SAUCELABS

Selenium Jenkins - How to Do it Yourself and Sauce Labs Advantage

Posted **June 10, 2016** by Sauce Labs

Click here for the Sauce Labs Selenium Jenkins Plug-in technical docs.

So you're looking to integrate Selenium WebDriver (or RC) with Jenkins, in order to run full end-to-end / UI automation testing as part of your build process? Well, we have good news and bad news:

- **The good news** is that it's relatively easy to run Selenium tests as part of the Jenkins build, assuming you're already using a testing framework like JUnit or TestNG. We'll explain a few ways to achieve it further on this page.
- **The bad news** is that there are quite a few challenges to consider:
  - Builds with Selenium tests take much longer to run.
  - You'll find it hard to cover all the relevant browsers and platforms.
  - You might have a lot of maintenance work installing browsers/operating systems.
  - Scaling up your testing will be complex and you'll need to deploy Selenium Grid and/or rewrite your tests to support multi-threading.

We at Sauce Labs run over a million Selenium tests every day on our cloud testing platform. We have an easy-to-use Selenium Jenkins Plugin that allows you to run your Selenium tests, with effortless scalability and cool features. (Keep in mind Sauce is a commercial service, but we have a free trial and is free for open source projects.)

If you'd like to do Selenium with Jenkins yourself, using open source software, check out our summary table of Jenkins Selenium solutions, or visit the sections below to see how it's done and the pros and cons of each solution:

- Option 1: Selenium on one machine- headless testing - running Selenium directly on your Jenkins workstation with a headless browser, probably the easiest option.
- Option 2: Selenium on one machine - real browsers - running Selenium on your Jenkins workstation with real browsers, limited to the capacity of your workstation.
- Option 3: Selenium Grid local- real browsers on multiple machines - running Selenium on several machines on premises, using Selenium Grid, and activating them via Jenkins.
- Option 4: Sauce Labs Jenkins Plugin how to setup the plugin and run Selenium Jenkins tests on over 250 browser/OS combinations in the Sauce cloud, with no need to install browsers and set up special infrastructure in house.

Heads Up: Challenges You Might Face When Running Selenium Tests with Jenkins

It's not fun reading about problems. But it's even less fun running into them further along your project! So we hope to save you a bit of pain with our take on Selenium Jenkins integration challenges.

(This discussion is based on our experience with over 500 million Selenium tests run on the Sauce Labs cloud testing platform - many of them through CI platforms.)

CHALLENGE 1: SELENIUM TESTS TAKE A LONG TIME TO RUN

It's true that in a typical project, there can be thousands of unit tests but only dozens or hundreds of Selenium tests. Nevertheless, those Selenium tests will take much much longer to run than all your unit tests put together. Selenium tests run on a real browser, they need to perform actual browser operations and often wait for an HTTP server to respond. Plus starting and stopping the browser takes time - you get the picture.

The implication is that if you include Selenium tests as part of your build, the build will take much longer to run - so that if currently you're running a build on every commit, or several times a day, you may have to resort to running the build overnight, and you might need to upgrade your Jenkins workstation or even add more machines to your Jenkins cluster. There are two ways to achieve this:

- Setting up a local Selenium Grid.
- Running your tests on Sauce Labs with the Sauce Jenkins Plugin.

CHALLENGE 2: LIMITED BROWSER COVERAGE

If you choose to run your Selenium tests on real browsers (as opposed to headless browsers), you will actually need to install multiple browsers on your Jenkins workstation. You will only be able to run Selenium tests against browsers that are actually installed:

- If the workstation is Windows, it is difficult to test on OSx. If it's Mac, you can't test on Internet Explorer (except by running a Windows VM, which can be complex to do automatically).
- With a Windows machine, you won't can't test how Firefox behaves on Linux, how Chrome behaves on Mac, etc.
- There are only so many browsers you can install on one machine.
- Because tests take a long time, even if you have installed numerous browsers, you probably won't be able to run all your tests on all of them. That will make the build even longer.

You might be thinking:

- **No biggie, I'll do headless testing!** Sure, and we explain below how it's done. But keep in mind it's not a true test of the browsers used by your visitors. In the end someone has to test the application on actual Firefox, Chrome, etc.

- **I'll use Selenium Grid!** Also a good option, here's how to do it. This way you can run many more browsers and reduce the build time. But it can turn out to be quite complex.
- **Let's do multi threading!** Okay, but you might need to rewrite all your tests. See our series of blog posts explaining how to do multi-threaded testing with JUnit 4. This will definitely speed up the build but it will still take a long time compared to non-Selenium testing.

