

The Selenium Browser Automation Project

- 1: [Selenium overview](#)
 - 1.1: [Selenium components](#)
 - 1.2: [A deeper look at Selenium](#)
- 2: [WebDriver](#)
 - 2.1: [Getting started](#)
 - 2.1.1: [Install a Selenium library](#)
 - 2.1.2: [Install browser drivers](#)
 - 2.1.3: [Open and close a browser with Selenium](#)
 - 2.1.4: [Write your first Selenium script](#)
 - 2.1.5: [Upgrade to Selenium 4](#)
 - 2.2: [WebDriver Capabilities](#)
 - 2.2.1: [Shared capabilities](#)
 - 2.2.2: [Capabilities specific to Chromium browsers](#)
 - 2.2.3: [Capabilities specific to Firefox browser](#)
 - 2.2.4: [Capabilities specific to Internet Explorer browser](#)
 - 2.2.5: [Capabilities specific to Safari browser](#)
 - 2.3: [Browser](#)
 - 2.3.1: [Browser navigation](#)
 - 2.3.2: [JavaScript alerts, prompts and confirmations](#)
 - 2.3.3: [Working with cookies](#)
 - 2.3.4: [Working with iFrames and frames](#)
 - 2.3.5: [Working with windows and tabs](#)
 - 2.4: [Web elements](#)
 - 2.4.1: [Locator strategies](#)
 - 2.4.2: [Finding web elements](#)
 - 2.4.3: [Interacting with web elements](#)
 - 2.4.4: [Information about web elements](#)
 - 2.4.5: [Working with select list elements](#)
 - 2.5: [Remote WebDriver](#)
 - 2.6: [Configuring driver parameters](#)
 - 2.7: [Waits](#)
 - 2.8: [Actions API](#)
 - 2.8.1: [Keyboard actions](#)
 - 2.8.2: [Mouse actions](#)
 - 2.8.3: [Pen actions](#)
 - 2.8.4: [Scroll wheel actions](#)
 - 2.9: [BiDirectional functionality](#)
 - 2.9.1: [BiDirectional API](#)
 - 2.9.2: [RemoteWebDriver BiDirectional API](#)
 - 2.9.3: [Chrome DevTools](#)
 - 2.10: [Additional features](#)
 - 2.10.1: [Working With Colors](#)
 - 2.10.2: [File Upload](#)
 - 2.10.3: [ThreadGuard](#)
- 3: [Selenium Grid 4](#)
 - 3.1: [Getting started with Selenium Grid](#)
 - 3.2: [When to Use a Grid](#)

