Advertise Here

Headless Functional Testing with Selenium and PhantomJS

Denis Sokolov on Mar 5th 2013 with 40 Comments

Tutorial Details

-
- **Difficulty:** Advanced
- **Completion Time:** 1 Hour

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

Let’s build a system for performing functional tests on web applications, using Selenium and PhantomJS. The resulting system will allow us to write easy test scenarios in JavaScript, and test those scenarios both in real browsers and a headless simulator.

Choosing Components

> The obvious downside to Selenium is that it requires a full graphical desktop for any and all tests.

To begin, we have to choose a browser control or emulation engine to simulate an end–user. For a long time, the primary player in this field was Selenium, and it still is. Selenium allows for automated control of real browsers on real operating systems, which is its primary advantage: you can be absolutely certain that the tests represent reality as closely as possible.

The obvious downside to Selenium is that it requires a full graphical desktop for any and all tests. As a result, your tests can become slow. However, Selenium can be fantastic, if you have the necessary resources to set up virtual machines for different operating systems and connect it all together.

On the opposite end of the spectrum is PhantomJS: A small, but excellent project, running a WebKit engine with full JavaScript access, but without the graphical portion. PhantomJS is a cinch to set up, runs on any machine, and is significantly faster.

> Selenium can now control PhantomJS in the same way that it does any other browser.

PhantomJS, being a full WebKit, covers 90% of your functional testing needs. After all, if your application runs in WebKit correctly, it’s likely that it will run correctly in other browsers. *Obviously, this excludes Internet Explorer 6–8.*

However, as your project becomes increasingly popular, that remaining 10% becomes a significant issue. If your functional testing suite is set up on PhantomJS directly, it would be a pain to rewrite the tests for Selenium.

Luckily, recently, near the tail end of 2012, we received a gift in the form of PhantomJS bindings to Selenium. In other words, Selenium can now control PhantomJS in the same way that it does any other browser.

Given that Selenium, itself, does not need any complicated setup and can run anywhere, we can use Selenium bindings to control PhantomJS and cover 90% of our testing needs. If you later need more powerful testing, you can set up extra browser connections to Selenium without changing a single line in your code.

Thus, our choice for browser engine are Selenium with PhantomJS.

Describing Tests

Selenium offers bindings in most popular programming languages, so we can choose a language according to our needs. This is perhaps the most controversial piece of this article: I consider JavaScript to be the best choice for describing functional tests for websites and web applications.

- No matter which back–end technology you use, your front–end will always use JavaScript (*This applies even if you use a language that compiles down to vanilla JavaScript, like CoffeeScript or TypeScript.*). As such, JavaScript will always be an understood language by at least one person on your team.
- Next, consider the possibility for your functional tests to be written by non–programmers. The popularity of JavaScript on the front–end, combined with expressiveness in the ability to create clear domain–specific languages, clearly allows more people to write functional tests.
- Lastly, it's only natural to control a test browser with JavaScript, given that it is highly asynchronous, and is what we control the browser with on a daily basis.

Selenium bindings for JavaScript are called, webdriverjs. Although the project is less mature than officially supported drivers for Java, C#, Ruby and Python, it nevertheless already contains most of the functionality that we require.

Test running

For the purposes of this article, Mocha with Chai have been selected.

Finally, we need a test runner, or an application to run tests by name, and pretty–

print the output, while noting how many tests succeeded or failed. This test runner should also offer an assertion library, which allows the coder to express if a test succeeds or fails.

The choice is absolutely free here. There are plenty of JavaScript test runners, but for the purposes of this article, Mocha with Chai have been selected. Mocha provides a considerable amount of flexibility, a wide variety of output formats, and the popular Jasmine-like syntax. Chai allows you to write descriptive BDD-like assertions.

Setup

Here's the final stack that we'll be using:

1. Mocha – test runner
2. Chai – assertion library
3. webdriverjs – browser control bindings
4. Selenium – browser abstraction and running factory
5. PhantomJS – fast headless browser

Node.js and npm

Because most of our stack is based on JavaScript, we need node.js and npm. Both of these are common tools in the community, and I'll assume that you already have them set up. If you don't, use the installer on the node.js website. Don't worry; if anything goes wrong, there are plenty of Node install guides available around the web.

Mocha, Chai and webdriverjs

All three of these can be installed, using `npm`:

```
sudo npm install -g mocha chai webdriverjs
```

Alternatively, you can install them locally in the directory where your tests are located:

```
npm install mocha chai webdriverjs
```

Selenium

Download Selenium Server. It is distributed as a single `jar` file, which you run simply:

```
java -jar selenium-server-standalone-2.28.0.jar
```

As soon as you execute this command, it boots up a server to which your testing code will connect later on. Please note that you will need to run Selenium Server every time that you run your tests.

PhantomJS

Quick version

Use `npm` to install PhantomJS globally:

```
sudo npm install -g phantomjs
```

Other options

We require a fresh version of PhantomJS – at least 1.8. This means that packages provided by your package manager (`apt-get`, MacPorts, ...) will most likely be outdated.

You can install using npm without a global installation, or using other methods manually. In this case, however, you will have to tell Selenium where you placed PhantomJS every time you run Selenium:

```
PATH="/path/to/node_modules/phantomjs/bin:$PATH" java -jar seleniun
```

Combining Everything

Now that we have all the pieces, we have to put everything together.

Remember: before running any tests, you have to run Selenium Server:

```
java -jar selenium-server-standalone-2.28.0.jar
```

Selenium will run PhantomJS internally; you don't have to worry about that.