Or - you can use the Sauce Labs Jenkins plugin to run the tests seamlessly on the Sauce cloud. This will give you access to a comprehensive and growing list of over 700 browser/OS combinations.

CHALLENGE 3: UPDATING BROWSERS

If you go down the "real browser" route (as opposed to headless testing), you'll need to worry about updating your browsers. Each time a new version of IE / Firefox / Chrome / Safari comes out, you'll need to install it on your local machine/s. And uninstall old browsers that are no longer relevant. And deal with browsers that want to auto-update themselves, while you want to test specifically on an older version. All of this creates a big ongoing maintenance effort.

At Sauce Labs, a big part of our service is taking care of this maintenance effort for you. We can set up a cloud VM for you on the fly which has practically any combination of browser and operating system, and we work hard all the time to add the latest platforms.

CHALLENGE 4: DIFFICULT TO SCALE UP

It's important to consider how you'll scale up your testing in the future. If you add more Selenium tests - for example, to test more functionality or new features - this will increase the build time and the system resources required. Also, if you want to test on additional browsers/platforms, this will increase build time.

Keep in mind that your test running capacity will be limited by either:

- The capacity of your Jenkins workstation (unless you distribute builds with Jenkins slave nodes), or
- The number of machines available in your local Selenium Grid if you set one up.

One approach to solving this is headless testing, which runs much faster than real-browser testing. Another approach is our Sauce Labs Jenkins Selenium plugin. Sauce Labs does Selenium testing on the cloud for thousands of organizations. We have an easy-to-use Jenkins integration that lets you to run your Selenium tests on our cloud service, with effortless scalability and cool features.

4 Approaches to Integrating Jenkins with Selenium - Pros and Cons

SOLUTION	COMPLEXITY	PROS	CONS
1. Selenium on one machine - headless	Low	• Tests take	• Not testing real

browsers. Run the Selenium tests directly on your Jenkins workstation. Use headless browsers to save time and at least approximate GUI behavior.		• much less time to run (compared to real browsers) • No need to update browsers - no real browsers used • Easier to scale, less system resources used • Open source	• browsers and platforms • Still increases build time compared to unit testing
2. Selenium on one machine - real browsers. Run Selenium tests directly on your Jenkins workstation against real browsers. Build takes longer, but you'll be able to test real browser reactions.	Medium	• Testing real platforms • Open source	• Tests take long to run • Difficult to scale • Limited browser coverage - only what you have installed on Jenkins workstation • Need to install, update, maintain browsers
3. Selenium Grid. Run a battery of Selenium tests quickly and on a wide range of browser/OS combinations, by investing in infrastructure. Set up Selenium Grid to distribute test execution across multiple nodes.	High	• Tests take much less time to run (compared to Selenium on one machine) • Testing real platforms (unlike headless testing)	• Need to update browsers • Difficult to scale further - need to add more machines and configure them • High cost of hardware and ongoing maintenance

		• Ability to support more browsers and operating systems • Open source	• Browser coverage still limited to the number of computers / operating systems you can physically set up. There are more than 700 OS/browser combinations in use today.
4. Sauce Labs Jenkins plugin. Outsource the infrastructure to Sauce Labs, run tests quickly on over 800 browser/OS combinations, without disrupting your CI process and without installing and maintaining browsers locally.	Low	• Tests take much less time to run • Full browser coverage • Testing real platforms • Ability to scale effortlessly • Browsers are updated for you	• Commercial service - beyond the free plan limit, priced per minute. (Sauce is free for open source projects.)

Option 1: Selenium on one machine - headless testing

The simplest option is to run the Selenium tests directly on your Jenkins workstation. In such a configuration, running tests serially against full-fledged browsers is going to take a lot of time and computer power. One way to economize is to run them using headless ("GUI-less") browsers. You'll save time configuring and running these emulators, and at least approximate GUI behavior.

There are several ways you can set headless Selenium testing, depending on your browser needs and operating system. Some popular headless browsers are HTMLUnit, Xvfb, Spynner, Cabybara, Ghost, and PhantomJS. As an example, we'll show how to perform functional tests on Web applications using Selenium and PhantomJS.

PhantomJS runs a WebKit engine with full JavaScript access, only without the GUI. It supports practically all platforms. PhantomJS is suitable for automated testing as part of a CI system like Jenkins because it can be installed directly on the Jenkins node and executed from the command-line.