- 3.3: [Selenium Grid Components](#)
- 3.4: [Configuration of Components](#)
 - 3.4.1: [Configuration help](#)
 - 3.4.2: [CLI options in the Selenium Grid](#)
 - 3.4.3: [TOML configuration options](#)
- 3.5: [Grid architecture](#)
- 3.6: [Advanced features of Selenium](#)
 - 3.6.1: [Observability in Selenium Grid](#)
 - 3.6.2: [GraphQL query support](#)
 - 3.6.3: [Grid endpoints](#)
- 4: [IE Driver Server](#)
 - 4.1: [Internet Explorer Driver Internals](#)
- 5: [Selenium IDE](#)
- 6: [Test Practices](#)
 - 6.1: [Design patterns and development strategies](#)
 - 6.2: [Overview of Test Automation](#)
 - 6.3: [Types of Testing](#)
 - 6.4: [Encouraged behaviors](#)
 - 6.4.1: [Page object models](#)
 - 6.4.2: [Domain specific language](#)
 - 6.4.3: [Generating application state](#)
 - 6.4.4: [Mock external services](#)
 - 6.4.5: [Improved reporting](#)
 - 6.4.6: [Avoid sharing state](#)
 - 6.4.7: [Tips on working with locators](#)
 - 6.4.8: [Test independency](#)
 - 6.4.9: [Consider using a fluent API](#)
 - 6.4.10: [Fresh browser per test](#)
 - 6.5: [Discouraged behaviors](#)
 - 6.5.1: [Captchas](#)
 - 6.5.2: [File downloads](#)
 - 6.5.3: [HTTP response codes](#)
 - 6.5.4: [Gmail, email and Facebook logins](#)
 - 6.5.5: [Test dependency](#)
 - 6.5.6: [Performance testing](#)
 - 6.5.7: [Link spidering](#)
 - 6.5.8: [Two Factor Authentication](#)
- 7: [Legacy](#)
 - 7.1: [Selenium RC \(Selenium 1\)](#)
 - 7.2: [Selenium 2](#)
 - 7.2.1: [Migrating from RC to WebDriver](#)
 - 7.2.2: [Backing Selenium with WebDriver](#)
 - 7.2.3: [Legacy Firefox Driver](#)
 - 7.2.4: [Selenium grid 2](#)
 - 7.2.5: [History of Grid Platforms](#)
 - 7.2.6: [Remote WebDriver standalone server](#)
 - 7.2.7: [Limitations of scaling up tests in Selenium 2](#)
 - 7.2.8: [Stealing focus from Firefox in Linux](#)
 - 7.2.9: [Untrusted SSL Certificates](#)
 - 7.2.10: [WebDriver For Mobile Browsers](#)
 - 7.2.11: [Frequently Asked Questions for Selenium 2](#)
 - 7.2.12: [Selenium 2.0 Team](#)
 - 7.3: [Selenium 3](#)
 - 7.3.1: [Grid 3](#)

- 7.3.2: [Setting up your own Grid 3](#)
- 7.3.3: [Components of Grid 3](#)
- 7.4: [Legacy Selenium IDE](#)
 - 7.4.1: [HTML runner](#)
 - 7.4.2: [Legacy Selenium IDE Release Notes](#)
- 7.5: [JSON Wire Protocol Specification](#)
- 7.6: [Legacy Selenium Desired Capabilities](#)
- 7.7: [Legacy developer documentation](#)
 - 7.7.1: [Crazy Fun Build Tool](#)
 - 7.7.2: [Buck Build Tool](#)
 - 7.7.3: [Adding new drivers to Selenium 2 code](#)
 - 7.7.4: [Selenium's Continuous Integration Implementation](#)
 - 7.7.5: [Google Summer of Code 2011](#)
 - 7.7.6: [Developer Tips](#)
 - 7.7.7: [Snapshot of Roadmaps for Selenium Releases](#)
- 8: [About this documentation](#)
 - 8.1: [Copyright and attributions](#)
 - 8.2: [Contributing to the Selenium site & documentation](#)
 - 8.3: [Style guide for Selenium documentation](#)
 - 8.4: [Musings about how things came to be](#)

Selenium is an umbrella project for a range of tools and libraries that enable and support the automation of web browsers.

It provides extensions to emulate user interaction with browsers, a distribution server for scaling browser allocation, and the infrastructure for implementations of the [W3C WebDriver specification](#) that lets you write interchangeable code for all major web browsers.

This project is made possible by volunteer contributors who have put in thousands of hours of their own time, and made the source code [freely available](#) for anyone to use, enjoy, and improve.

Selenium brings together browser vendors, engineers, and enthusiasts to further an open discussion around automation of the web platform. The project organises [an annual conference](#) to teach and nurture the community.

