Getting Started With Selenium

WRITTEN BY DAVE HAEFFNER
AUTHOR OF ELEMENTAL SELENIUM

UPDATED BY MARCUS MERRELL
DIRECTOR OF TECHNICAL SERVICES, SAUCE LABS

CONTENTS

WHAT IS SELENIUM?

Selenium is a free and open-source browser automation library used by millions of people for testing purposes and to automate repetitive web-based administrative tasks.

It has the support of the largest browser vendors, who have integrated Selenium into the browsers themselves. It is also the core technology in countless other automation tools, APIs, and frameworks used to automate application testing.

Selenium/WebDriver is now a W3C (World Wide Web Consortium) Recommendation, which means that Selenium is the official standard for automating web browsers.

GETTING STARTED

Selenium language bindings allow you to write tests and communicate with browsers in multiple languages:

- Java
- JavaScript
- Python
- Ruby
- C#

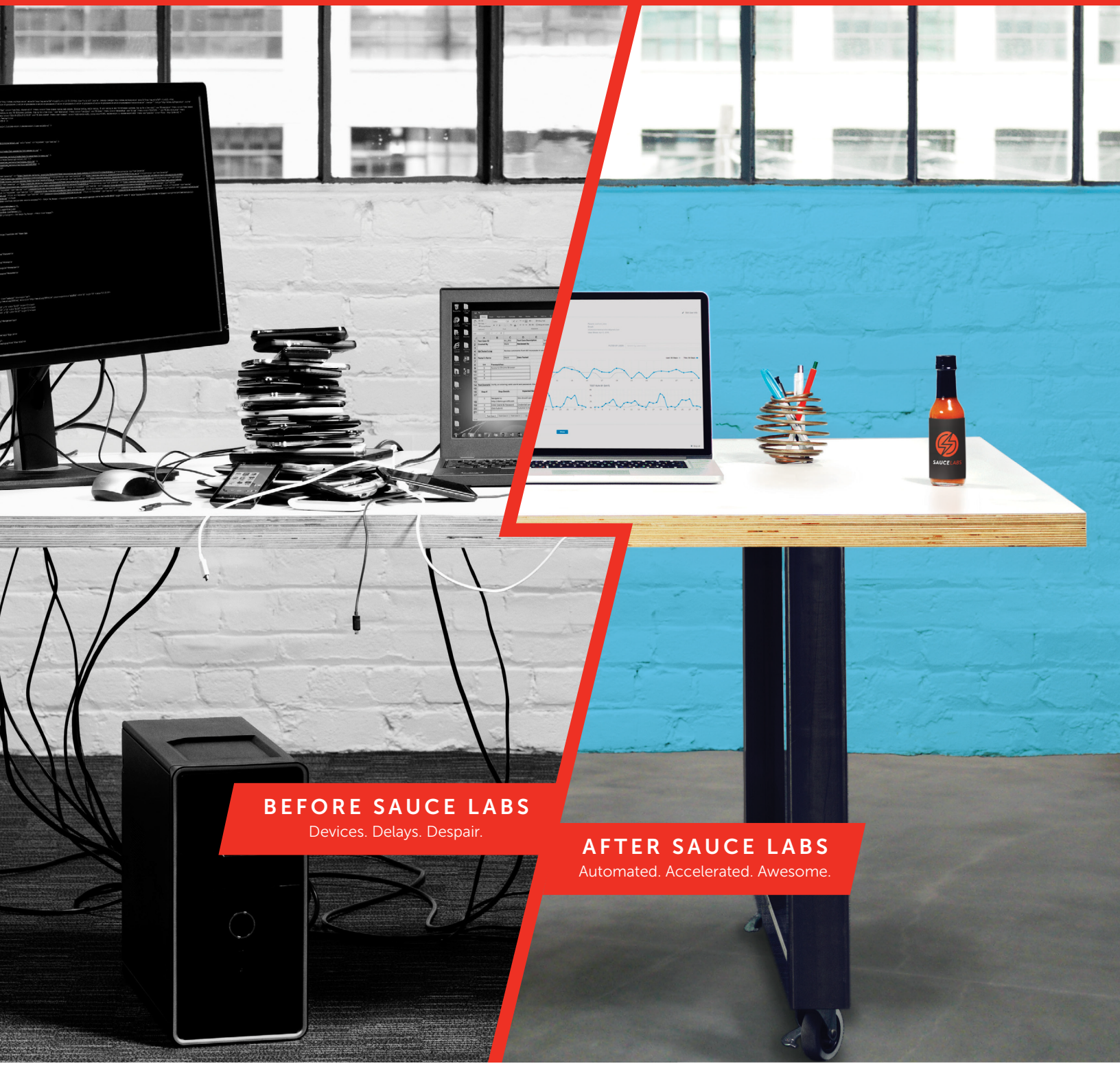
In order to start writing tests, you first need to install the bindings for your preferred programming language.