Here are the general steps:

1. Create a Maven pom.xml file in a new project directory that includes all the dependencies you need to get the project going.
2. From the command-line, run `mvn clean install` to download the dependencies.
3. Install PhantomJS from phantomjs.org/download.html. Make sure phantomjs is accessible from the command line.
4. If you have not done so yet, create a test.
5. Run the test. At the prompt, type: `mvn test` The test will run using PhantomJS and output the results to target/surefire-reports in a JUnit XML format that is understood by Jenkins.
6. Make sure PhantomJS is installed on all the Jenkins Nodes that will run the tests.
7. Create a new Jenkins job and set it to run after a build is deployed.
8. Save the job, run it and view the test results.

For more details, see this blog post from AppNeta.

Pros and cons of headless Selenium testing

Pros

- The main advantage of headless browser emulators is that tests run much faster.
- Headless browsers are easy to run and configure, you can run one on practically any platform.
- They are easy to automate as part of a continuous integration flow, because they run from the command line and do not execute a full browser GUI.

Cons

- The browser is abstract, does not simulate user actions on real platforms. Headless browsers can mask many problems that only real browsers encounter.
- Headless browsers have partial JavaScript/AJAX support. So, you cannot fully test web applications which include Javascript and AJAX elements.

Option 2: Selenium on one machine - real browsers

Let's say you want to run Selenium tests directly on your Jenkins workstation against real (non-"headless") browsers. Your build will take longer, as it runs full-blown browsers and full UI automation, but you'll be able to measure real browser reactions on the most common browsers used by site visitors.

Following is one method of achieving WebDriver integration with Jenkins, using Maven.

Here are the general steps:

1. Install, on the Jenkins workstation, all the browsers you want to test your application on (e.g. Chrome, Firefox, Internet Explorer).

2. Ensure Jenkins is up and running.

3. Ensure Java and Maven are installed.

4. Run Tests locally first (4 steps)

    1. Create a folder with this structure:

    ```
    ProjectName
         -- pom.xml
         -- src
         ----- test
         ----- java
         ----- YourTest.java
    ```

    2. Create a pom.xml file
    3. Put your Selenium scripts in Java folder: `[/ProjectName/src/test/java]`
    4. Open a terminal, and cd to `[/ProjectName]`, then run: `mvn test`

       You should see your tests running and the results once they are complete.

5. Now, integrate with Jenkins (9 steps):

    1. Create a "Build a free-style software project" Jenkins job.
    2. Configure newly created job.
    3. Reference your source code using any available version management system (Git, SVN, CVS or other).
    4. Run the Jenkins job against your local directory.
    5. Select 'Invoke Top-Level Maven-targets' from the Build dropdown box.
    6. Enter 'test' as Goals.
    7. Save the configuration.
    8. Run the job. You should expect a failure first time round.
    9. Now copy all you data under your project directory to the Jenkins workspace and run the job again. Now you can see tests running.

For more details, see this blog post from Anoop Philip.

Pros and cons of Selenium on one machine with real browsers

Pros

- **Testing real platforms** - in this method you can test at least a few of the most popular platforms used by your visitors.
- **Open source, no infrastructure costs** - Selenium is free and since the browsers will typically run on the existing Jenkins workstation, there is no additional investment in infrastructure.

Cons

- **Tests take long to run** - because Selenium tests involve starting up real browsers, waiting for one or more web pages to load, and performing actions on them, these tests take significantly longer to run than regular unit tests. If you were previously used to running builds on every commit, or several times a day, you will be able to run much less frequently, or possibly resort to a nightly build. It is still possible to run builds during the day, skipping the Selenium tests.

- **Difficult to scale** - if you want to scale up the number of tests, you might need to add more machines, and this will require using Selenium Grid see how to do this below.

- **Limited browser coverage** - this scenario involves testing on browsers physically installed on the local Jenkins workstation. There are only so many browsers you can install on one machine, and if, for example, you are running on Windows, you can't test Safari, nor can you test Internet Explorer on a Mac, without running virtual machines. So this scenario is inherently limited in the type and number of platforms you can test.

- **Need to install, update, maintain browsers** - you will probably install more than a few browsers on the local machine, and then, you need to install new versions of the browsers as they come up, uninstall old versions, and deal with browsers wanting to automatically update themselves, while you are interested in testing specifically the old version. This becomes a large, ongoing maintenance effort.

Option 3: Selenium Grid local - real browsers on multiple machines

If you need to run a battery of Selenium tests quickly and on a wide range of browser/OS combinations, you will need to invest in infrastructure in the form of additional machines, installed with different operating systems and browsers, to cover the additional scenarios. Then, set up Selenium Grid to distribute test execution across multiple nodes.