At the core of Selenium is [WebDriver](#), an interface to write instruction sets that can be run interchangeably in many browsers. Once you've installed everything, only a few lines of code get you inside a browser. You can find a more comprehensive example in [Writing your first Selenium script](#)

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
from selenium import webdriver

driver = webdriver.Chrome()

driver.get("http://selenium.dev")

driver.quit()
```

 [Check code on GitHub](#)

```
require 'selenium-webdriver'

driver = Selenium::WebDriver.for :chrome
```

```
driver.get 'https://selenium.dev'  
  
driver.quit
```

 [Check code on GitHub](#)

See the [Overview](#) to check the different project components and decide if Selenium is the right tool for you.

You should continue on to [Getting Started](#) to understand how you can install Selenium and successfully use it as a test automation tool, and scaling simple tests like this to run in large, distributed environments on multiple browsers, on several different operating systems.

1 - Selenium overview

Is Selenium for you? See an overview of the different project components.

Selenium is not just one tool or API but it composes many tools.

WebDriver

If you are beginning with desktop website or mobile website test automation, then you are going to be using WebDriver APIs. [WebDriver](#) uses browser automation APIs provided by browser vendors to control browser and run tests. This is as if a real user is operating the browser. Since WebDriver does not require its API to be compiled with application code, it is not intrusive. Hence, you are testing the same application which you push live.

IDE