JAVA (WITH MAVEN)

In your test project, add the following to your `pom.xml`. Once done, you can either let your IDE (Integrated Development Environment) use Maven to import the dependencies or open a command-prompt, `cd`, into the project directory, and run `mvn clean test-compile`.

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
```

A brief history of web and mobile app testing.



Start your free trial today

saucelabs.com/sign-up

Email sales@saucelabs.com or call (855) 677-0011 to learn more.

SAUCELABS

Continuous Confidence

You will need the Java Development Kit (version 8+ for 3.x and 4.x versions of Selenium) and Maven installed on your machine. For more details on Selenium Java bindings, see the API documentation.

JAVASCRIPT (NPM)

JavaScript offers two different approaches for incorporating Selenium/WebDriver into your tests:

TRADITIONAL JAVASCRIPT BINDINGS

Use the following command into a command-prompt to install the JavaScript bindings for Selenium:

```
npm install selenium-webdriver
```

You will need Node.js and NPM installed on your machine. For more information about the Selenium JavaScript bindings, check out the API documentation.

WEBDRIVER.IO

WebDriver.IO is a "next-gen" test framework for getting started with WebDriver in JavaScript. It's a fully-featured, W3C-compliant test framework, available with full documentation at webdriver.io.

PYTHON

Use the command below to install the Python bindings for Selenium:

```
pip install selenium
```

You need to install Python, pip, and setuptools in order for this to work properly. For more information on the Selenium Python bindings, check out the API documentation.

RUBY

Use the following command to install the Selenium Ruby bindings:

```
gem install selenium-webdriver
```

You need to install a current version of Ruby, which comes with RubyGems. You can find instructions on the Ruby project website. For more information on the Selenium Ruby bindings, check out the API documentation.

C# (WITH NUGET)

Use the following commands from the Package Manager Console window in Visual Studio to install the Selenium C# bindings:

```
Install-Package Selenium.WebDriver
Install-Package Selenium.Support
```

You need to install Microsoft Visual Studio and NuGet to install these libraries and build your project. For more information on the Selenium C# bindings, check out the API documentation.

The remaining examples use Java.

LAUNCHING A BROWSER

Selenium requires a "browser driver" in order to launch your intended browser. In all cases (except Safari), this driver must be downloaded and installed separately from the browser itself. For each example below, the code snippet will do no more than launch a single browser on your local machine.

CHROME

To use Chrome, you must download the ChromeDriver binary for your operating system (the highest number is the latest version) and add it to your System Path or specify its location during your test setup.