To get set up, you'll need:

- Standalone jar for selenium.
- Chrome driver

Both can be downloaded here: https://code.google.com/p/selenium/downloads/list

These are the general steps to get it to work:

1. Set up two or more computers on which you want to run Selenium tests with real browsers. This is your Selenium Grid cluster.

2. Create a directory that looks like this:

```
(dir) chrome-<ver_1>
(dir) chrome-<ver_2>
(dir) chrome-drive-zipfiles
```

```
(dir) jdk android-server
chromedriver
grid-startup-batch
selenium-server-standalone
```

3. Write a batch file which spawns a WebDriver node as well as the Selenium Grid hub (we'll call it grid-startup-batch.bat). Run the batch file on one of the Selenium Grid machines.

4. Open the Grid console, by firing up a web browser of your choice and type: `http://localhost:4444/grid/console`

5. Inspect the grid console. Move your mouse over the browser icons to check browser names, number of instances, and node types.

6. Basing yourself on information shown in the mouse-over, try to connect to an instance (let's say of Internet Explorer) that is being provided by your Selenium Grid. Create a new instance of a DesiredCapabilities object, setting the browser you are testing on, for example:`capabilities.setBrowserName(DesiredCapabilities.internetExplorer().getBrowserName());`

7. In order to run tests such that your execution is routed to a specific browser running on a specific platform, you need to use a Node Configuration file when starting your WebDriver node. To create the Node Configuration file, copy the example here and modify it according to your needs.

8. Modify the grid-startup-batch.bat file (from earlier) and add the argument `-nodeConfig webconfig.txt`, where`webconfig.txt` is your configuration file, available in the current directory where you created your grid-startup-batch.bat file.

9. Now test against a different version of a browser in your grid, say Firefox 8, by setting the browser version also in your DesiredCapabilities object, like this: `capabilities.setVersion("8");` and modifying the webconfig.txt file, adding a version parameter to it, like this: `"version":"8"`

10. Restart the grid to rerun your test.

11. To connect Jenkins to the Selenium Grid and run it as part of your build, use the Jenkins Selenium Plugin (actually a Selenium Grid Plugin), here.

For more details, see this blog post from Krishnan Mahadevan.

Pros and cons of a local Selenium Grid running real browsers

Pros

- **Tests take much less time to run (compared to Selenium on one machine)** - because they can run simultaneously, reducing build time significantly.
- **Testing real platforms (unlike headless testing)** - you now test against real browsers, and if you add enough machine to your grid, the tests will run fast enough to allow frequent builds.

- **Ability to support more browsers and operating systems** - with an extended infrastructure of more machines, you can run your tests against more browser/OS platforms.
- **Open source** - Selenium Grid is free software, and since you own the infrastructure, you don't need to pay per minute of tests run (unlike commercial Selenium services).

Cons

- **Complex initial setup** - getting Selenium Grid to work is non-trivial, like any clustered solution.
- **Need to update browsers** - to keep up with the ongoing release of new browser versions, you must continually add and update the browsers you have installed in your cluster.
- **Difficult to scale further** - to broaden your testing ability, you need to add more machines and configure them.
- **High cost of hardware and ongoing maintenance** - purchasing additional machines, and possibly having to hire personnel to maintain, configure, and update, has its price.
- **Browser coverage still limited** - you can only test on as many OS/browser combinations as can be supported by the number of machines in your cluster. There are more than 700 OS/browser combinations in use today.

