The [Selenium](http://www.seleniumhq.org/) WebDriver scripts are very slow because they run through the browser.

There are multiple things that can improve the Selenium WebDriver scripts’ speed:

1. use fast selectors
2. use fewer locators
3. create atomic tests
4. dont test the same functionality twice
5. write good tests
6. use only explicit waits
7. use the chrome driver
8. use drivers for headless browsers
9. re-use the browser instance
10. run scripts in parallel
11. use HTTP parse libraries
12. pre-populate cookies
13. do not load images in the web page

**USE FAST SELECTORS**

Selenium WebDriver scripts allow using many types of locators.

When selecting the locator, start with the fast ones:

1. **search by ID**

This works if the html element has the id attribute.

It is the fastest locator as it uses the document.getElementById() javascript command which is optimized for all browsers.

2. **NAME selector**

This works if the element has a name attribute.

3. **CSS selector**

It is faster than XPATH but not as flexible.

4. **XPATH selector**

This is the most flexible strategy for building locators.

But xpath selectors are the slowest of all as the browser dom of the web page needs to be traversed for finding the element.

**USE FEW LOCATORS IN SCRIPTS**

Keep the number of locators as low as possible especially if using XPATH.

This goes together with keeping scripts focused and short.



**CREATE ATOMIC TESTS**

**Avoid testing too many things in a single script.**

It is better to have more small scripts focused on one thing only than fewer big scripts that do too many things.

* **Each test should test a single tiny bit of functionality**
* Each test should have up to 20 steps.
* Each test should run in under 10 seconds.

Having atomic test is very useful in case of failures.

The failures are very clear about where the problem is.

Example

Lets assume that one of the scripts is called testSignupLoginChangePasswordLogout().

This script checks 4 different things:

1. Signup form works
2. Login works
3. Change password works
4. Logout works

If the script fails, which part has an error? Signup? Login? ChangePassword? Logout?

Instead, we should use smaller and atomic scripts:

1. testSignup()
2. testLogin()
3. testChangePassword()
4. testLogout()

If any of the smaller tests fails, it is very clear where the problem is.

**DONT TEST THE SAME FUNCTIONALITY TWICE**



Make sure that your scripts are not checking the same features over and over.

If you checked that the login works correctly in one script, there is no benefit from checking it again in another script.

**WRITE GOOD TESTS**

Good tests are created by following a few simple rules:**Good written tests are fast tests**.

* Eliminate step duplication
* Keep the scripts independent
* Write just enough steps to go from A to B in the quickest way possible
* Remove steps that are not related to the final result

[[](http://test-able.blogspot.com/2015/08/how-to-create-your-first-selenium-webdriver-project.html)](http://test-able.blogspot.com/2015/08/how-to-create-your-first-selenium-webdriver-project.html)

**USE ONLY EXPLICIT WAITS**

One of the best ways of making a script faster is by using only explicit waits.

If your test scripts uses delays and implicit waits like this,

*WebDriver driver = new FirefoxDriver();*  
 *driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);*  
 *driver.get(“*[*http://somedomain/url\_that\_delays\_loading&#8221*](http://somedomain/url_that_delays_loading&#8221)*;);*  
 *WebElement myElement=driver.findElement(By.id(“myElement”));*

replace them with explicit waits.

*WebDriver driver = new FirefoxDriver();*  
 *driver.get(“*[*http://somedomain/url\_that\_delays\_loading&#8221*](http://somedomain/url_that_delays_loading&#8221)*;);*  
 *WebElement element = (new WebDriverWait(driver, 10))*  
*.until(ExpectedConditions.presenceOfElementLocated(By.id(“element”)));*



The script performance is better with **explicit waits as HTML elements are accessed as soon as they become available.**

No additional waiting times are needed.

**USE THE CHROME DRIVER AND CHROME BROWSER**

**If you need a real browser, use Chrome and the Chrome driver.**

The Chrome browser is faster than IE, Firefox, Safari and Opera.

