Automating JavaScript Unit Tests with Python

Mitch Chapman
Mesa Analytics & Computing, Inc.
212 Corona Street, Santa Fe, NM 87501
http://www.mesaac.com/

Motivation

Web applications span multiple programming languages and execution environments. Python can help automate testing for many of those languages and environments – including JavaScript running within a web browser.

Testing Web Applications

With the advent of Ajax, much responsibility for user interaction in web applications has moved from the web server to the browser, and to JavaScript.

JavaScript developers have created toolkits for testing client-side code. QUnit[1] is one example. It offers functions for defining test suites and test cases, and it displays test results within the browser.

Automating Test Invocation

Desktop applications typically run in the same environment – an interactive shell, for example – as the revision control systems in which their source code resides. This makes it easy to create pre-commit hooks which ensure all unit tests pass before allowing any code changes to be shared with the world.

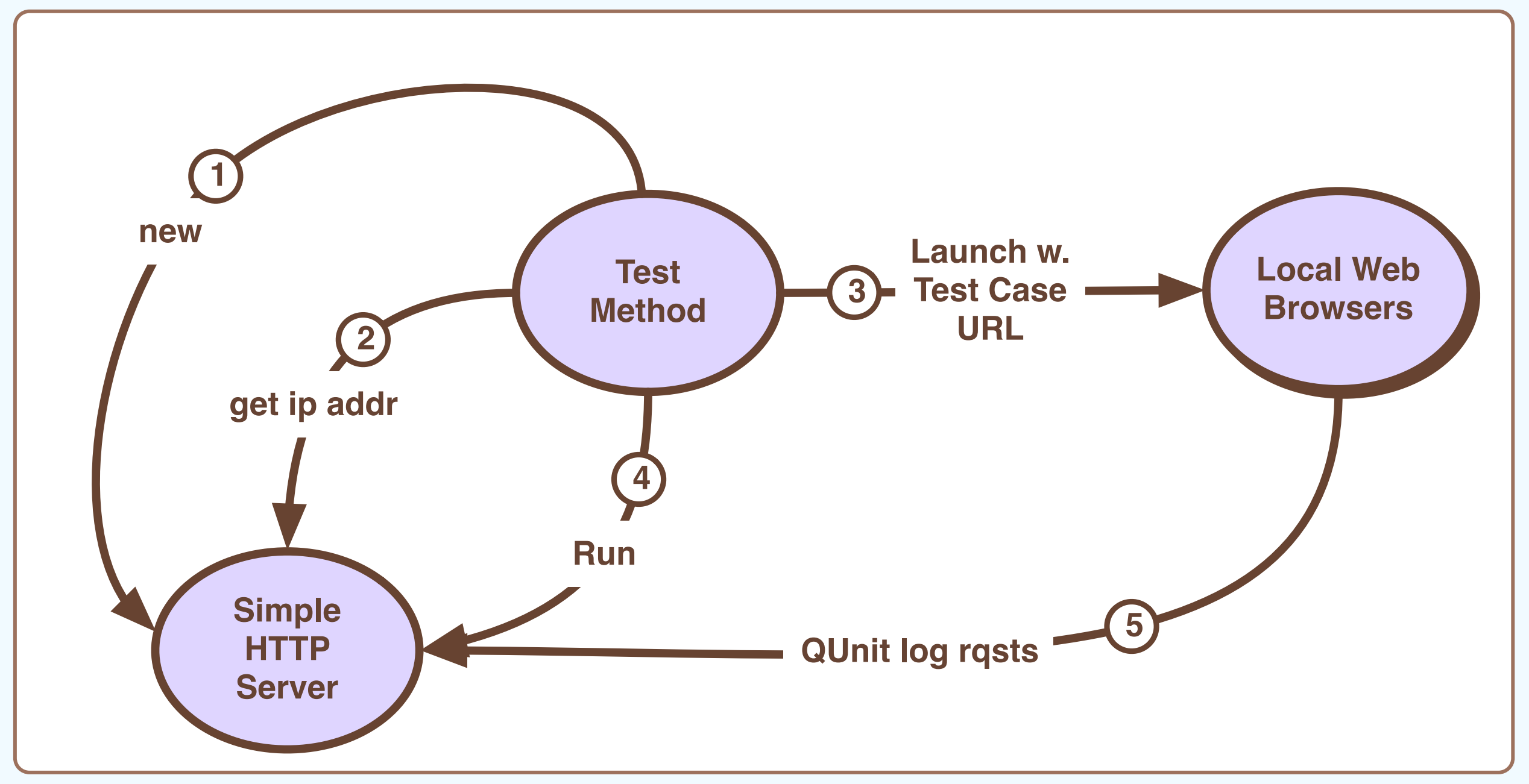
Not so with applications that run in a web browser. To ensure that these applications behave as expected, their developer must manually load and run the tests in all supported browsers, verifying manually that all tests pass, before committing any source code changes.