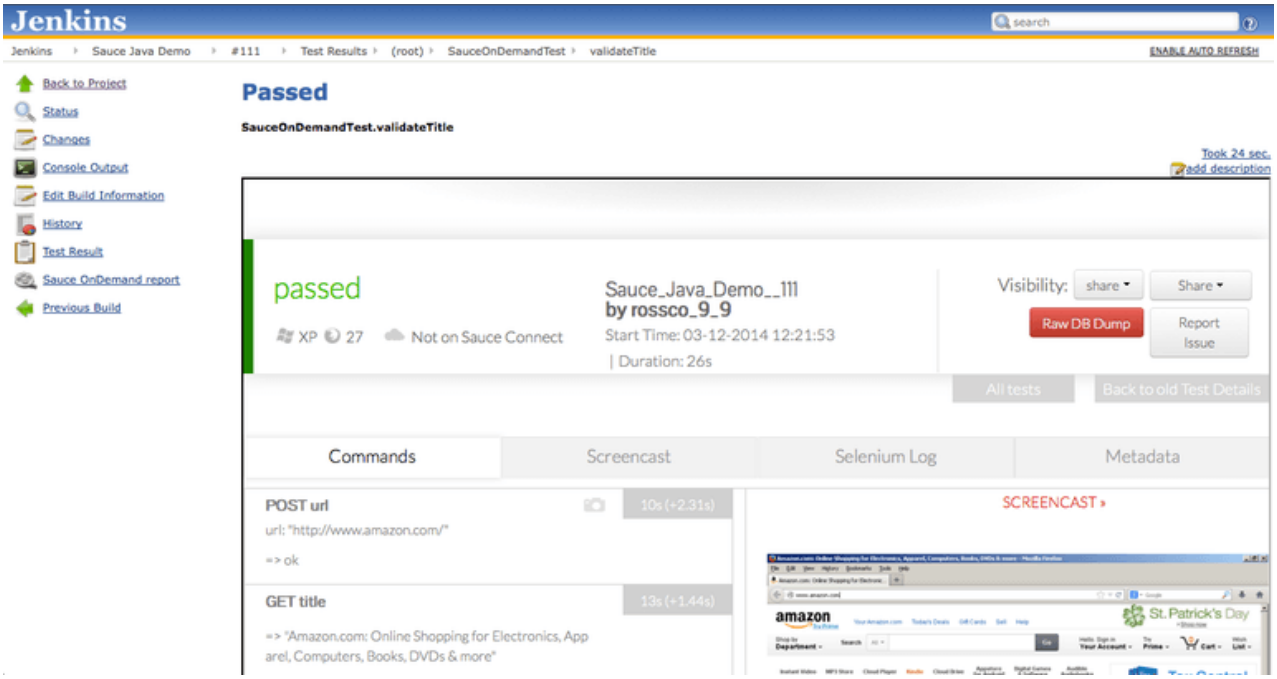
Option 4: Sauce Labs Jenkins Plugin - Selenium testing with no maintenance and effortless scalability

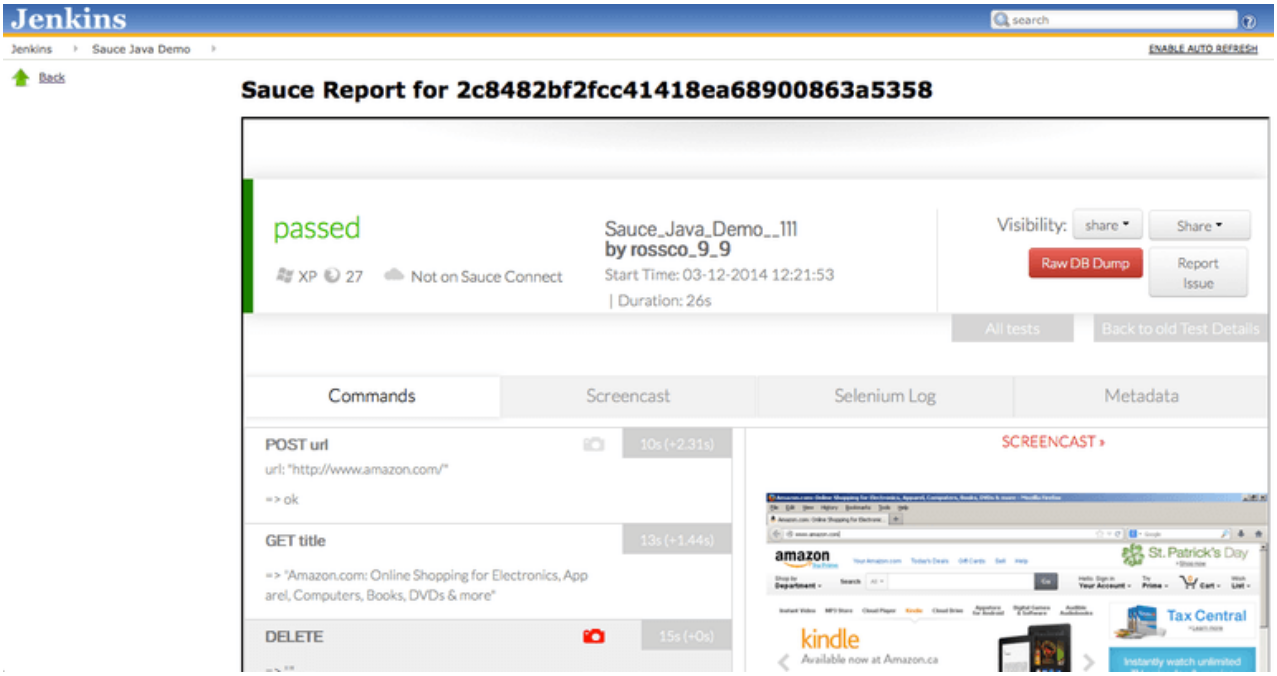
Let's say you want Jenkins to run Selenium tests automatically several times a day, or even each time there is a code commit, across a wide range of browser/OS platforms. Because Selenium tests are resource-hogs, this requires a lot of computing power and ability to run tests in parallel. It's complex to set up the entire testing cluster yourself, so an alternative is to outsource the test infrastructure to Sauce Labs.