The same is valid for the Chrome web driver.

|  |
| --- |
| [https://seleniumjavadotcom.files.wordpress.com/2015/07/7d233-browsers.png?w=528&h=640](https://seleniumjavadotcom.files.wordpress.com/2015/07/7d233-browsers.png) |
| **RESULTS OF RUNNING THE SAME SCRIPT 10 TIMES** |

**USE HEADLESS BROWSERS AND DRIVERS**

If you dont need a real browser or you use continuous integration, **headless browsers and drivers can make your scripts faster.**

A headless browser is a browser that does not have a user interface.

The most popular headless browsers are HTML UNIT and Phantom JS:

**HTML UNIT**

* headless browser written in Java
* uses the RHINO engine; this is not great as this engine is not used by any of the existing real browsers
* provides javascript and ajax support and partial rendering capability

Example  
driver = new HtmlUnitDriver();

[**PHANTOM JS**](http://phantomjs.org/)

* headless browser that uses the WEBKIT layout engine for rendering web pages and Javascript Core for executing scripted tests
* the WEBKIT engine is used by real browsers such as CHROME and SAFARI
* uses [Ghost Driver](https://github.com/detro/ghostdriver) as an implementation of the WebDriver Wire Protocol



Example

WebDriver driver;

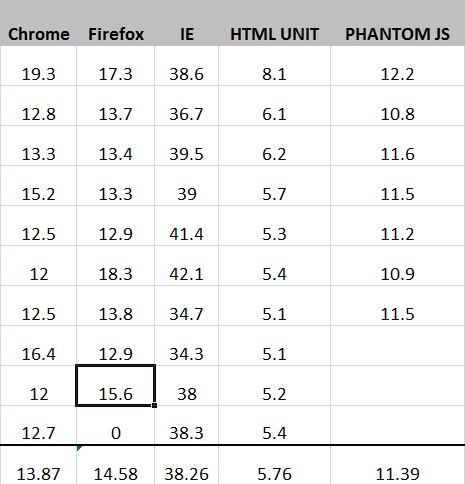
DesiredCapabilities capabilities= DesiredCapabilities.phantomjs();

capabilities.setJavascriptEnabled(true);

capabilities.setCapability(“phantomjs.binary.path”,”c:\\selenium\\browserdrivers\\phantomjs\\bin\\phantomjs.exe”);

capabilities.setJavascriptEnabled(true);

driver = new PhantomJSDriver(capabilities);

[](https://seleniumjavadotcom.files.wordpress.com/2015/07/dd298-headless2bbrowsers.png)

**REUSE THE BROWSER INSTANCE**

A typical JUNIT class includes the following components:

1. constructor
2. setUp() method
3. tearDown() method
4. test scripts

The setUp() method uses the @Before annotation and runs before each test script.

Its purpose is usually to create the driver object and open the browser window.

The tearDown() method uses the @After annotation and runs after each test script.

Its purpose is to destroy the driver object and close the browser window.

Each test script is using its own browser instance when the @Before and @After are used:

setUp() –> opens browser  
Test Script 1  
tearDown() –> closes browser

setUp() –> opens browser  
Test Script 2  
tearDown() –> closes browser

setUp() –> opens browser  
Test Script 3  
tearDown() –> closes browser

setUp() –> opens browser  
Test Script 4  
tearDown() –> closes browser

The browser  instance can be shared by all test scripts if we use different annotations:

1. @BeforeClass instead of @Before for the setUp() method
2. @AfterClass instead of @After for the tearDown() method

In this case, the setUp() method does not run before each test script but just once before all test scripts.

The tearDown() method runs once only after all test scripts.

After each test script, the browser is not closed so the next script can re-use it:

setUp() –> opens browser  
Test Script 1

Test Script 2

Test Script 3

Test Script 4  
tearDown() –> closes browser



**RUN THE SCRIPTS IN PARALLEL**

By default, JUNIT test scripts are run sequentially.