It would be nice if pre-commit hooks could run both desktop and browser-based tests.

This poster shows one way to automate execution of browser-based (JavaScript) tests. The approach shown here is an alternative to py.test[2] and oejskit[3]; it may be useful in environments where py.test cannot be used.[4]

Strategy

Define a unittest test method representing each JavaScript unittest suite, as follows:



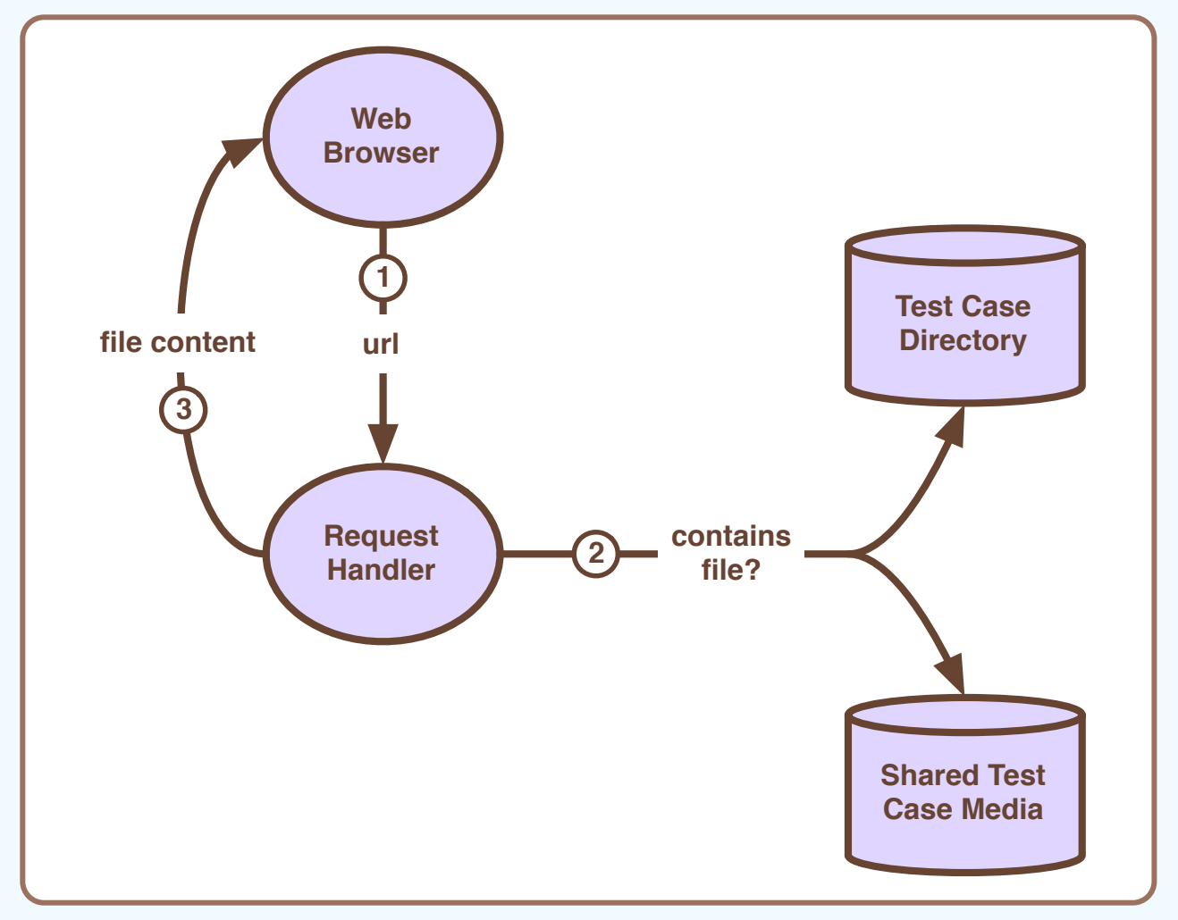
1. Instantiate a SimpleHTTPServer subclass, binding it to any available TCP/IP port.
2. Get the address at which the server is listening.
3. Launch an instance of each supported web browser in turn. Direct each web browser to the server's IP address, and the URL path of the JavaScript test page.
4. Run the SimpleHTTPServer, which provides POST URLs corresponding to the logging "hooks" in QUnit.
5. In the browser, run the JavaScript tests. Start by adding QUnit hook functions. Each function reports a QUnit action – start of a test suite, execution of an assert, etc. – by making an Ajax POST to the SimpleHTTPServer.
6. (Not shown) Shut down the SimpleHTTPServer in response to either a QUnit done action or a timeout preceding the first QUnit POST.