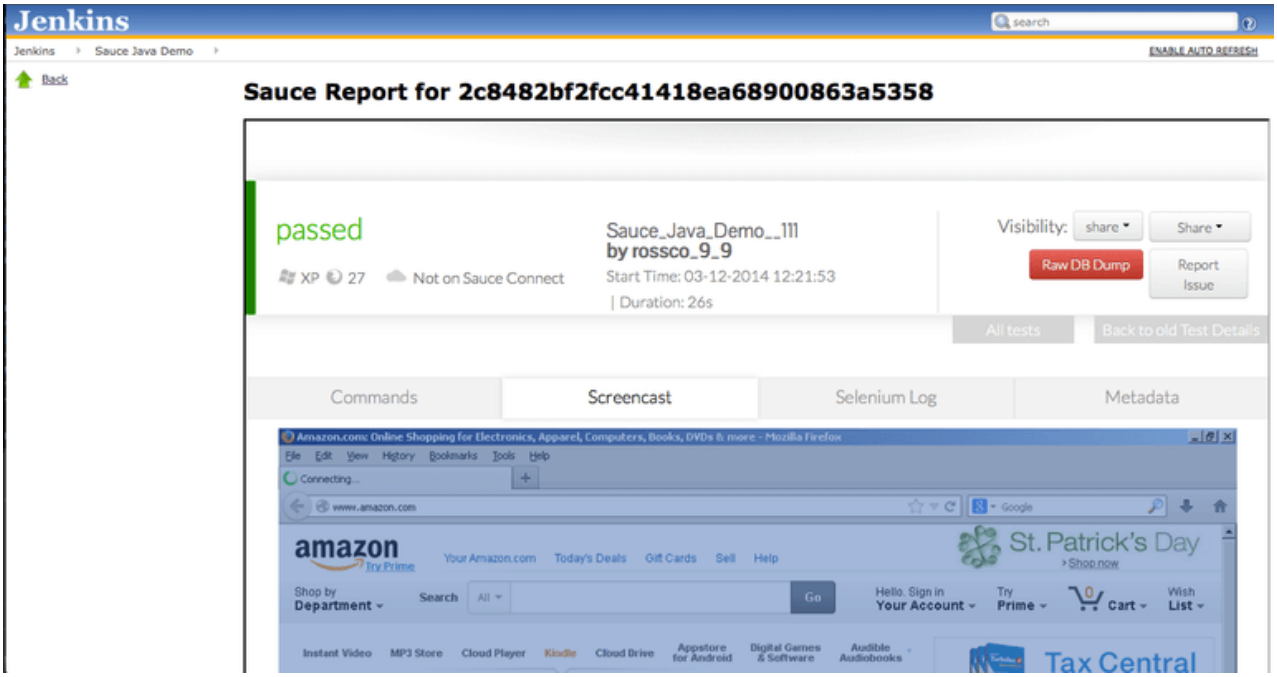
Sauce Labs' cloud testing platform has run over 90 million Selenium tests for Yelp, Firefox, the BBC, and thousands of other organizations. Our cloud can run your Selenium tests quickly, on as many platforms as required, without disrupting your CI process, and without requiring you to install and maintain browsers.