There are a few options for running them in parallel:

**1. Run the scripts in parallel on the local computer**

This can be done by

* creating a Maven project in Eclipse
* adding the [Maven Surefire](https://maven.apache.org/surefire/maven-surefire-plugin/) plugin to the POM.XML file

Maven Surefire allows methods or classes to be run in parallel.

It also allows the thread count to be set up.

See below a POM.xml file sample:

*<project xmlns=”*[*http://maven.apache.org/POM/4.0.0&#8243*](http://maven.apache.org/POM/4.0.0&#8243)*; xmlns:xsi=”*[*http://www.w3.org/2001/XMLSchema-instance&#8221*](http://www.w3.org/2001/XMLSchema-instance&#8221)*; xsi:schemaLocation=”*[*http://maven.apache.org/POM/4.0.0*](http://maven.apache.org/POM/4.0.0)[*http://maven.apache.org/xsd/maven-4.0.0.xsd”&gt*](http://maven.apache.org/xsd/maven-4.0.0.xsd%E2%80%9D&gt)*;*

*<modelVersion>4.0.0</modelVersion>*

*<groupId>com.siminiuc</groupId>*

*<artifactId>MavenProject</artifactId>*

*<version>0.0.1</version>*

*<name>Maven Project</name>*

*<url>http://maven.apache.org</url&gt;*

*<dependencies>  
     <dependency>*

*<groupId>junit</groupId>*

*<artifactId>junit</artifactId>*

*<version>4.12</version>*

*<scope>test</scope>*

*</dependency>*

*<dependency>*

*<groupId>org.seleniumhq.selenium</groupId>*

*<artifactId>selenium-firefox-driver</artifactId>*

*<version>2.47.1</version>*

*</dependency>*

*</dependencies>*

*<build>*

*<plugins>*

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-compiler-plugin</artifactId>*

*<version>2.3.2</version>*

*<configuration>*

*<source>1.7</source>*

*<target>1.7</target>*

*<executable>C:\JDK\bin\javac.exe</executable>*

*</configuration>*

*</plugin>*

*<plugin>*

*<groupId>org.apache.maven.plugins</groupId>*

*<artifactId>maven-surefire-plugin</artifactId>*

*<version>2.19</version>*

*<configuration>*

*<parallel>methods</parallel>*

*<threadCount>5</threadCount>*

*</configuration>*

*</plugin>*

*</plugins>*

*</build>*

*</project>*

Running scripts in parallel using Maven Surefire reduces the execution time with about 50%.

[[](http://test-able.blogspot.com/2015/08/how-to-create-your-first-selenium-webdriver-project.html)](http://test-able.blogspot.com/2015/08/how-to-create-your-first-selenium-webdriver-project.html)

**2. Use Selenium Grid**

Generally speaking, there’s two reasons why you might want to use Selenium-Grid.

* To run your tests against multiple browsers, multiple versions of browser, and browsers running on different operating systems.
* To reduce the time it takes for the test suite to complete a test pass.

Selenium-Grid is used to speed up the execution of a test pass by using multiple machines to run tests in parallel.

For example, if you have a suite of 100 tests, but you set up Selenium-Grid to support 4 different machines to run those tests, your test suite will complete in (roughly) one-fourth the time as it would if you ran your tests sequentially on a single machine.

How does it work?

A grid consists of a single hub, and one or more nodes.

Both are started using the selenium-server.jar executable.

The hub receives a test to be executed along with information on which browser and ‘platform’ (i.e. WINDOWS, LINUX, etc) where the test should be run.

It ‘knows’ the configuration of each node that has been ‘registered’ to the hub.

Using this information, it selects an available node that has the requested browser-platform combination.

Once a node has been selected, Selenium commands initiated by the test are sent to the hub, which passes them to the node assigned to that test.

The node runs the browser, and executes the Selenium commands within that browser against the application under test.