```
//Create a new instance of the ChromeDriver
WebDriver driver = new ChromeDriver();
```

Note: *For more information about ChromeDriver, check out the Chromium team's page for ChromeDriver.*

FIREFOX

To use Firefox, you must download the latest GeckoDriver. Please see this link for more details. You just need to request a new instance:

```
WebDriver driver = new FirefoxDriver();
```

Note: *For more information about FirefoxDriver, check out Mozilla's geckodriver project page.*

EDGE

Microsoft Edge requires Windows 10. Download a free virtual machine with Edge for testing purposes from Microsoft's Modern.IE developer portal and the appropriate Microsoft WebDriver server for your build of Windows (go to `Start > Settings > System > About` and locate the number next to OS Build on the screen). Then it's just a simple matter of requesting a new instance of Edge:

```
WebDriver driver = new EdgeDriver();
```

Note: *For more information about EdgeDriver, check out the main page on the Microsoft Developer portal and the download page for the EdgeDriver binary.*

As of early 2019, Microsoft Edge now uses the same rendering engine (Chromium) as Google's Chrome driver. This should generally give users confidence that they will operate similarly, but it is still currently recommended to test them as separate browsers.

SAFARI

Safari on OS X works without having to download a browser driver:

```
WebDriver driver = new SafariDriver();
```

Note: *Safari only runs on MacOS systems. For more information about SafariDriver, check out Apple's page for SafariDriver.*

INTERNET EXPLORER

For Internet Explorer on Windows, download IEDriverServer.exe (pick the highest number for the latest version) and either add it to your System Path or specify its location as part of your test setup.

```
WebDriver driver = new InternetExplorerDriver("/path/
to/IEDriverServer.exe");
```

Note: *Internet Explorer versions older than 11 are no longer supported by Microsoft. The* `InternetExplorerDriver` *still maintains support for some older versions but will not guarantee any such support after Selenium v4 ships. For more information, check out the Selenium project Wiki page for InternetExplorerDriver.*

COMMANDS AND OPERATIONS

The most common operations you'll perform in Selenium are navigating to a page and examining `WebElements`. You can then perform actions with those elements (e.g., click, type text, etc.), ask questions about them (e.g., Is it clickable? Is it displayed?), or pull information out of the element (e.g., the text of an element or the text of a specific attribute within an element).

VISIT A PAGE

```
driver.get("http://the-internet.herokuapp.com");
```

FIND AN ELEMENT

```
// find just one, the first one Selenium finds
WebElement element = driver.findElement(locator);
// find all instances of the element on the page
List<WebElement> elements = driver.
findElements(locator);
```

WORK WITH A FOUND ELEMENT

```
// chain actions together
driver.findElement(locator).click();
// store the element and then click it
WebElement element = driver.findElement(locator);
element.click();
```

PERFORM MULTIPLE COMMANDS

```
element.click();                  // clicks an element
element.submit();                 // submits a form
element.clear();                  // clears an input
field of its text
element.sendKeys("input text");   // types text into an
input field
```

ASK A QUESTION

Each of these returns a Boolean:

```
element.isDisplayed();    // is it visible to the human
eye?
element.isEnabled();      // can it be selected?
element.isSelected();     // is it selected?
```

RETRIEVE INFORMATION

Each of these returns a string:

```
// directly from an element
element.getText();
// by attribute name
element.getAttribute("href");
```

Note: *For more information about working with elements, check out the Selenium WebElement API Documentation.*

COMPLEX USER GESTURES

Selenium's Actions Builder enables more complex keyboard and mouse interactions. Things like drag-and-drop, click-and-hold, double-click, right-click, hover, etc.