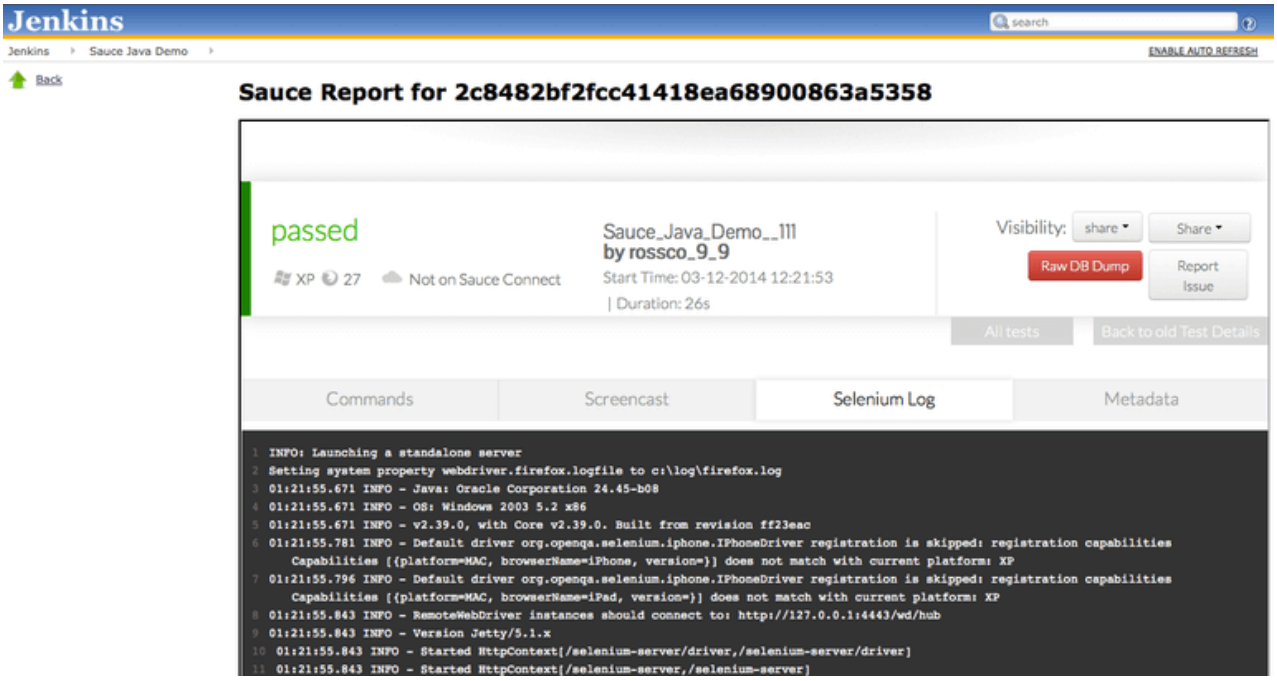
Sauce Labs offers an easy-to-use Jenkins plugin, which allows you to:

- Run Selenium tests as part of your build,
- Have them execute quickly on the Sauce cloud, on more than 700 browser/OS combinations,
- Feed the results back to Jenkins including screenshots and even video screencasts showing what happened on the browser in each test. You can view these directly from the Jenkins interface.


Jenkins
search
Jenkins › Sauce Java Demo › #111 › Test Results › (root) › SauceOnDemandTest › validateTitle
ENABLE AUTO REFRESH
Back to Project
Status
Changes
Console Output
Edit Build Information
History
Test Result
Sauce OnDemand report
Previous Build
Passed
SauceOnDemandTest.validateTitle
Took 24 sec.
add description
passed
XP 27
Not on Sauce Connect
Sauce_Java_Demo__111
by rossco_9_9
Start Time: 03-12-2014 12:21:53
| Duration: 26s
Visibility: share
Share
Raw DB Dump
Report Issue
All tests
Back to old Test Details
Commands
Screencast
Selenium Log
Metadata
POST url
10s (+2.31s)
url: "http://www.amazon.com/"
=> ok
GET title
13s (+1.44s)
=> "Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more"
SCREENCAST »


Jenkins
search
Jenkins › Sauce Java Demo ›
ENABLE AUTO REFRESH
Back
Sauce Report for 2c8482bf2fcc41418ea68900863a5358
passed
XP 27
Not on Sauce Connect
Sauce_Java_Demo__111
by rossco_9_9
Start Time: 03-12-2014 12:21:53
| Duration: 26s
Visibility: share
Share
Raw DB Dump
Report Issue
All tests
Back to old Test Details
Commands
Screencast
Selenium Log
Metadata
POST url
10s (+2.31s)
url: "http://www.amazon.com/"
=> ok
GET title
13s (+1.44s)
=> "Amazon.com: Online Shopping for Electronics, Apparel, Computers, Books, DVDs & more"
DELETE
15s (+0s)
SCREENCAST »





What about testing applications running behind your firewall?

When integrating Jenkins with Sauce Labs, it's common that the system under test is running locally behind a firewall. Because Sauce is a remote cloud service, it needs to have secure access to your application. For this purpose we've developed Sauce Connect, a secure tunnel between your web server and the Sauce cloud. It lets you access your data and run your tests on Sauce from behind your firewall, while keeping everything encrypted.