**3. Use**[**Sauce Labs**](https://saucelabs.com/)**(selenium grid in the cloud)**

It works similar with Selenium Grid with the advantage that you do not need to set up the grid environment.

It has hundreds of combinations of browser/device/operating system available.

**USE HTTP PARSE LIBRARIES**

There are cases when Selenium WebDriver is not the best option for creating test scripts.

For example, lets take a simple test scenario:

1. the user opens the home page of a website
2. the user does a keyword search
3. the results page is opened and it displays multiple results
4. for each result displayed, open the details page and check that result information is included

it is quite simple to implement this with Selenium WebDriver.

See below a code sample:

*@Test*  
*public void test1() throws InterruptedException {*  
 *//1. open the home page of the site*  
*driver.get(“*[*http://www.vpl.ca&#8221*](#8221)*;);*  
 *//2. execute keyword search*  
*WebElement searchField = wait.until(ExpectedConditions.*  
*visibilityOfElementLocated(searchFieldLocator));*  
 *searchField.click();*  
*searchField.sendKeys(“java”);*  
 *WebElement searchButton = wait.until(ExpectedConditions.*  
*elementToBeClickable(searchButtonLocator));*  
 *searchButton.click();*  
 *//4. iterates through all results links*  
*for (int i = 1; i <= 10; i++) {*  
 *WebElement resultTitle = wait.until(ExpectedConditions.*  
*elementToBeClickable(resultTitleLocator));*  
 *resultTitle.click();*  
 *//checks that the result title exists on details page*  
*WebElement bookTitleElement = wait.until(ExpectedConditions.*  
*visibilityOfElementLocated(bookTitleLocator));*  
 *assertTrue(bookTitleElement.getText().length() > 0);*  
 *//checks that the author value exists on the details page*  
*try {*  
 *WebElement bookAuthorElement = wait.until(ExpectedConditions.*  
*visibilityOfElementLocated(bookAuthorLocator));*  
 *assertTrue(bookAuthorElement.getText().length() > 0);*  
 *}*  
*catch(Exception e) { }*  
 *//go back to results page*  
*driver.navigate().back();*  
*}*  
*}*

Since everything happens through the browser, the script executes in approximately 100 seconds.

In cases like this, **HTML parsing libraries like**[**JSOUP**](http://jsoup.org/)**are a better choice than Selenium WebDriver.**

Lets see what JSOUP is about.

jsoup is a Java library for working with real-world HTML.

It provides a very convenient API for extracting and manipulating data, using the best of DOM, CSS, and jquery-like methods.

jsoup implements the WHATWG HTML5 specification, and parses HTML to the same DOM as modern browsers do.

* scrape and parse HTML from a URL, file, or string
* find and extract data, using DOM traversal or CSS selectors
* manipulate the HTML elements, attributes, and text
* clean user-submitted content against a safe white-list, to prevent XSS attacks
* output tidy HTML

Lets see the same script using JSOUP:

*@Test public void test1() throws IOException {*  
 *Document resultsPage = Jsoup.connect(“*[*https://vpl.bibliocommons.com/search?q=java&t=keyword&#8221*](https://vpl.bibliocommons.com/search?q=java&t=keyword&#8221)*;).get();*  
 *Elements titles = resultsPage.select(“span.title”);*  
 *for (Element title : titles) {*  
 *Element link = title.child(0);*  
 *String detailsPageUrl = “*[*https://vpl.bibliocommons.com&#8221*](#8221)*; + link.attr(“href”);*  
 *Document detailsPage = Jsoup.connect(detailsPageUrl).get();*  
 *Elements bookTitle = detailsPage.getElementsByAttributeValue(“testid”, “text\_bibtitle”);*  
 *if (bookTitle.size() > 0)*  
*assertTrue(bookTitle.get(0).text().length() > 0);*  
 *Elements bookAuthor = detailsPage.getElementsByAttributeValue(“testid”, “author\_search”);*  
 *if (bookAuthor.size() > 0)*  
*assertTrue(bookAuthor.get(0).text().length() > 0);*  
*}*  
*}*  
*}*

The script is not only shorter but much faster (9 seconds).