```
// a hover example
WebElement avatar = driver.findElement(By.
name("target"));
(new Actions(driver)).moveToElement(avatar).build().
perform();
```

Note: *For more details about the Action Builder, check out the Actions API documentation.*

LOCATORS

In order to find an element on the page, you need to specify a locator. There are several locator strategies supported by Selenium:

BY LOCATOR	EXAMPLE (JAVA)
Class	`driver.findElement(By.className("dues"));`
CSS Selector	`driver.findElement(By.cssSelector(".flash.success"));`
ID	`driver.findElement(By.id("username"));`
Link Text	`driver.findElement(By.linkText("Link Text"));`
Name	`driver.findElement(By.name("elementName"));`
Partial Link Text	`driver.findElement(By.partialLinkText("nk Text"));`
Tag Name	`driver.findElement(By.tagName("td"));`
XPath	`driver.findElement(By.xpath("//input[@id='username']"));`

Note: *Good locators are unique, descriptive, and unlikely to change. So it's best to start with ID and Class locators. These are the most performant locators available and the most likely ones to be helpfully named. If you need to access something that doesn't have a helpful ID or Class, then use CSS selectors or XPath. But be careful when using these approaches, since they can be very brittle (and slow).*

Alternatively, talk with a developer on your team when the app does not present simple locators. Tell them what you're trying to automate and work with them to get more semantic markup added to the page. This will make the application more testable and make your tests far easier to write and maintain.

CSS SELECTORS

APPROACH	LOCATOR	DESCRIPTION
ID	`#example`	# denotes an ID
Class	`.example`	. denotes a Class
Classes	`.flash.success`	use . in front of each class for multiple
Direct Child	`div > a`	finds the element in the next child
Child/ subchild	`div a`	finds the element in a child or child's child
Next Sibling	`input.username + input`	finds the next adjacent element
Attribute Values	`form input[name ='username'] a`	great alternative to id and class matches
Attribute Values	`input[name='continue'] [type='button'] can ch`	main multiple attribute filters together
Location	`li:nth-child(4)`	finds the 4th element only if it is an li
Location	`li:nth-of-type(4)`	finds the 4th li in a list
Location	`*:nth-child(4)`	finds the 4th element regardless of type
Sub-string	`a[id^='beginning_'] find`	s a match that starts with (prefix)
Sub-string	`a[id$='_end'] find`	s a match that ends with (suffix)
Sub-string	`a[id*='gooey_center'] find`	s a match that con-tains (substring)
Inner Text	`a:contains('Log Out')`	an alternative to sub-string matching

For more info see one of the following resources:

- CSS Selectors Reference
- XPath Syntax Reference
- CSS & XPath Examples by Sauce Labs
- The difference between nth-child and nth-of-type
- How To Verify Your Locators

AN EXAMPLE TEST

To tie these concepts together, here is a simple test written in Java that demonstrates how to use Selenium to exercise a common functionality (e.g., login) by launching a browser, visiting the target page, interacting with the necessary elements, and verifying the page is in the correct place. Note that this example is intended to familiarize users with manipulating elements and the WebDriver API. A better method for abstracting and combining commands follows.

```
import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class TestLogin {
   private WebDriver driver;
   @Before
   public void setUp() {
       driver = new FirefoxDriver();
   }
   @After
   public void tearDown() {
       driver.quit();
   }
   @Test
   public void succeeded() {
       driver.get("http://the-internet.herokuapp.com/
login");
       driver.findElement(By.id("username")).
sendKeys("tomsmith");
       driver.findElement(By.id("password")).
sendKeys("SuperSecretPassword!");
       driver.findElement(By.cssSelector("button")).
click();
       assertTrue("success message not present",
             driver.findElement(By.cssSelector(".flash.
success")).isDisplayed());
   }
}
```

PAGE OBJECTS

Rather than integrate the calls to Selenium directly into your test methods, you can model your application's behavior in simple objects. This allows you to write your tests using user-centric language, rather than Selenium-centric language. This is called the "Page Object Model."

When your application changes and your tests break, you only have to update your Page Objects in one place in order to accommodate the changes. This gives us reusable functionality across our suite of tests, as well as more readable tests.

Let's create a Page Object for the login example shown before, then let's update the test to utilize it:

