Best Practices for Running Tests

From runnings tests in parallel to how to use small, atomic, autonomous tests check out this comprehensive guide on 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

```
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 `driver.findElement(By.id("someId")); or driver.findElement(By.name("someName"));` 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 to 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 of your web applications, front-end web performance, and tests of your network and servers under load.

- Functional tests should, as the name indicates, test some functionality or feature of your application. The output of these tests should generally be a simple "pass" or "fail" - either your functionality worked as expected, or it didn't. While running functional tests, it can also be advantageous to run front end performance tests that can help identify any regressions in JavaScript logic executed in the browser. When you use Sauce Labs for functional testing, you can also use custom extensions for WebDriver that will allow you test the performance of your website under specific network conditions, and also collect network and application-related metrics.

- Load tests, in contrast, should gauge and output network and server 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 load testing, so all baseline performance metrics are well established and understood before you start the test.

By maintaining the distinction between functional, front end performance tests, and load 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 application more robust under any conditions.

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

By assigning unique attributes (e.g., test name, tags, and build ID) in your test capabilities, you can then apply these annotations to filter results on your Sauce Labs **Test Results** and **Archive** pages. Although not required, following this best practice can make it easier to monitor tests and builds in your CI pipeline.

You can set these capabilities to be any combination of letters and numbers. To differentiate between builds, it's also a good practice to add a timestamp or CI job/build number at the end of your build tag.

See the following sections for more information:

- Code Examples: Build, Tags, and Name
- Video: Organize Tests by Build
- More Information

Please note: the **`build`** name and **`tags`** capabilities are not supported in automated real device testing at this time, please check back for future updates with regards to this functionality.

Warning

While it's technically possible to use the same build name for multiple test runs, this will cause all of your test results to appear incorrectly as part of a single run. This, in turn, will cause your test results for those builds to be inaccurate.

Code Examples: Build, Tags, and Name

Video: Organize Tests by Build

This video shows you how you can associate your tests with Builds in Sauce Labs, making it easier to understand how your tests are performing within your CI pipeline.

More Information

- W3C Capabilities Support
- Test Configuration Options

Use Environment Variables for Authentication Credentials

As a best practice, we recommend setting your Sauce Labs authentication credentials as environment variables on your local system, that can then be referenced from within your tests. 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.

What You'll Need

- The SAUCE_USERNAME and SAUCE_ACCESS_KEY specific to your Sauce Labs account. You can find them by logging into saucelabs.com and going to **Account > User Settings**.

Setting Up Environment Variables on Mac OSX/Linux Systems

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 Up Environment Variables on Windows Systems

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 **User 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`.

Referencing Environment Variables in Test Scripts

Once you've set up the environment variables for your credentials, you need to reference them within the test scripts that you want to run on Sauce. You can find examples of test scripts that use environment variables for authentication in the **demo** directory for each language in the Sauce Labs Training repo on GitHub.

Below are examples of how to set environment variables in a given language/framework:

Use Explicit Waits

As a best practice, use explicit waits on Selenium when you run into timeouts and failing tests.

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. Whether you use explicit or implicit waits, you should not mix the two types in the same test.

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."

```
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()
```

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

```
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"));
    });
```

```
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

Use Breakpoints to Diagnose Flaky Tests

A "flaky" test is one that normally runs without any issues, but every so often, maybe one in a hundred times, fails for seemingly random reasons. Debugging the issue with flaky tests is challenging, but by using breakpoints in flaky tests, you can identify the specific conditions that led to the flake, and dig into the problem more deeply using your developer tools. There are two ways to use breakpoints with your Sauce Labs test: through the `sauce: break` Selenium command, and by using the **Pause** button on the **Test Details** page while the test is running.

`sauce: break`

`sauce: break` is a JavaScript statement that you can insert into your Selenium `execute_script` command to both identify and interrupt flaky tests for further diagnosis. You can find more information in the topics Annotating Tests with Selenium's JavaScript Executor and Live Testing on Virtual Mobile Devices.

The Pause Button

When your test is running, you can use the **Pause** button on the **Test Details** page to interrupt the test and assume manual control of the browser. When you're done investigating, click **Stop**, and the test that you breakpointed will be marked as such on the Test Details page.

Use New Accounts for Each Test

Reusing your test accounts is an antipattern

Reusing an account between test runs can lead to:

- Problems with account state before testing starts
- Failures when account setup code has changed
- Failures that only show on other accounts (like your production customers)
- Parallelisation problems between tests using the same account

When should you create a new account?

Roughly speaking, whenever your tests aren't interacting with previous or future tests:

- Running a test on different platforms
- Running a test that doesn't depend on other test state (tests with shared state are **also** an antipattern)
- Every time you run a test suite

Avoid Leakage of Credentials

Sauce Labs test logs are securely stored, protecting them from external access. However, there are still some groups that can see test logs, including Sauce Support, your parent account, and other accounts in your company (depending on test privacy settings).

Solution - Don't use real credentials

The best way to avoid this is to avoid using "real" credentials in tests, through the creation of temporary accounts.

