DZone REFCARDZ

BROUGHT TO YOU IN PARTNERSHIP WITH SAUCE LABS

CONTENTS

Selenium

BY DAVE HAEFFNER
WITH CONTRIBUTIONS FROM LUKE INMAN-SEMERAU, SELENIUM PROJECT COMMITTER

WHAT IS SELENIUM?

Selenium is a free and open-source browser automation library used by millions of people for testing purposes as well as automation of repetitive web-based administrative tasks. It has the support of the largest browser vendors who have taken (or are currently taking) steps to make Selenium a native part of their browser. It is also the core technology in countless other browser automation tools, APIs and frameworks.

It is effectively the de facto standard for automating web browsers. WebDriver (the API to automate browsers, maintained by the Selenium project) is currently going through a W3C (World Wide Web Consortium) specification. Once the specification becomes ratified, Selenium will become the official standard for automating web browsers.

GETTING STARTED

There are Selenium language bindings for multiple programming languages. The officially supported ones (in order of use) are:

- 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`.

```
<!— https://mvnrepository.com/artifact/org.seleniumhq.
    selenium/selenium-java —>
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-server</artifactId>
    <version>LATEST</version>
    <scope>test</scope>
</dependency>
```

Note: You will need to have the Java Development Kit (version 7+, or 8+ for 3.x versions of Selenium) and Maven installed on your machine. For more information on the Selenium Java bindings, check out the API documentation.

JAVASCRIPT (NPM)

Type the following command into a command-prompt to install the JavaScript bindings for Selenium.

```
npm install selenium-webdriver
```

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

PYTHON

Type the following command to install the Python bindings for Selenium.

```
pip install selenium
```

Note: You will 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

Type the following command to install the Selenium Ruby bindings.

```
gem install selenium-webdriver
```

Note: You will need to install a current version of Ruby which comes with RubyGems. You can find instructions for that 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
```

Note: You will 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.

Note: The remaining examples will show Java demonstrations.

LAUNCHING A BROWSER

With Selenium you have the ability to launch any major browser on any major operating system. Depending on the browser, there may be some configuration required in order to get it running locally.

FIREFOX

FirefoxDriver comes built into the Selenium language bindings. It's a simple matter of requesting a new instance of it.

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

Note: For more information about FirefoxDriver, check out the Selenium project Wiki page for FirefoxDriver.

A brief history of web and mobile app testing.

Find out how Sauce Labs can accelerate your testing to the speed of awesome.

For a demo, please visit saucelabs.com/demo
Email sales@saucelabs.com or call (855) 677-0011 to learn more.

SAUCELABS
Testing at the speed of awesome.

CHROME

In order to use Chrome, you need to download the ChromeDriver binary for your operating system (pick the highest number for the latest version). You either need to add it to your System Path or specify its location as part of your test setup.

```
System.setProperty("webdriver.chrome.driver",
    "/path/to/chromedriver");
driver = new ChromeDriver();
```

Note: For more information about ChromeDriver, check out the Selenium project Wiki page for ChromeDriver.

INTERNET EXPLORER

For Internet Explorer on Windows, you need to 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: As of July 19, 2016 Internet Explorer 8 and older are no longer supported by the Selenium project. Also, if you're trying to run Windows 11, then you will need to add a registry key to your system. For more information about this and other InternetExplorerDriver details, check out the Selenium project Wiki page for InternetExplorerDriver.

EDGE

In order to use Microsoft Edge, you need to have access to Windows 10. You can download a free virtual machine with it from Microsoft for testing purposes from Microsoft's Modern.IE developer portal. After that, you need to download the appropriate `Microsoft WebDriver` server for your build of Windows. To find that, 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: Currently Edge is only supported in the C#, Java, and JavaScript bindings. For more information about EdgeDriver, check out the main page on the Microsoft Developer portal and the download page for the EdgeDriver binary.

SAFARI

Safari on OS X works without having to download a browser driver. But you do need to download and install a SafariDriver browser extension which you can get from this direct download link from the Selenium project.

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

Note: There is no Selenium support for Safari on Windows. For more information about SafariDriver, check out the Selenium project Wiki page for SafariDriver.

OPERA

Versions 15 and newer of Opera are built from the same rendering engine as Chrome. So if you run your tests with ChromeDriver then you are essentially testing Opera too.

There are some slight differences with it though. So if you have a business need to test with Opera, be sure to check out the OperaChromiumDriver for current versions of Opera and the OperaPrestoDriver for older versions of Opera.

COMMANDS & OPERATIONS

The most common operations you'll end up doing in Selenium are navigating to a page and finding an element (or a set of elements). Once found, you can 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 (listed alphabetically):

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 you can write very brittle locators that will not perform well. Alternatively, talk with a developer on your team when the app is hard to automate. 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/ subschild	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 chain 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_']	finds a match that starts with (prefix)
Sub-string	a[id$='_end']	finds a match that ends with (suffix)
Sub-string	a[id*='gooey_center']	finds a match that contains (substring)

Note: Older browsers (e.g. Internet Explorer 8) don't support CSS Pseudo-classes, so some of these locator approaches won't work on them.

For more info see one of the following resources:

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

AN EXAMPLE TEST

To tie all of these concepts together, here is a simple test that demonstrates using Selenium to exercise 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.