```
public class Login {
   private WebDriver driver;
   private By loginFormLocator = By.id("login");
   Private By usernameLocator  = By.id("username");
   private By passwordLocator  = By.id("password");
   private By submitButton     =
By.cssSelector("button");
   private By successMessageLocator = By.cssSelector(".
flash.success");

   public Login(WebDriver driver) {
       this.driver = driver;
       driver.get("http://the-internet.herokuapp.com/
login");
       assertTrue("The login form is not present",
               driver.findElement(loginFormLocator).
isDisplayed());
   }

   public void with(String username, String password) {
       driver.findElement(usernameLocator).
sendKeys(username);
       driver.findElement(passwordLocator).
sendKeys(password);
       driver.findElement(submitButton).click();
   }

   public Boolean successMessagePresent() {
       return driver.findElement(successMessageLocator).
isDisplayed();
   }
}
```

By storing the page's locators and behavior in a central place, we're able to reuse it in multiple tests and extend it for future use cases. This also enables us to pull all of our Selenium commands and locators out of our tests, making tests more concise and easier to construct.

We're also able to verify that the page is in a "ready state" before letting the test proceed — in this case, the constructor asserts that the login form is displayed. If it's not visible to the user, an exception will be thrown and the test will fail.

Now let's incorporate the Page Object into the test case itself:

```
public class TestLogin {
   private WebDriver driver;
   private Login login;

   @Before
   public void setUp() {
       driver = new FirefoxDriver();
       login = new Login(driver);
   }

   @After
   public void tearDown() {
       driver.quit();
   }

   @Test
   public void succeeded() {
       login.with("tomsmith", "SuperSecretPassword!");
       assertTrue("success message not present",
               login.successMessagePresent());
   }
}
```

CODE CONTINUES IN NEXT COLUMN

Page Objects should always return some piece of information (e.g., a predicate method like the one used in this example) or a new Page Object that represents the page the login took you to. How you write your Page Objects will vary depending on your context.

Here are some additional resources to consider as your use of Page Objects grows:

- Page Objects documentation from the Selenium project
- Martin Fowler's original Page Object article

WAITING

Waiting for the whole page to load should be a thing of the past. To make your tests work in an asynchronous, JavaScript-heavy world, we need to tell Selenium how to wait for particular elements more intelligently. There are two types of functions for this in Selenium: Implicit Waits and Explicit Waits.

The recommended approach from the Selenium project is to use Explicit Waits, or at the very least to choose either Implicit or Explicit Waits, and to not mix them in your code.

EXPLICIT WAITS

- Recommended approach
- Specify an amount of time and a Selenium action
- Selenium will try that action repeatedly until either:
  - the action can be accomplished, or
  - the amount of time specified has been reached, throwing a TimeoutException.

```
WebDriverWait wait = new WebDriverWait(driver,
timeout);
wait.until(ExpectedConditions.
visibilityOfElementLocated(locator));
```

Note: *For more info, check out the case against using Implicit and Explicit Waits together and Explicit vs. Implicit Waits.*

IMPLICIT WAITS

Using Implicit Waits is no longer recommended. It can cause unnecessary delays in testing time, and it masks the "intent" of your tests. There are many discussions online to study the topic more in depth. Also see the official documentation for Implicit Waits.

SCREENSHOTS ON FAILURE

Selenium can take screenshots of the browser window. We recommend taking a screenshot whenever a test fails. In JUnit, this done with a `TestWatcher` rule.

```
@Rule
public TestRule watcher = new TestWatcher() {
  @Override
  protected void failed(Throwable th, Description desc)
{
    File scrFile = ((TakesScreenshot)driver).
getScreenshotAs(OutputType.FILE);
    try {
      FileUtils.copyFile(scrFile,
        new File("failshot_"
          + desc.getClassName()
          + "_" + desc.getMethodName()
          + ".png"));
    } catch (IOException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

RUNNING TESTS IN PARALLEL

In order to run your tests on different browser/operating system combinations simultaneously, you need to initialize a special kind of WebDriver: a `RemoteWebDriver`. This allows you to execute your tests on a different machine that you maintain (using the Selenium Grid) or one of the many cloud providers (e.g., Sauce Labs, BrowserStack, etc). These providers allow you to pay for the use of cloud servers for test execution but in an environment that you don't have to spend time or resources to maintain.

