Best Practices for Running Tests

Avoid External Test Dependencies

Use Setup and Teardown

- If there are are "prerequisite" tasks that need to be taken care of before your test runs, you should include a `setup` section in your script that executes them before the actual testing begins. For example, you may need to to log in to the application, or dismiss an introductory dialog that pops up before getting into the application functionality that you want to test.
- Similarly, if there are "post requisite" tasks that need to occur, like closing the browser, logging out, or terminating the remote session, you should have a `teardown` section that takes care of them for you.

Don't Hard Code Dependencies on External Accounts or Data

Development and testing environments can change significantly in the time between the writing of you test scripts and when they run, especially if you have a standard set of tests that you run as part of your overall testing cycle. For this reason, you should avoid building into your scripts any hard coded dependencies on specific accounts or data, Instead, use API requests to dynamically provide the external inputs you need for your tests.

Avoid Dependencies between Tests to Run Tests in Parallel

Dependencies between tests prevent tests from being able to run in parallel. And running tests in parallel is by far the best way to speed up the execution of your entire test suite. It's much easier to add a virtual machine than to try to figure out how to squeeze out another second of performance from a single test.

What are dependencies? Imagine a test suite with two tests:

Java Example of Test Dependencies

```
@Test
public void testLogin()
{
    // do some stuff to trigger a login
    assertEquals("My Logged In Page", driver.getTitle());
}

@Test
public void testUserOnlyFunctionality()
{
    driver.findElement(By.id("userOnlyButton")).click();
    assertEquals("Result of clicking userOnlyButton",
driver.findElement(By.id("some_result")));
}
```

PHP Example of Test Dependencies

```
<?php
function testLogin()
{
    // do some stuff to trigger a login
    $this->assertEquals("My Logged In Page", $this->title());
}

function testUserOnlyFunctionality()
{
    $this->byId('userOnlyButton')->click();
    $this->assertTextPresent("Result of clicking userOnlyButton");
}
```

In both of these examples, `testLogin()` triggers the browser to log in and asserts that the login was successful. The second test clicks a button on the logged-in page and asserts that a certain result occurred.

This test suite works fine as long as the tests run in order. But second test makes an assumption that you are already logged in, which creates a dependency on the first test. If these tests run at the same time, or if the second one runs before the first one, the browser's cookies will not yet allow Selenium to access the logged-in page, and the second test fails. You can get rid of this dependency by making sure that each test can run independently independently of the others, as shown in these examples.

Java Example of Corrected Test Dependency

```
@Test
public void doLogin()
{
 //do some stuff to trigger a login
 assertEquals("My Logged In Page", driver.getTitle());
}

@Test
public void testLogin()
{
 this.doLogin();
}

@Test
public void testUserOnlyFunctionality()
{
 this.doLogin();
 driver.findElement(By.id("userOnlyButton")).click();
 assertEquals("Result of clicking userOnlyButton",
driver.findElement(By.id("some_result")));
}
```

PHP Example of Corrected Test Dependency

```
<?php
function doLogin()
{
    // do some stuff to trigger a login
    $this->assertEquals("My Logged In Page", $this->title());
}

function testLogin()
{
    $this->doLogin();
}

function testUserOnlyFunctionality()
{
    $this->doLogin();
    $this->byId('userOnlyButton')->click();
    $this->assertTextPresent("Result of clicking userOnlyButton");
}
```

The main point is that it is dangerous to assume any state when developing tests for your app. Instead, you should find ways to quickly generate desired states for individual tests. In the example, this is accomplished with the `doLogin()` function, which generates a logged-in state instead of assuming it. You might even want to develop an API for the development and test versions of your app that provides URL shortcuts that generate common states. For example, a URL that's only available in test that creates a random user account and logs it in automatically.

Don't Use Brittle Locators in Your Tests

WebDriver provides a number of locator strategies for accessing elements on a webpage. It's tempting to use complex XPath expressions like `//body/div/div/*[@class="someClass"]` or CSS selectors like `#content .wrapper .main`. While these might work when you are developing your tests, they will almost certainly break when you make unrelated refactoring changes to your HTML output.

Instead, use sensible semantics for CSS IDs and form element names, and try to restrict yourself to using these semantic identifiers. For example, in Java you could designate elements with `$this->byId()` or `$this->byName()` or, in the example of PHP, you could use `$this->byId()` or `$this->byName()` . This makes it much less likely that you'll inadvertently break your page by shuffling around some lines of code.

Have a Retry Strategy for Handling Flakes

There will always be flaky tests, and tests that once breezed through with no problem can fail for what seems like no reason. The trick is figuring out whether a test that fails does so because it found a real problem in your app functionality, or because there was an issue with the test itself.

The best way to handle this problem is log your failing tests into a database and then analyze them. Even tests that fail intermittently with no apparent cause may turn out to have a pattern when you are able to analyze them in detail and as a larger data set. If this is beyond the scope of your testing setup, the next best strategy is to log your failing cases into a log file that records the browser, version, and operating system for those tests, and then retry those tests. If they continue to fail after a second or third retry, chances are that the issue is with the functionality you're testing, rather than the test itself. This isn't a total solution for dealing with flakes, but it should help you get closer to the source of the problem.