[IDE](#) (Integrated Development Environment) is the tool you use to develop your Selenium test cases. It's an easy-to-use Chrome and Firefox extension and is generally the most efficient way to develop test cases. It records the users' actions in the browser for you, using existing Selenium commands, with parameters defined by the context of that element. This is not only a time-saver but also an excellent way of learning Selenium script syntax.

Grid

Selenium Grid allows you to run test cases in different machines across different platforms. The control of triggering the test cases is on the local end, and when the test cases are triggered, they are automatically executed by the remote end.

After the development of the WebDriver tests, you may face the need of running your tests on multiple browser and operating system combinations. This is where [Grid](#) comes into the picture.

1.1 - Selenium components

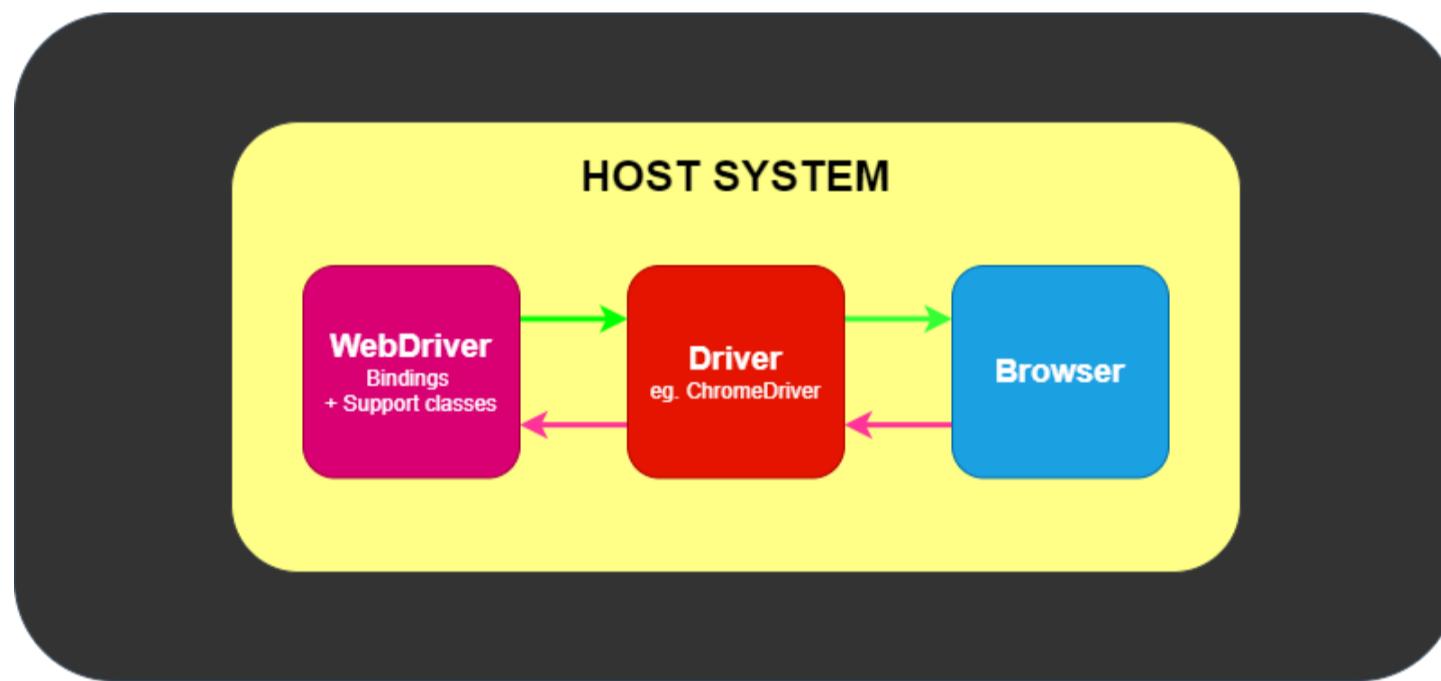
Building a test suite using WebDriver will require you to understand and effectively use a number of different components. As with everything in software, different people use different terms for the same idea. Below is a breakdown of how terms are used in this description.

Terminology

- **API:** Application Programming Interface. This is the set of “commands” you use to manipulate WebDriver.
- **Library:** A code module which contains the APIs and the code necessary to implement them. Libraries are specific to each language binding, eg .jar files for Java, .dll files for .NET, etc.
- **Driver:** Responsible for controlling the actual browser. Most drivers are created by the browser vendors themselves. Drivers are generally executable modules that run on the system with the browser itself, not on the system executing the test suite. (Although those may be the same system.) NOTE: *Some people refer to the drivers as proxies.*