SELENIUM GRID

There are two main elements to the Selenium Grid — a hub to manage the tests, and nodes to execute them. The hub ensures your tests end up on the right node and manages all communication between the nodes and your test code. Nodes host the browser/OS combinations and execute your test commands while providing constant feedback to the hub.

Selenium Grid comes built into the Selenium Standalone Server, which you can download here.

Then start the hub:

```
> java -jar selenium-server-standalone.jar -role hub
19:05:12.718 INFO - Launching Selenium Grid hub
...
```

After that, we can register nodes to the hub:

```
> java -jar selenium-server-standalone.jar -role node
-hub http://ip-address-or-dns-name-to-your-hub:4444/
grid/register
19:05:57.880 INFO - Launching a Selenium Grid node
...
```

Note: *To run node processes on multiple machines, you will need to place the standalone server on each machine and launch it with the same registration command (providing the IP Address or DNS name of the hub, and specifying additional parameters as needed).*

Once these processes are running, you must make a small change to your test config to create a `RemoteWebDriver` that will use the Grid.

```
FirefoxOptions options = new FirefoxOptions();
"http://ip-address-or-dns-name-to-your-hub:4444/wd/
hub";
driver = new RemoteWebDriver(new URL(url), options);
```

Selenium Grid is a great option for scaling your test infrastructure, but it doesn't give you parallelization for free. It can handle as many connections as you throw at it (within reason), but you must still find a way to execute your tests in parallel (with your test framework, for instance). Also, if you're considering standing up your own grid, check out docker-selenium, ecs-selenium (requires AWS), and Zalenium.

Note: *When Selenium 4.0 is officially released, these examples are subject to change and may no longer hold true.*

SELENIUM SERVICE PROVIDERS

Rather than take on the overhead of a standing up and maintaining a test infrastructure, you can easily outsource things to a third-party cloud provider (aka someone else's Grid) like Sauce Labs.

Note: *You'll need an account to use Sauce Labs. Their free trial offers enough to get you started. And if you're signing up because you want to test an open-source project, then check out their Open Sauce account.*

```
//Create the SauceOptions object with your credentials
and other info
MutableCapabilities sauceOptions = new
MutableCabilities();
sauceOptions.setCapability("username", "<username");
sauceOptions.setCapability("accesskey", "<accesskey");

//Create the ChromeOptions object with the browser info
you require
```

CODE CONTINUES ON NEXT PAGE

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setCapability("browserName", "chrome");
chromeOptions.setCapability("browserVersion", "75.0");
chromeOptions.setCapability("platform", "Windows 10");

//Apply the Sauce Options to the ChromeOptions object
chromeOptions.setCapability("sauce:options",
sauceOptions);

//Now tie it all together in a RemoteWebDriver
String sauceUrl = "http://ondemand.saucelabs.com:80/wd/
hub";
driver = new RemoteWebDriver(new URL(sauceUrl),
capabilities);
```

Note: *Check out Sauce Labs' documentation portal for more details.*

MOBILE SUPPORT

Within the WebDriver ecosystem, there are a few mobile testing solutions for both iOS and Android. Appium supports Selenium/WebDriver as well as the W3C standard for WebDriver and has its own Refcard. To get started with mobile, explore the many resources online, most notably:

- Appium (both iOS and Android)
- Selendroid (Android)
- WebDriverAgent (iOS)

Note: *If you're interested in Appium, then be sure to check out the Getting Started With Appium Refcard, as well as the Appium Bootcamp.*

UPDATED BY MARCUS MERRELL,
DIRECTOR OF TECHNICAL SERVICES, SAUCE LABS

Marcus Merrell has been working in quality engineering since 2001. He is a contributor to the Selenium project, as well as the co-chair of the Selenium Conference Organizing Committee. He has released several open source projects for testing and IoT and speaks at conferences worldwide. As the Director of Technical Services at Sauce Labs, Marcus manages the Solutions Architect team, which helps customers find success with the Sauce Labs platform. He is also experienced in software development management, user analytics, and marketing automation.

DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of tousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 919.678.0300