Keep Functional Tests Separate from Performance Tests

It's very important to maintain a distinction between functional tests and performance tests to make sure that you understand what your test results should be, and not have performance tests fail because they not have been designed with system capabilities and the desired outputs in mind. You should use Sauce Labs exclusively for functional testing, and for being able to integrate your functional testing into your continuous delivery pipeline.

- Functional tests should, as the name indicates, test some functionality or feature of your application. The output of these tests should be

either a simple "pass" or "fail" - either your functionality worked as expected, or it didn't.

- Performance tests, in contrast, should gauge and output performance metrics. For example, can your application server handle a particular load, and does it behave as expected when you push it to its limit? These types of tests are better undertaken with a testing infrastructure that has been specifically developed for performance testing, so all baseline performance metrics are well established and understood before you start the test.

By maintaining the distinction between functional and performance tests, and the different outputs that you expect from them, you should be able to more precisely design your tests to uncover the specific kinds of issues that you need to address to make your app more robust under any conditions.

Use Build IDs, Tags, and Names to Identify Your Tests

When you set the desired capabilities for your test, you can also add identifying information such as a name, tags, and build numbers that you can then use to filter results in your test results or Web Archive page, or to identify builds within your continuous integration pipeline.

Build, Tags, Name Example for Java

```
DesiredCapabilities caps = DesiredCapabilities.firefox();
caps.setCapability("platform", "Windows XP");
caps.setCapability("version", "37.0");
caps.setCapability("name", "Web Driver demo Test");
caps.setCapability("tags", "Tag1");
caps.setCapability("build", "build-1234");
WebDriver driver = new RemoteWebDriver(
new
URL("http://YOUR_USERNAME:YOUR_ACCESS_KEY@ondemand.saucelabs.com:80/wd/hub
"),
    caps);
```

Python Example Build

```
desired_cap = {
    'platform': "Mac OS X 10.9",
    'browserName': "chrome",
    'version': "31",
    'build': "build-1234",
}
```

Python Example Tags

```
desired_cap = {
    'platform': "Mac OS X 10.9",
    'browserName': "chrome",
    'version': "31",
    'build': "build-1234",
    'tags': [ "tag1", "tag2", "tag3" ]
```

Use Environment Variables for Authentication Credentials

As a best practice, we recommend setting your Sauce Labs authentication credentials as environment variables that can be referenced from

within your tests, as shown in this code example. This provides an extra layer of security for your tests, and also enables other members of your development and testing team to write tests that will authenticate against a single account.

Setting Environment Variables for Mac OS X/Linux

1. In Terminal mode, enter `vi ~/.bash_profile`, and then press **Enter**.
2. Press **i** to insert text into your profile file.
3. Enter these lines:

```
export SAUCE_USERNAME="your Sauce username"
export SAUCE_ACCESS_KEY="your sauce access key"
```

4. Press **Escape**.
5. Hold **Shift** and press **Z** twice (z z) to save your file and quit `vi`.
6. In the terminal, enter `source ~/.bash_profile`.

Setting Environment Variables for Authentication Credentials on Windows

1. Click **Start** on the task bar.
2. For **Search programs and fields**, enter `Environment Variables`.
3. Click **Edit the environment variables**.
   This will open the **System Properties** dialog.
4. Click **Environment Variables**.
   This will open the **Environment Variables** dialog.
5. In the **System variables** section, click **New**.
   This will open the **New System Variable** dialog.
6. For **Variable name**, enter `SAUCE_USERNAME`.
7. For **Variable value**, enter your Sauce username.
8. Click OK.
9. Repeat 4 - 8 to set up the `SAUCE_ACCESS_KEY`.

> **Finding Your Username and Access Key**
> You can find your Sauce Labs username and access key in the **User Profile > User Settings** section of your Sauce Labs dashboard. You should also check out our topic on setting your username and access key as environment variables.

Java Example of Using Environment Variables for Authentication

```
public static final String USERNAME = System.getenv("SAUCE_USERNAME");
public static final String ACCESS_KEY =
System.getenv("SAUCE_ACCESS_KEY");
public static final String URL = "http://" + USERNAME + ":" + ACCESS_KEY
+ "@ondemand.saucelabs.com:80/wd/hub";
```

PHP Example of Using Environment Variables for Authentication

```
define('SAUCE_HOST',
SAUCE_USERNAME.':'.SAUCE_ACCESS_KEY.'@ondemand.saucelabs.com');
```

Python Example of Using Environment Variables for Authentication

```
driver = webdriver.Remote(

command_executor='http://YOUR_USERNAME:YOUR_ACCESS_KEY@ondemand.saucelab
s.com:80/wd/hub',
   desired_capabilities=desired_cap)
```

Ruby Example of Using Environment Variables for Authentication

```
username = ENV["SAUCE_USERNAME"]
access_key = ENV["SAUCE_ACCESS_KEY"]
remote_server_url =
"http://#{username}:#{access_key}@ondemand.saucelabs.com:80/wd/hub"
```

C# Example of Using Environment Variables for Authentication

```
using System; //Importing the System library

internal readonly static string SAUCE_LABS_ACCOUNT_NAME =
System.Environment.GetEnvironmentVariable("SAUCE_USERNAME");
internal readonly static string SAUCE_LABS_ACCOUNT_KEY =
System.Environment.GetEnvironmentVariable("SAUCE_ACCESS_KEY");
```

Use Explicit Waits

There are many situations in which your test script may run ahead of the website or application you're testing, resulting in timeouts and a failing test. For example, you may have a dynamic content element that, after a user clicks on it, a loading appears for five seconds. If your script isn't written in such a way as to account for that five second load time, it may fail because the next interactive element isn't available yet.

