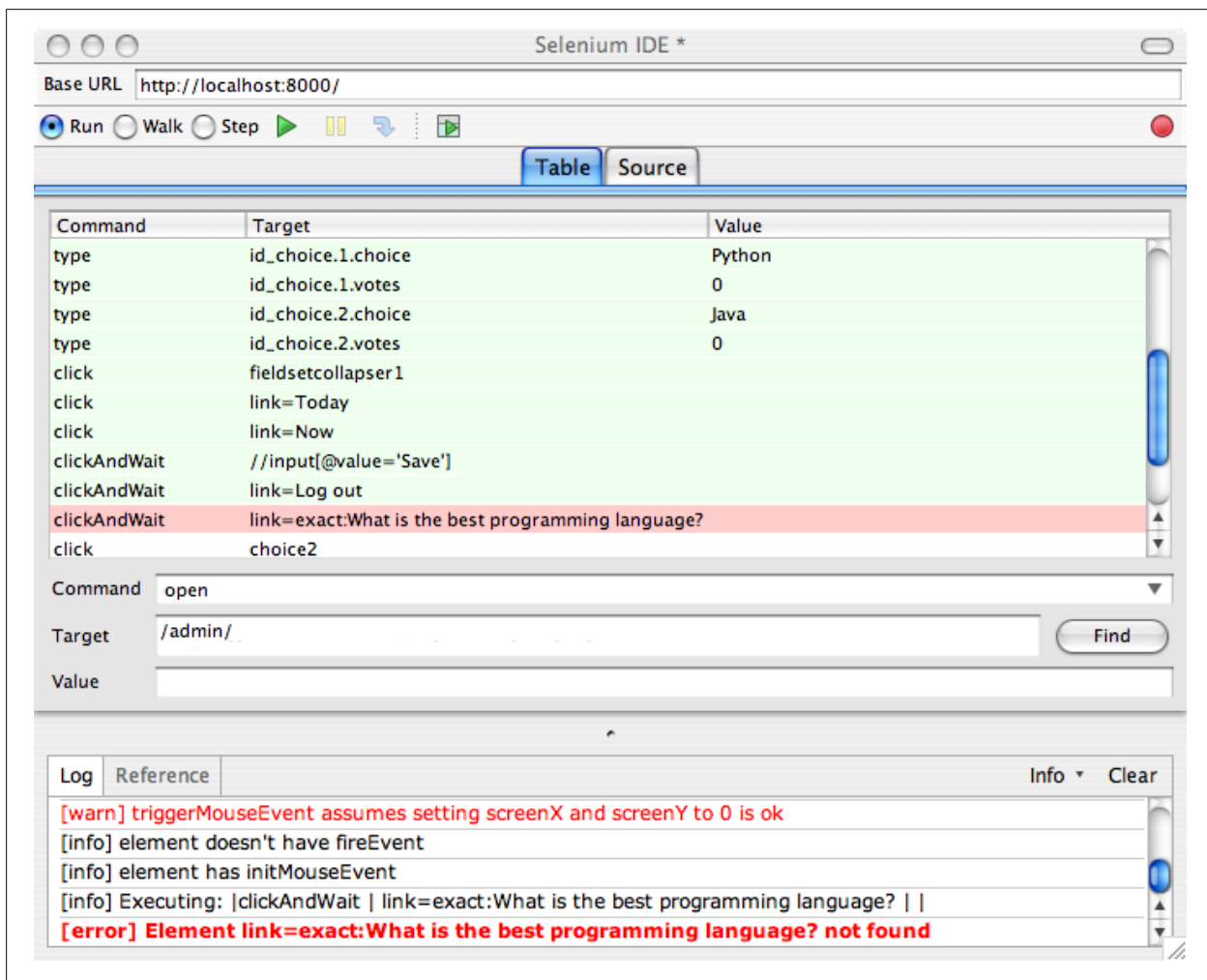


Figure 12.

You *could* just run these tests from inside the Selenium IDE, but you'd omit one of the big advantages of Selenium: cross-browser compatibility. Tests run from within the Selenium Functional Test Runner can be run from *any* Selenium-supported browser, including not only Firefox but IE, Opera, Safari, and Konqueror. (Try it for yourself!) Also, of course, tests saved into the web application can be put into version control and made available to other developers more easily.

Adding a Test Condition to the Script

So far, our recorded Selenium test script automates the process of adding a poll through the administration interface of our Django application. However, it's not really much of a test, because this Selenium script will still complete correctly in a number of error situations. For example, if you remove the second `clickAndWait` command to save the new poll, the script still completes, because we aren't testing whether or not the poll is *actually* added. Let's start by fixing that.

How do we make sure that the poll was actually added? One simple way is to check for the confirmation text displayed by Django, which displays 'The poll "What part of the day is it?" was added successfully' upon success. To check for this text, add the following command into your Selenium test file just above the `clickAndWait | link=Log out` command:

```
<tr>
  <td>verifyTextPresent</td>
  <td>The poll "What part of the day is it?" was added successfully.</td>
  <td></td>
</tr>
```

This checks to make sure that that text is present somewhere on the page.

Now try running the test again; it should pass. If you want to test your test (always a good idea!), alter the text in the argument to `verifyTextPresent` and make sure the test fails.

Let's add two more tests; right after the first `clickAndWait | //input [@value='Save']` command, where a date and time field are required, we can check to make sure that the proper errors are reported:

```
<tr>
  <td>verifyTextPresent</td>
  <td>Please correct the errors below.</td>
  <td></td>
</tr>
<tr>
  <td>verifyTextPresent</td>
  <td>This field is required.</td>
  <td></td>
</tr>
```

Some notes on these Selenium tests:

- As with `twill`, we recommend defining a functional path through the application and writing a single test for that path. You can then "decorate" that path with tests for specific errors later on.
- In our Selenium example above, we carefully avoided the issue of fixtures. For example, rather than checking that our poll was added to the full list of polls, we verified the presence of the message confirming that the poll was added. What if we had wanted to test for the presence of the new poll in the list? We would have had to distinguish the poll question we were adding from existing polls in some way, either by using a random indicator in the poll question, or by looking at the time it was added, or by making sure that we deleted existing polls that had the same name.

Of course, we could also have simply added hooks into the web application to reset the list of polls. However, these hooks would need to be accessible to the client web browser, which raises questions of security: if we add client-side hooks for testing purposes, we have to be sure to disable those hooks when we deploy the application, which in turn renders the deployed application untestable.

A better option is to write your Selenium tests to depend only on behavior that should be accessible to clients. In this case, probably the best option would be to add in a cleanup section at the end of the test that deletes the poll we've just added. (This is left as an exercise for the reader!)

- Selenium tests are considerably more charismatic than twill tests. They are an excellent way to demo web site features to managers and customers.
- Selenium tests can also make sure that your web site is functional in multiple browsers. This is an essential feature for testing JavaScript-heavy web sites.
- However, Selenium tests are also much heavier-weight than twill tests, because Selenium does require a full browser. We recommend using twill to test server-side functionality, and using Selenium for testing client-side functionality.

Concluding Remarks

In this short cut, we've demonstrated how to write twill and Selenium tests for a simple (yet still interesting) web site. We hope that this gives you some ideas of how to use these tools for your own web development. In particular, we strongly support "testing everywhere," which has been difficult with web sites, especially with JavaScript-heavy web sites. With Selenium and twill, you don't have to be afraid of writing untestable web code.