```
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 write your test code directly against your application, you can model the behavior of it into simple objects and write your tests against them instead. That way, when your application changes and your tests break, you only have to update your test code in one place to fix it. With this approach we not only get the benefit of controlled chaos, but we also get reusable functionality across our suite of tests and more readable tests.

Let's create a page object for the login example shown before and update the test from before to use it.

```
// import statements omitted for brevity
public class Login {
    private WebDriver driver;
    By loginFormLocator = By.id("login");
    By usernameLocator  = By.id("username");
    By passwordLocator  = By.id("password");
    By submitButton     = By.cssSelector("button");
    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 are able to easily 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 our tests more concise and easier to construct. We also have the advantage of doing additional things like verifying that the page is in a ready state before letting the test proceed (e.g., like we're doing in the constructor by asserting that the login form is present).

```
// import statements omitted for brevity
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());
    }
}
```

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 may vary depending on your preference/experience. The example demonstrated above is a simple approach. Here are some additional resources to consider as your usage of Page Objects grows:

- Page Objects documentation from the Selenium project
- Page Factory (a Page Object generator/helper built into Selenium)
- HTML Elements (a simple open-source Page Object framework)

WAITING

The notion of waiting for the page to finish loading is a thing of the past. In order to make your tests work in an asynchronous, JavaScript-heavy world, we need to tell Selenium how to wait for things intelligently. There are two types of function for this built into Selenium—implicit waits and explicit waits. The recommended approach from the Selenium project is to only use Explicit Waits.

IMPLICIT WAITS

- Only needs to be configured once at test setup
- Tells Selenium to wait for a specified amount of time before raising an exception (typically a NoSuchElementException)
- Less flexible than explicit waits
- Known to cause transient test failures

```
driver.manage().timeouts().implicitlyWait(
    5, TimeUnit.SECONDS);
```

Note: This is not a recommended approach for waiting with Selenium. The recommended approach is to use Explicit Waits.

EXPLICIT WAITS

- Recommended approach to wait in your tests
- 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.

SCREENSHOTS ON FAILURE

Selenium has the ability to take screenshots of the browser viewport window. The most common way to leverage this functionality is when there is a test failure. In JUnit, this is simple to accomplish with a `TestWatcher` Rule.

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

SELENIUM REMOTE

In order to run your tests on a myriad of browser and operating system combinations, you need a way to run them at scale. With Selenium Remote, you can point your tests at a Selenium Grid that you maintain or someone else's Grid (e.g. Sauce Labs), which they maintain and you pay for usage.

SELENIUM GRID

There are two main elements to Selenium Grid—a hub, and nodes. First you need to stand up a hub. Then you can connect (or "register") nodes to that hub. Nodes are where your tests will run, and the hub is responsible for making sure your tests end up on the right node (e.g., the machine with the operating system and browser you specified in your test setup).

Selenium Grid comes built into the Selenium Standalone Server, which you can download here (pick the highest number for the latest version).

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 span nodes across 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 your hub, and specifying additional parameters as needed).

Then it is a simple matter of connecting to the Grid Hub in our test setup.

```
DesiredCapabilities capabilities =
    DesiredCapabilities.firefox();
String url =
    "http://ip-address-or-dns-name-to-your-hub:4444/wd/hub";
driver = new RemoteWebDriver(new URL(url), capabilities);
```

Note: 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 still need to find a way to execute your tests in parallel (with your test framework, for instance). Also, if you are considering standing up your own grid, be sure to check out docker-selenium, Selenium-Grid-Extras, and SeleniumGridScaler.

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 (a.k.a. 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 be sure to check out their Open Sauce account.

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setCapability("browserName", "firefox");
capabilities.setCapability("version", "33");
capabilities.setCapability("platform", "Windows XP");
String sauceUrl = String.format(
    "http://%s:%s@ondemand.saucelabs.com:80/wd/hub",
    sauceUser, sauceKey);
driver = new RemoteWebDriver(new URL(sauceUrl),
    capabilities);
```

Note: You can see a full list of Sauce Labs' available platform options here. There's also a handy configuration generator that will tell you what values to plug into your test. Be sure to check out Sauce Labs' documentation portal for more details.

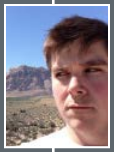
MOBILE SUPPORT

Within the WebDriver ecosystem, there are a few mobile testing solutions for both iOS and Android, 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 Appium Bootcamp.

ABOUT THE AUTHOR



DAVE HAEFFNER is the writer of Elemental Selenium — a free, once weekly Selenium tip newsletter that's read by thousands of testing professionals. He's also the creator and maintainer of the-internet (an open-source web app that's perfect for writing automated tests against), and author of The Selenium Guidebook. He's helped numerous companies successfully implement automated acceptance testing; including The Motley Fool, ManTech International, Sittercity, and Animoto. He's also an organizer of Selenium Conf, a member of the Selenium project, and has spoken at numerous conferences and meet-ups around the world about automated functional testing.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

BROUGHT TO YOU IN PARTNERSHIP WITH
SAUCELABS