The general advice from the Selenium community on how to handle this is to use explicit waits. While you could also use implicit waits, an implicit wait only waits for the appearance of certain elements on the page, while an explicit wait can be set to wait for broader conditions. Selenium guru Dave Haeffner provides an excellent example of why you should use explicit waits on his Elemental Selenium blog.

These code samples, from the SeleniumHQ documentation on explicit and implicit waits, shows how you would use an explicit wait. In their words, this sample shows how you would use an explicit wait that "waits up to 10 seconds before throwing a `TimeoutException`, or, if it finds the element, will return it in 0 - 10 seconds. `WebDriverWait` by default calls the `ExpectedCondition` every 500 milliseconds until it returns successfully. A successful return for `ExpectedCondition` type is Boolean return `true`, or a not null return value for all other `ExpectedCondition` types."

Python Example of an Explicit Wait from SeleniumHQ.org

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait # available
since 2.4.0
from selenium.webdriver.support import expected_conditions as EC #
available since 2.26.0

ff = webdriver.Firefox()
ff.get("http://somedomain/url_that_delays_loading")
try:
    element = WebDriverWait(ff,
10).until(EC.presence_of_element_located((By.ID, "myDynamicElement")))
finally:
    ff.quit()
```

Java Example of an Explicit Wait from SeleniumHQ.org

```
WebDriver driver = new FirefoxDriver();
driver.get("http://somedomain/url_that_delays_loading");
WebElement myDynamicElement = (new WebDriverWait(driver, 10))

.until(ExpectedConditions.presenceOfElementLocated(By.id("myDynamicEleme
nt")));
```

C# Example of an Explicit Wait from SeleniumHQ.org

```
IWebDriver driver = new FirefoxDriver();
driver.Url = "http://somedomain/url_that_delays_loading";
WebDriverWait wait = new WebDriverWait(driver,
TimeSpan.FromSeconds(10));
IWebElement myDynamicElement = wait.Until<IWebElement>((d) =>
    {
        return d.FindElement(By.Id("someDynamicElement"));
    });
```

Ruby Example of an Explicit Wait from SeleniumHQ.org

```
require 'selenium-webdriver'

driver = Selenium::WebDriver.for :firefox
driver.get "http://somedomain/url_that_delays_loading"

wait = Selenium::WebDriver::Wait.new(:timeout => 10) # seconds
begin
  element = wait.until { driver.find_element(:id =>
"some-dynamic-element") }
ensure
  driver.quit
end
```

Use the Latest Version of Selenium Client Bindings

The Selenium Project is always working to improve the functionality and performance of its client drivers for supported languages like Java, C#, Ruby, Python, and JavaScript, so you should always be using the latest version of the driver for your particular scripting language. You can find these on the Downloads page of the SeleniumHQ website, under **Selenium Client & WebDriver Language Bindings**.

Use Small, Atomic, Autonomous Tests

When a test fails, the most important thing is knowing what went wrong so you can easily come up with a fix. The best way to know what went wrong is to keep three words in mind when designing your tests: Small, Atomic, and Autonomous.

Small

Small refers to the idea that your tests should be short and succinct. If you have a test suite of 100 tests running concurrently on 100 VMs, then the time it will take to run the entire suite will be determined by the longest/slowest test case. Keeping your tests small ensures that your suite will run efficiently and provide you with results faster.

Atomic

An atomic test is one that focuses on testing a single feature, and which makes clear exactly what it is that you're testing. If the test fails, then you should also have a very clear idea of what needs to be fixed.

Autonomous

An autonomous test is one that runs completely independently of other tests, and is not dependent on the results of one test to run successfully. In addition, an autonomous test should use its own data to test against, and not create potential conflicts with other tests over the same data.

Use Page Objects to Model Repeated Interactions and Elements

Within your applications and sites there are elements that your tests interact with, sometimes on a repeating basis. Rather than having to repeatedly code these interactions into your test, you can use page objects to abstract these interactions into a single functional unit. For example, your tests may require logging into a a site or application. Rather than coding all these interactions into your test, you can create a LoginPage object that contains these interactions, which you then refer to in your test. This means taking a sort of object-oriented approach to test construction that enables you to simplify your test code and reduce duplication of effort. For more information, check out these references.

- The SeleniumHQ documentation of page objects, hosted on Google Code
- The documentation for the Intern testing framework provides a good explanation of page objects and an example in JavaScript
- The cheezy/page-object GitHub repository includes the page-object gem for Ruby, as well as a good tutorial on how to create page objects