Stabilizing the Tests

> *"And the rain descended, and the floods came, and the winds blew, and beat upon that house; and it fell not: for it was founded upon a rock."*
>
> *--Matthew 7:25, King James Version*

When the test suite becomes large enough, our job becomes less about the fixing every flaky test. In fact, it centers on engineering a solution that will prevent all similar flaky behavior from happening.

In this chapter, we will give our tests a good solid foundation that will prevent a lot of instability in the long run. We waited until this chapter to start fixing the behavior that drives anyone who writes web tests insane, because we had to first build up a foundation of good data management and coding skills. These skills are crucial for long-term use and without them, all of the fixes of instability discussed in this chapter would be useless. Now we're ready to talk about the following topics:

- Culture of stability
- jQuery
- Waiting for AJAX requests to finish
- Waiting for jQuery animations to finish
- The Action Wrapper pattern
- The Black Hole Proxy pattern
- Screenshot on failure practice

Engineering the culture of stability

I'd like to start the current chapter with a personal tale of a past experience. The majority of projects that I worked on had similar situation to what you are probably used to. Typically, the Selenium build is treated as a second-class citizen, not having a single passing build for days or weeks at the time. Eventually, the tests become so embarrassingly riddled with failures and instabilities that any further development is stopped, and the Selenium build is completely ignored.

On my last project I inherited 300 Selenium tests, which were red 90 percent of the time. So, I started to fix them but that was not enough; no sooner that I would fix a broken test, somebody would make a commit that broke another test somewhere else. I did not have a technical problem, I had a cultural problem; nobody but me seemed to care about Selenium tests.

The team that I was a part of was given the task of maintaining builds; with a lot of trial and error, we came up with several key goals that would lead all of our builds to be

passing 99 percent of the time (less actual failures due to bad code). Here are the key goals, as I see them, for any CI system:

- Running fast and failing fast
- Running as often as possible
- Keeping a clean and consistent environment
- Discarding bad code changes
- Maintaining a stable test suite

Running fast and failing fast

A developer's time is very expensive. We cannot afford to let them sit around for 40 minutes to see whether all of the tests are passing after every minor code change. The goal is to run the whole test suite under 10 minutes, or the developer will not have an incentive to run the tests at all. Doing the simple math of the man hours spent by each developer on daily basis waiting for the build, compared to doing actual work, we had a very convincing argument to purchase a lot more test nodes for CI. With these new computers, we were able to run the test suite in parallel across multiple computers, bringing the whole build down to 12 minutes. Furthermore, we added some code to send an e-mail to the developer as soon as a test failed. This allows the developers to start fixing a broken test even before the build is complete.

Running as often as possible

Creating a cascading build chain, starting with unit tests and finishing with Selenium, is a common practice. However, this practice turned out to be an *anti-pattern*, a term discussed in Chapter 2, *The Spaghetti Pattern*. A typical Selenium build is the slowest in the series; thus, it occupies the last place where everyone can easily ignore it. Often, a failure early in the chain will prevent the Selenium build from ever being executed. By the time the long forgotten Selenium build is finally executed, a dozen code commits have occurred. Making sure that the Selenium build is triggered on every single commit seems excessive, but the whole idea of CI is to catch a test failure as soon as it occurs, not 20 changes down the road. Taking this idea to its logical conclusion, a code change should always be considered bad if even a single test fails.

TIP

Having the whole code base being deployed and tested with every code change also has an advantage of testing the deploy scripts continuously.

Keeping a clean and consistent environment

Unlike instability caused by test implementation, instability caused by inconsistent testing nodes can be more frustrating and harder to track down. Having different

versions of Firefox or Internet Explorer on every test node might not seem like a big deal, but when a test fails because of such minor differences and the failure cannot be easily replicated, a lot of frustration will be experienced.

We discussed test fixtures in Chapter 4, *Data-driven Testing*; reloading the test database for every build is a great way to keep a clean and consistent test environment. Also, using a configuration management tool to keep all of the dependencies, such as Java versions, consistent on all of the test nodes will save you a lot of headaches. Finally, make sure that the test environment that serves your website is as close of a physical clone of production as you can make it. All of your tests can be completely invalid if your production uses Linux servers to host the website, but your test environment is hosted on a Windows computer.

NOTE

There are several open source, free tools for the configuration management of computers. Two of the more popular ones are Chef (http://www.getchef.com/) and Puppet (http://puppetlabs.com/).

Discarding bad code changes

We set up a simple system that prevented anybody from committing changes to the master/trunk unless all of the tests, including Selenium, were passing. Needless to say, this was not a popular approach because tests from unrelated parts of the application were sometimes preventing new features from going into **Version Control System** (**VCS**). However, as the test suite stabilized, this became a great way to prevent unintended defects from going into production, and making sure that the whole test suite, including Selenium, was always passing!

NOTE

There are multiple ways to implement this, since most VCS systems allow users to define precommit or postcommit hooks. The other approach is to prevent direct commits to the trunk/master branches, instead deferring to a build that automatically merges the changes after all tests pass. The latter approach works best in GIT and Mercurial VCS tools.

Maintaining a stable test suite

Cultural changes will never last if your tests will fail at random due to technical problems such as not dealing with **AJAX** properly or not accounting for external influences that will make the test environment run slow. In this chapter, we will concentrate on some of the most common technical solutions that make tests unstable. Let's get going!

NOTE

Asynchronous JavaScript and XML (**AJAX**) is a relatively new web development technique that allows the web page to send and receive content in the background.