Serving Test Case Media

The server subclass provides its own SimpleHTTPRequestHandler subclass, for handling incoming requests. The request handler class translates incoming GET requests into requests for static files, e.g., for the HTML, JavaScript and CSS comprising the current test case.

The test server and request handler resolve request paths relative to two "docpaths", or directories:

- The directory containing the JavaScript test case. This directory typically contains index.html, JavaScript and CSS files specific to the test.
- The directory containing all unit tests. This directory may include CSS, JavaScript and HTML which are shared among all of the unit tests.



Other search locations may be added.

When the request handler processes a GET request, it tries to resolve it to a path rooted at one of the docpaths.

When the request handler processes a POST request, it resolves it to a QUnit handler method defined by the test server. This lets the test server maintain QUnit processing state across multiple requests.

Configuring Browsers

```
import webbrowser as wb
if sys.platform[:3] == "win":
    for browser in ("chrome",):
        if wb._iscommand(browser):
            wb.register(
                browser, None,
                wb.BackgroundBrowser(browser))
```

Python's webbrowser module might not offer bindings for all of the types of browsers available on your platform. For example, on Windows Internet Explorer is included, but Google Chrome is not. You can easily add support for additional browsers.

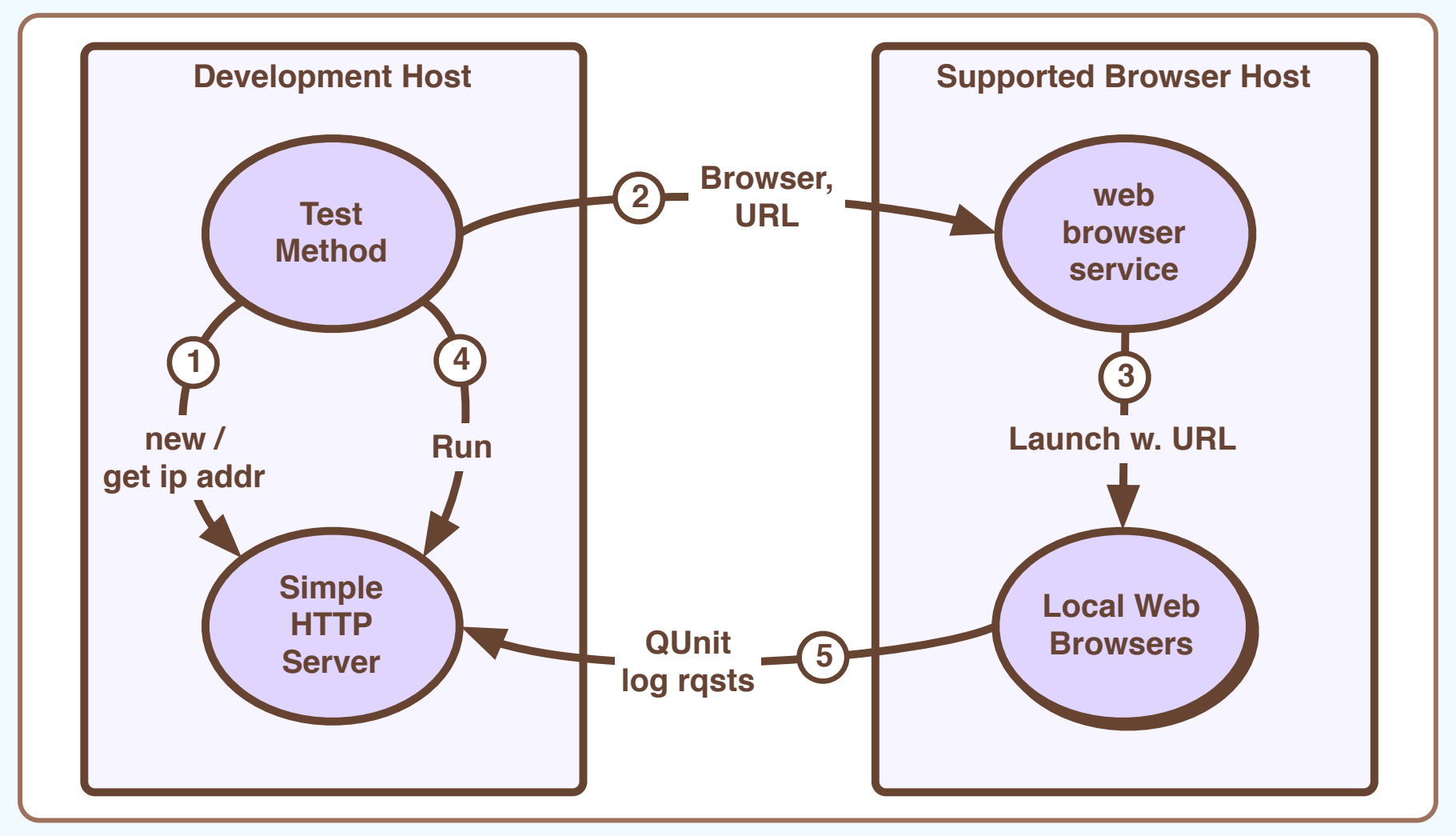
Launching Remote Browsers

Most web apps need to be able to run on multiple operating systems. To automate testing with such remote browsers, some one-time software installation and configuration are needed.

On participating remote hosts, a "remote web browser service" script needs to be installed and run. This script runs a SocketServer, listening on a well-known port for remote browser service requests.

With all required web browser services running, test execution proceeds as follows:

1. The test method instantiates a SimpleHTTPServer and records its address, as before.