Workaround - Transmit session tokens only

You can also avoid sensitive credentials using Selenium's ability to extract and inject cookies into accounts:

1. Create a session in your environment, either directly in the application engine, or by using a local Selenium session or headless browser.
2. Extract the session tokens (local storage objects, credentials, cookies, etc.).
3. Use Selenium to push these objects and tokens into the browser under Sauce Labs' control

This technique avoids sending plain text passwords, however, the sent tokens and cookies are still logged. If your session tokens are not time-sensitive, this provides only security through obscurity. We recommend using time-sensitive session tokens.

Workaround - Change passwords after tests

If generating tokens and using unique temporary accounts is not possible, we recommend you have test actions your suite always takes, in order to change to a new, randomly generated password.

After each test, use a locally automated browser, a direct connection to your application database or a headless browser to change your test account's password to a new, randomly generated password. Ensure this password is stored in your CI environment, a credential store, or some other method.

In order to prevent credential loss from blocking test suites, you may want to start each test suite by changing the password, again, either by using a headless browser or local Selenium session to perform your password recovery process, or by directly interacting with your application's database.

Be Aware of the Load on Your Servers

As you move into fully automated testing and builds, with tests running in parallel and against multiple device/browser/platform/operating system combinations, be aware of the load that this will place on both your CI/CD server and the site under test. A best practice in this situation is make sure that both the CI/CD server and the site under test are on machines that are not running additional processes or open to additional network traffic, and can handle the additional number of simultaneous jobs and tests.

Imperative v. Declarative Test Scenarios

Imperative v. declarative test scenarios is a concept that is often discussed in the context of Cucumber and Behavior Driven Development (BDD), but it is applicable to all languages and test runners. A great post by Aslak, the author of Cucumber, gives a great description of his intentions along with a number of additional links at the bottom for further reading.

- **Imperative testing** or programming is essentially spelling out with as much detail as necessary how to accomplish something.
- **Declarative testing** or programming is only specifying (or declaring) what needs to be accomplished.

This is seen acutely in BDD circles because the goal of BDD is to get all of the interested parties (Project, Dev, Test, Business, etc) to collaborate on the requirements of a feature before anyone begins working on the implementation. Many testers have latched on to BDD tools as glorified test runners rather than a way to actually facilitate BDD practices. This results in features that include actual code and data structures. Less problematic, but still usually missing the point, is a heavy reliance on imperative scenarios. For example:

1. Given I open a browser
2. And I navigate to `http://example.com/login`
3. When I type in the username field `bob97`
4. And I type in the password field `F1d0`
5. And I click on **Submit** button
6. Then I should see the message `Welcome Back Bob`

This scenario is not focused solely on the business requirements, and actually needs to have knowledge implementation specific details in order to work. The fact that the user's username is `bob97` has nothing to do with the business requirements of the company. If BDD features are designed to represent the business logic, then they should only be changed if the business requirements change. If `bob97` changed his password to `1<3MyD0g`, the page location changes ,or the success message changes, this test would fail, even though the business needs are exactly the same.

A declarative example of the same functionality looks like this:

1. Given I am on the Login Page
2. When I sign in with correct credentials
3. Then I should see a welcome message

This is all information that the business cares about, is easier to read, and leaves it to the implementation to specify how a successful login is accomplished.

This principle can be applied to any language or test runner. Tests should largely focus on what needs to be accomplished, not the details of how it is done. They should mostly be understandable when read by non-developers. This approach goes very well with using the Page Object Pattern. Keep the business logic in the test, and put all of the information about the drivers, the element locators, the timing, etc in the Page Object.

Use Maven to Manage Project Dependencies

Do you use use Maven as the dependency manager for your projects? If so, this article will give you some insight into how it works, along with some useful commands. You will find these especially helpful if you are using a private repository as the main repo to source the dependencies for your project. By using the dependency list generation methods described in this article, you can greatly reduce the workload needed to make dependencies available in your private repository, or locally if you intend to run your project offline.

How Does Maven Manage Dependencies?

You add dependencies for your project to your Maven configuration file (also known as the pom.xml file, for Project Object Model). As you build your project using Maven, it resolves these dependencies and downloads the dependencies to your local repository folder. This folder is usually located in your user's home folder and is named .m2. Each dependency downloaded from the repository is a project itself, and has its own dependencies. Maven recursively resolves all of these dependencies for you, and then merges shared dependencies and downloads them. At the end of the process you end up with a list of dependencies that are needed to run your project on your local machine. For full details about how this process works, check out the Maven documentation.

How Do I Get the Dependencies for a Project?

From this brief description of how Maven dependencies work, you may notice a problem: How do you know exactly what dependencies are required for a project, and if you don't have Internet access or are trying to run your project offline, how do you make sure you have all the dependencies you need available locally? Fortunately, Maven includes several commands that you can use to make sure you have all the dependencies and repositories set up so that your project will build and run with no errors.

1. First, check for version updates to your dependencies, and then update the outdated dependencies in your pom.xml file as necessary.
   `$ mvn versions:display-dependency-updates`
2. Get a list of your repositories, and make sure they're pointing to all the correct dependencies.
   `$ mvn dependency:list-repositories`
3. Get a list of your plugin and project dependencies, and make sure they're all available in your private or local repository.
   `$ mvn dependency:go-offline`

If you want to automate the process, or just get a cleaner output of your dependencies, you can also use this bash command:

```
$ mvn -o dependency:go-offline|grep ":*.jar"|awk '{split($0,a,":");print a[2]}'
```