Now, we need to connect to Selenium from our JavaScript. Here's a sample snippet, which will initiate a connection to Selenium and have a ready object to control our Selenium instance:

```
// Use webdriverjs to create a Selenium Client
var client = require('webdriverjs').remote({
    desiredCapabilities: {
        // You may choose other browsers
        // http://code.google.com/p/selenium/wiki/DesiredCapabilit
        browserName: 'phantomjs'
    },
    // webdriverjs has a lot of output which is generally useless
    // However, if anything goes wrong, remove this to see more de
    logLevel: 'silent'
});

client.init();
```

Now, we can describe our tests and use the `client` variable to control the browser. A full reference for the webdriverjs API is available in the documentation, but here's a short example:

```
client.url('http://example.com/')
client.getTitle(function(title){
    console.log('Title is', title);
});
client.setValue('#field', 'value');
client.submitForm();
client.end();
```

Let's use the Mocha and Chai syntax to describe a test; we'll test some properties of the `example.com` web page:

```
describe('Test example.com', function(){
    before(function(done) {
        client.init().url('http://example.com', done);
    });

    describe('Check homepage', function(){
        it('should see the correct title', function(done) {
            client.getTitle(function(title){
                expect(title).to.have.string('Example Domain');
                done();
            });
        });

        it('should see the body', function(done) {
            client.getText('p', function(p){
                expect(title).to.have.string(
                    'for illustrative examples in documents.'
                );
                done();
            })
        });
    });

    after(function(done) {
        client.end();
        done();
    });
});
```

You might want to share one `client` initialization over many test files. Create a small

Node module to initialize and import it into every test file:

`client.js`:

```
exports.client = require('webdriverjs').remote({
    // Settings
};
```

`test.js`:

```
var client = require('./client').client;
var expect = require('chai').expect;

// Perform tests
```

Running

Mocha test suites are execute with the `mocha` binary. If you followed this guide and installed Mocha locally, then you should describe a full path to the binary yourself: `node_modules/mocha/bin/mocha`.

By default, Mocha treats any test that takes longer than two seconds as failed. Given that we are actually initializing a web browser and making an HTTP request, we need to increase this timeout to `5` or `10` seconds:

```
node modules/mocha/bin/mocha test.js -t 10000
```

If everything went according to plan, you should see output like this:

```
.

✔ 1 <span class="nb">test complete</span>
```

The Next Steps

Once you've achieved your desired functional testing results, you may want to consider improving your setup further.

Two obvious directions are continuous integration and distributed Selenium testing.

Continuous integration

Your goal should be to minimize the time that you spend running tests.

You might want to use a full automatic continuous integration server, which will run the tests whenever needed automatically, and inform you if anything goes wrong.

In the world of open source, the role of such a server is covered by Jenkins CI: a convenient, powerful, easy to install service, which will run the tests whenever needed, execute them in any configuration that you provide, and possibly run many more build–related tasks, such as deploying your code to remote servers.

Alternatively, if you feel adventurous, you might experiment with a new project, called GitLab CI, which offers less features, but looks better and is integrated with GitLab, a self hosted GitHub clone.

In any case, your goal should be to minimize the time that you spend running tests. Instead, the tests should be run automatically and should only inform you if anything goes wrong.

Selenium Grid

Selenium has a number of implementation limitations. For example, you cannot run more than a few browsers on the same machine to be tested with Selenium.

In addition, you will notice that, once you have many tests, running all of them can become a lengthy process. Although continuous integration partly alleviates this problem, you might still want to run some tests in parallel on different machines.

Finally, you will soon notice that you want to test different browsers on different operating systems. And, while your testing code can, in theory, talk to different Selenium servers, once you grow a little, this setup needs centralization.

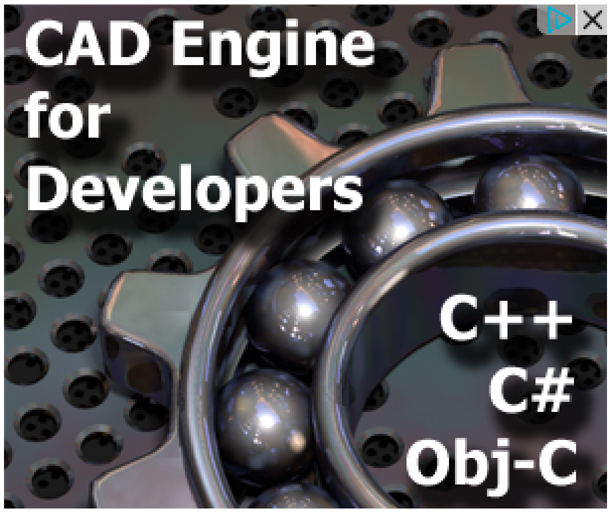
Selenium Grid setup tries to provide exactly that. Instead of having one Selenium server control a bunch of browsers on a machine, you have one Selenium server, which controls multiple Selenium nodes, each which controls only a few browsers on a single operating system.

Conclusion

The resulting stack, although not trivial, in reality, is quite simple. The addition of PhantomJS to the Selenium end allows us to begin using Selenium without much initial investment, such as setting up graphical test servers.

The usage of JavaScript as a testing engine ensures that our tests will be kept relevant in the context of web development for the foreseeable future.

Like 90 people like this. Be the first of your friends.



Tags: phantomjsseleniumTesting

By Denis Sokolov

This author has yet to write their bio.