2. The test method launches an instance of each supported web browser in turn. To launch a remote web browser it sends a (browser, URL) request to the remote web browser service.
3. The service handles the request by launching the requested browser type, and directing it to the URL. It reports back to the test method whether or not the launch succeeded.



4. The test method runs the SimpleHTTPServer as before.
5. The browser logs QUnit requests from the remote browser as before.

Source Code

You can download the source code for this poster from http://bitbucket.org/mchapman87501/pycon_js_testing/

Future Work

Django Integration

If your JavaScript code is meant to access specific URLs of a Django application, this strategy does not address creating mock versions of those URLs. It does not even take advantage of Django's URL dispatch mechanism.

The Django test client doesn't help directly, in part because it does not generate actual network traffic.[5]

One workaround is to use static files, for both GET and POST requests, to mimic the dynamic content generated by Django applications. Another is to implement incoming requests by invoking corresponding test client methods, then relaying the client method results back to the browser.

Multi-Platform Testing

This strategy uses hardwired hostnames for machines on which to run remote browsers. It also requires remote web browser services to be started manually.

Although this works, it makes the unit tests less portable than they should be. A developer can't simply check out a copy of the test code on a new host and expect it to run, unless all of the specified remote web browser services are reachable from the new machine.

It should be easier for a developer to check out and begin using a workspace, without performing much manual test environment configuration.

References

1. http://docs.jquery.com/QUnit
2. http://codespeak.net/py/dist/test/
3. http://www2.openend.se/~pedronis/oejskit/doc/doc.html#rest-of-the-docs
4. Or just use oejskit! When submitting this poster I was not aware of the extent of its support for unittest.
5. http://docs.djangoproject.com/en/1.1/topics/testing/#module-django.test.client