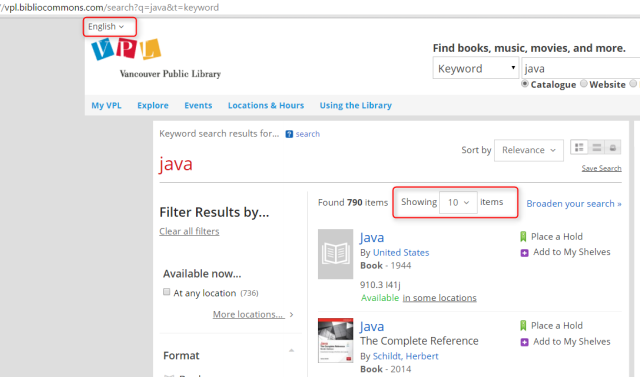
**PRE-POPULATE COOKIES**



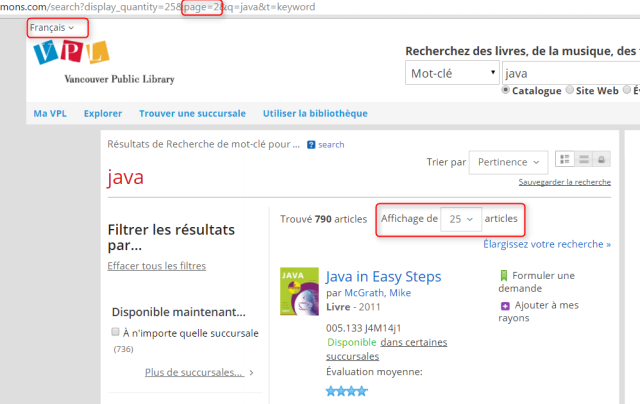
Pre-populating site cookies can speed up scripts by avoiding additional site navigation.

Lets take the following test case:

1. open the the home page of a site
2. execute a keyword search
3. the results page is displayed with 10 results; language is english

[](https://seleniumjavadotcom.files.wordpress.com/2015/07/b7549-page2b1.png)

1. go to page 2 of results
2. change the language to french; the page reloads in french
3. change the number of results to 25; the page reloads with 25 results

[](https://seleniumjavadotcom.files.wordpress.com/2015/07/7c1b9-page2b2.png)

1. go to page 3 of results
2. change the language to chinese; the page reloads in chinese
3. change the number of results to 3; the page reloads with 3 results

[](https://seleniumjavadotcom.files.wordpress.com/2015/07/ba28e-page2b3.png)

For both pages 2 and 3, the script needs to

* change the language through a listbox; the page reloads when a different language is selected.
* change the number of results through another listbox; the page reloads with the new number of results

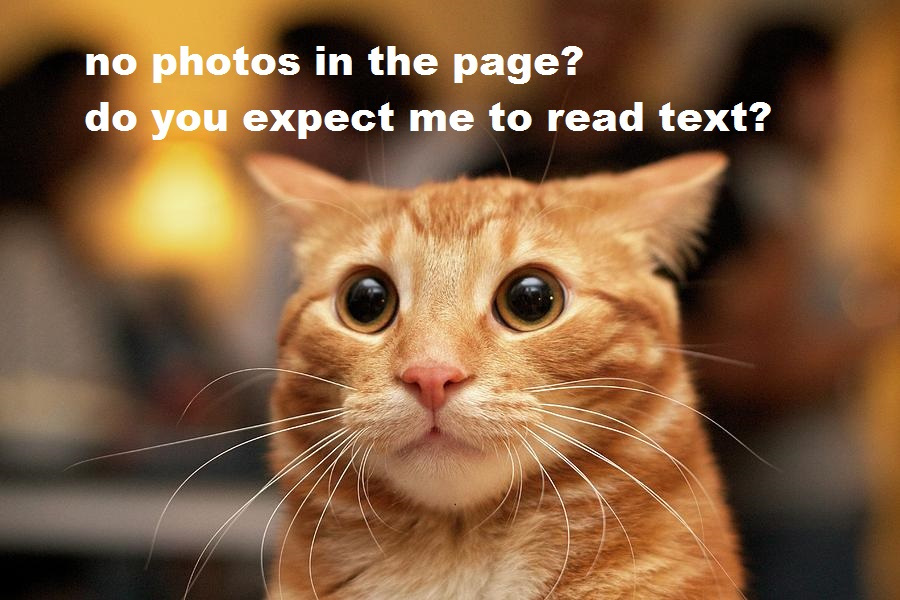
Since the page sets cookies when the results number and language change, it is quite easy to add cookies to the site before loading page 2 and 3.