- **Framework:** An additional library used as a support for WebDriver suites. These frameworks may be test frameworks such as JUnit or NUnit. They may also be frameworks supporting natural language features such as Cucumber or Robotium. Frameworks may also be written and used for things such as manipulating or configuring the system under test, data creation, test oracles, etc.

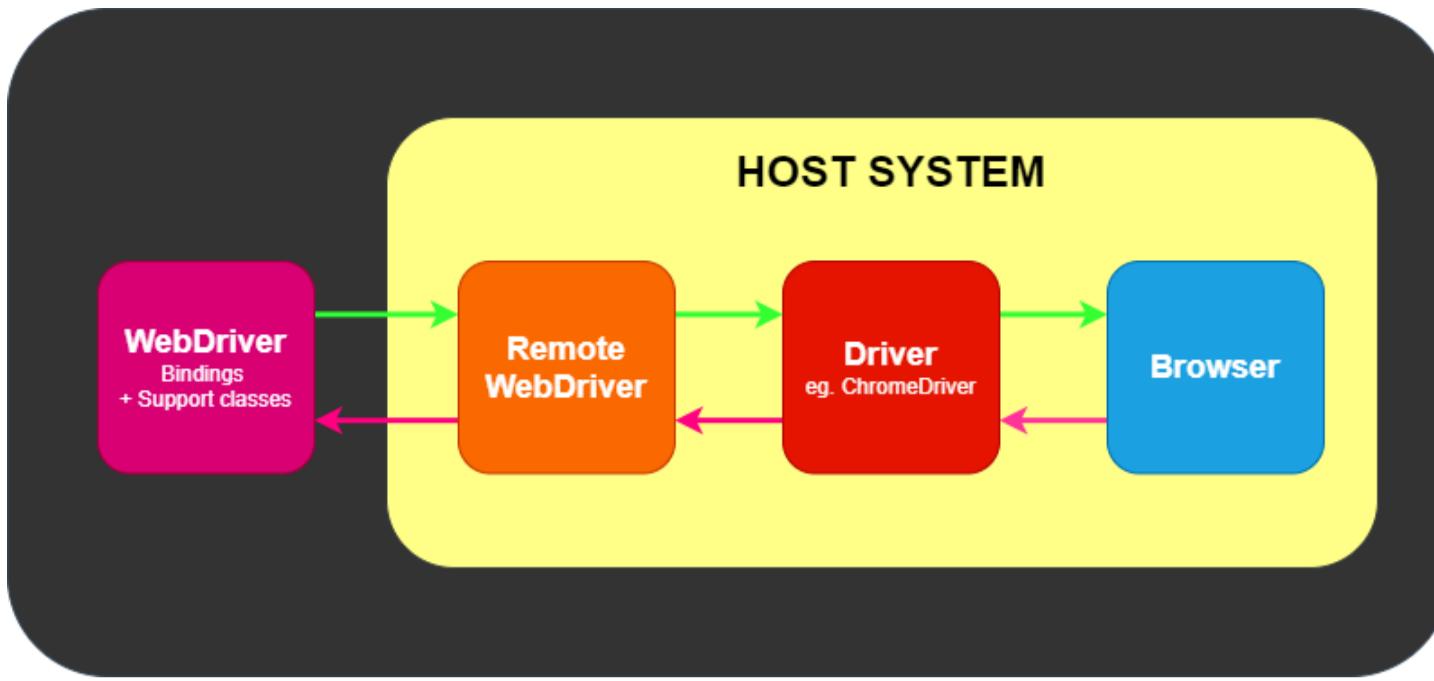
The Parts and Pieces

At its minimum, WebDriver talks to a browser through a driver. Communication is two way: WebDriver passes commands to the browser through the driver, and receives information back via the same route.

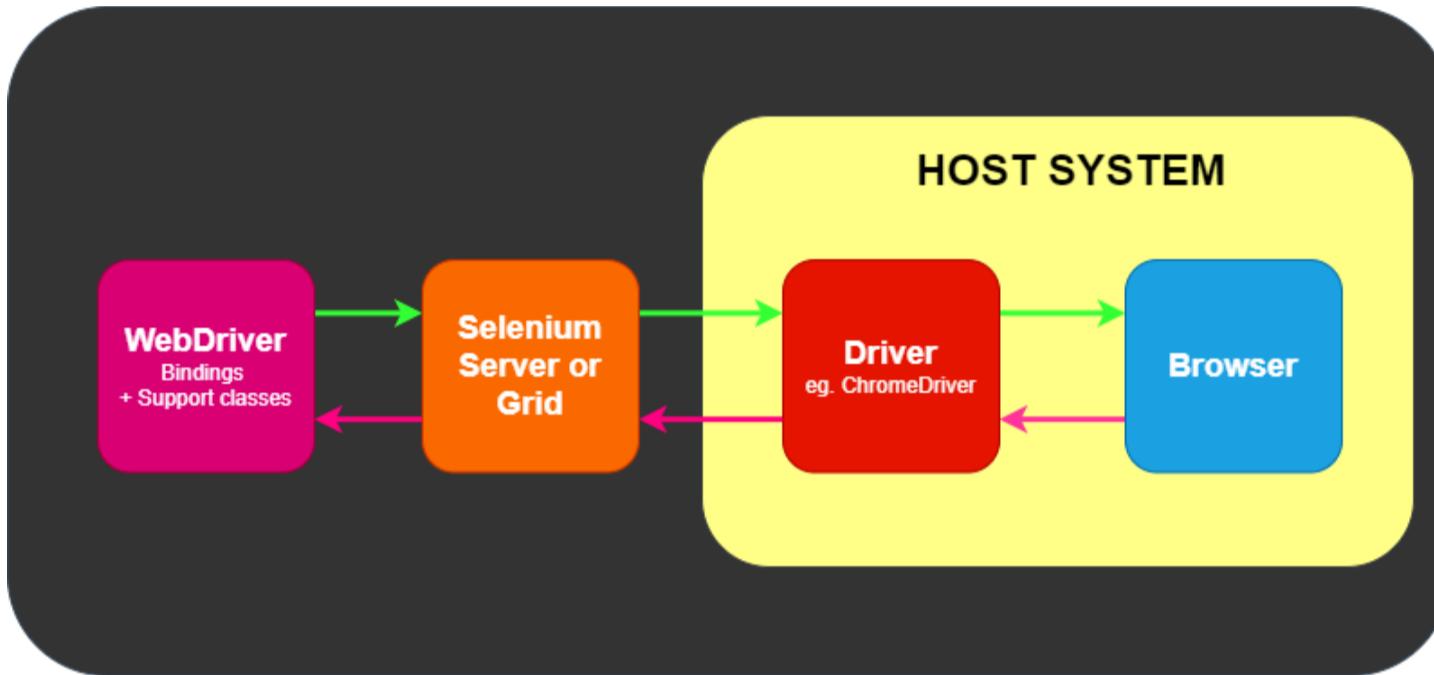


The driver is specific to the browser, such as ChromeDriver for Google's Chrome/Chromium, GeckoDriver for Mozilla's Firefox, etc. The driver runs on the same system as the browser. This may, or may not be, the same system where the tests themselves are executing.

This simple example above is *direct* communication. Communication to the browser may also be *remote* communication through Selenium Server or RemoteWebDriver. RemoteWebDriver runs on the same system as the driver and the browser.



Remote communication can also take place using Selenium Server or Selenium Grid, both of which in turn talk to the driver on the host system

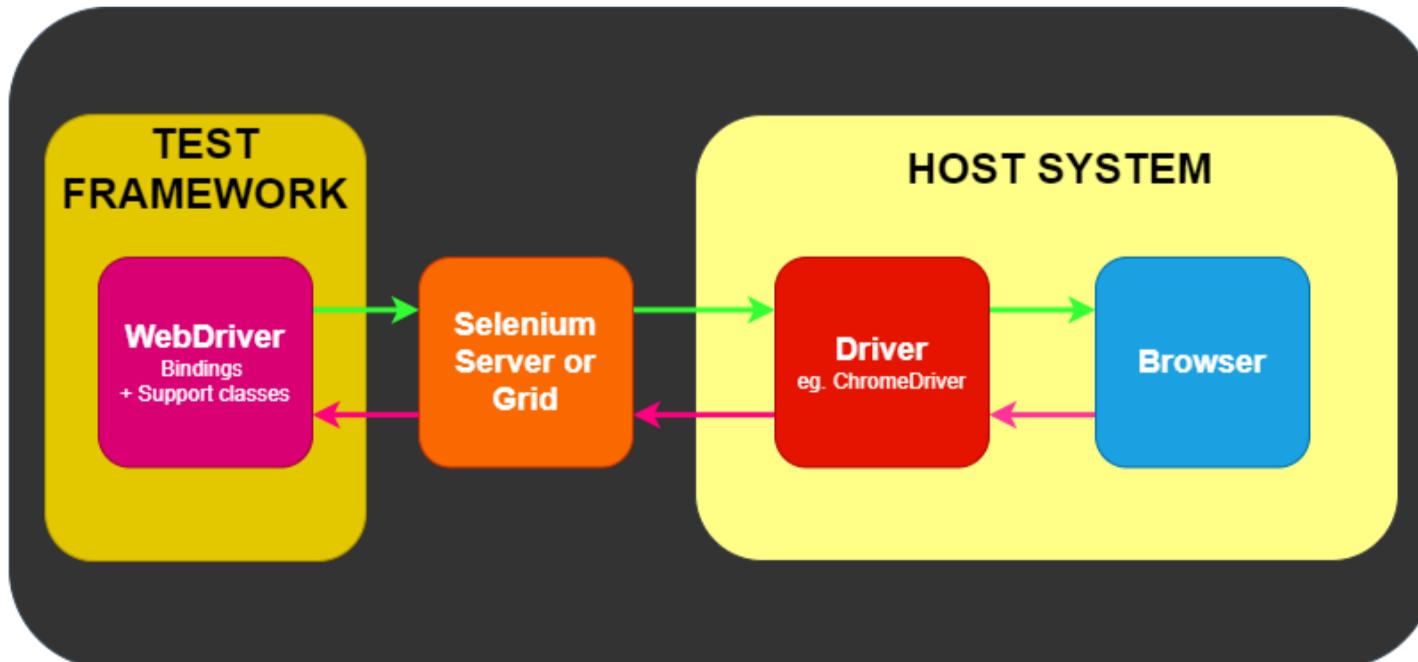


Where Frameworks fit in

WebDriver has one job and one job only: communicate with the browser via any of the methods above. WebDriver does not know a thing about testing: it does not know how to compare things, assert pass or fail, and it certainly does not know a thing about reporting or Given/When/Then grammar.

This is where various frameworks come in to play. At a minimum you will need a test framework that matches the language bindings, e.g. NUnit for .NET, JUnit for Java, RSpec for Ruby, etc.

The test framework is responsible for running and executing your WebDriver and related steps in your tests. As such, you can think of it looking akin to the following image.