Setting up the Sauce Labs Jenkins Plugin

Before you begin, make sure you have a Sauce Labs user account. You'll need to specify your username and API access key during the setup. Sign up for the free plan.

These are the general steps for setting up the Sauce Labs Jenkins Plugin:

1. In the Jenkins Administration page, select **Manage Jenkins** > **Manage Plugins** > **Available.**
2. Download the Sauce Labs Jenkins plugin. Ensure you select the **Restart Jenkins when installation is complete and no jobs are running** option. See more details on Installing the Sauce Labs Jenkins plugin.

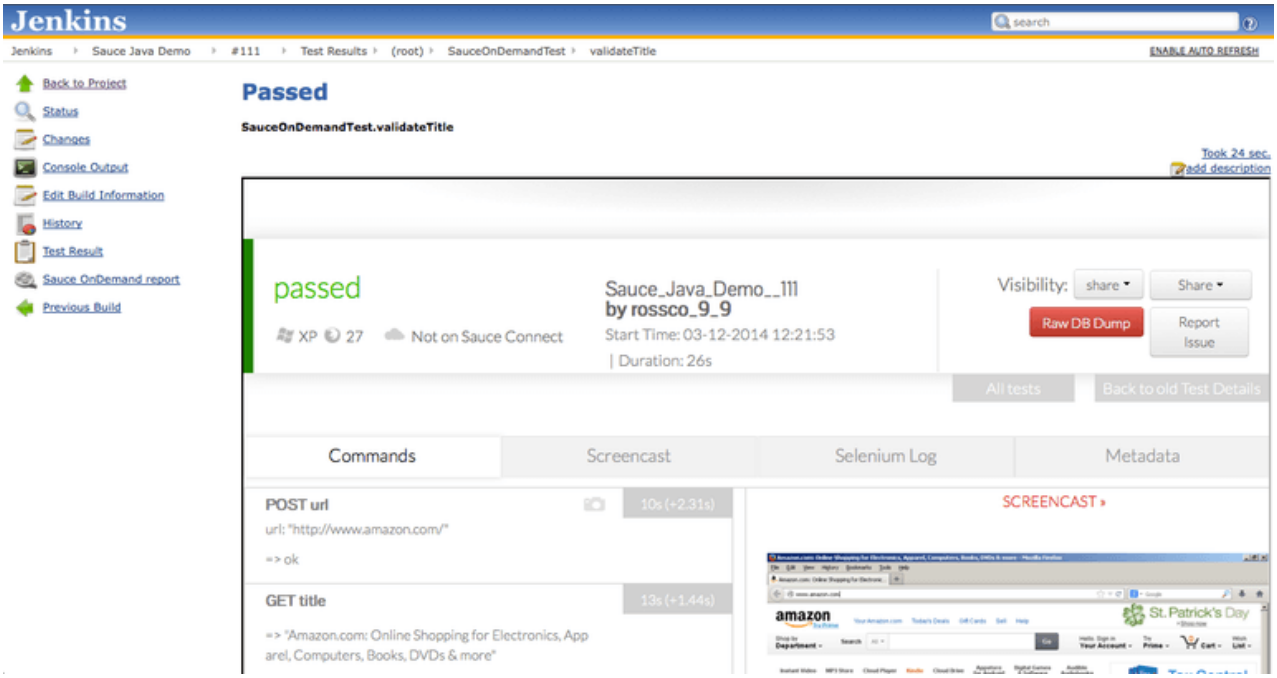
3. Authenticate your Sauce Labs user account. In the Jenkins Administration page, select **Manage Jenkins** > **Configure System**. In the **Sauce OnDemand** section, provide your username and API access key. There are additional authentication options in the Sauce Jenkins plugin configuration.

4. Click **Test Connection** to complete authentication.

5. Configure a Jenkins Job. The most critical steps are:

    - Updating tests to reference the environment variables set by the plugin.
    - Outputting the Sauce session ID to the stdout to allow the Sauce plugin to associate test results to Sauce Jobs.
6. Now run the tests by clicking **Build Now** in the Jenkins Job navigation pane.There are additional options for post build activities that can enrich the data received from the Selenium tests to Jenkins.

7. Inspect build results: navigate to the Jenkins **Build Summary** page, and click the link to a build.



You're done! You can now run Selenium tests as part of your Jenkins build, and rely on the Sauce cloud to execute the tests on as many browsers and operating systems as needed.