This way, page 2 will load directly with 25 results and in french.  
Page 3 will load directly in chinese with 3 results.

**Code Sample:**

*public void addCookie(String name, String value) {*  
 *Cookie pageSize = new Cookie.Builder(name, value)*  
*.domain(“vpl.bibliocommons.com”)*  
*.expiresOn(new Date(2016, 12, 31))*  
*.isSecure(true)*  
*.path(“/”)*  
*.build();*  
 *driver.manage().addCookie(pageSize);*  
*}*  
  
*@Test*  
*public void testResult() throws InterruptedException {*  
  
*driver.get(“*[*https://vpl.bibliocommons.com/search?q=java&t=keyword&#8221*](https://vpl.bibliocommons.com/search?q=java&t=keyword&#8221)*;);*  
*Thread.sleep(5000);*  
  
*addCookie(“page\_size”, “25”);*  
*addCookie(“language”, “fr-CA”);*  
  
*driver.get(“*[*https://vpl.bibliocommons.com/search?page=1&q=java&t=keyword&#8221*](https://vpl.bibliocommons.com/search?page=1&q=java&t=keyword&#8221)*;);*  
  
*Thread.sleep(5000);*  
  
*addCookie(“page\_size”, “3”);*  
*addCookie(“language”, “zh-CN”);*  
 *driver.get(“*[*https://vpl.bibliocommons.com/search?page=2&q=java&t=keyword&#8221*](https://vpl.bibliocommons.com/search?page=2&q=java&t=keyword&#8221)*;);*  
*Thread.sleep(5000);*  
*}*

**DO NOT LOAD ANY IMAGES IN THE WEBPAGE**



Many site pages are very rich in images so they do not load fast in the browser.

If the page loads faster in the browser, the script will be faster as well.

**The page will load faster if no images are loaded.**

This can be accomplished by setting up a new profile for the browser and disable loading images for it.

Then, in the script, use the new profile when starting the browser driver:

1. In chrome type : chrome://version/

 2. Using the console, navigate to the directory “Executable Path” using CD command.

 3. type : chrome.exe –user-data-dir=”your\custom\directory\path” where to create your custom profile.

 4. When chrome opens and ask you for account, click a small link that declines to use an account.

 5. In the opened chrome goto settings>Advanced settings> Privacy > Content Settings > Images > Do Not Show Any Image

 6. If you wish to disable some Chrome Extensions that will be unnecessary for your driver and may impact the driver performance, disable them in : chrome://extensions/

 7. Close chrome and reopen it again from the console with the same command sequence used in 3. Verify that it is not loading the images and that the extensions you disabled (if any) are disabled.

8. Now, in your code, create the capabilities for the driver.

System.setProperty(“webdriver.chrome.driver”, “C:\\Selenium\\BrowserDrivers\\chromedriver.exe”);

ChromeOptions options = new ChromeOptions();  
options.addArguments(“user-data-dir=C:\\Selenium\\ChromeProfile”);  
options.addArguments(“–start-maximized”);

driver = new ChromeDriver(options);