Natural language frameworks/tools such as Cucumber may exist as part of that Test Framework box in the figure above, or they may wrap the Test Framework entirely in their own implementation.

1.2 - A deeper look at Selenium

Selenium is an umbrella project for a range of tools and libraries that enable and support the automation of web browsers.

Selenium controls web browsers

Selenium is many things but at its core, it is a toolset for web browser automation that uses the best techniques available to remotely control browser instances and emulate a user's interaction with the browser.

It allows users to simulate common activities performed by end-users; entering text into fields, selecting drop-down values and checking boxes, and clicking links in documents. It also provides many other controls such as mouse movement, arbitrary JavaScript execution, and much more.

Although used primarily for front-end testing of websites, Selenium is at its core a browser user agent *library*. The interfaces are ubiquitous to their application, which encourages composition with other libraries to suit your purpose.

One interface to rule them all

One of the project's guiding principles is to support a common interface for all (major) browser technologies. Web browsers are incredibly complex, highly engineered applications, performing their operations in completely different ways but which frequently look the same while doing so. Even though the text is rendered in the same fonts, the images are displayed in the same place and the links take you to the same destination. What is happening underneath is as different as night and day. Selenium "abstracts" these differences, hiding their details and intricacies from the person writing the code. This allows you to write several lines of code to perform a complicated workflow, but these same lines will execute on Firefox, Internet Explorer, Chrome, and all other supported browsers.

Tools and support

Selenium's minimalist design approach gives it the versatility to be included as a component in bigger applications. The surrounding infrastructure provided under the Selenium umbrella gives you the tools to put together your [grid of browsers](#) so tests can be run on different browsers and multiple operating systems across a range of machines.

Imagine a bank of computers in your server room or data center all firing up browsers at the same time hitting your site's links, forms, and tables—testing your application 24 hours a day. Through the simple programming interface provided for the most common languages, these tests will run tirelessly in parallel, reporting back to you when errors occur.

It is an aim to help make this a reality for you, by providing users with tools and documentation to not only control browsers but to make it easy to scale and deploy such grids.

Who uses Selenium

Many of the most important companies in the world have adopted Selenium for their browser-based testing, often replacing years-long efforts involving other proprietary tools. As it has grown in popularity, so have its requirements and challenges multiplied.

As the web becomes more complicated and new technologies are added to websites, it's the mission of this project to keep up with them where possible. Being an open source project, this support is provided through the generous donation of time from many volunteers, every one of which has a "day job".

Another mission of the project is to encourage more volunteers to partake in this effort, and build a strong community so that the project can continue to keep up with emerging technologies and remain a dominant platform for functional test automation.

2 - WebDriver

WebDriver drives a browser natively, learn more about it.

WebDriver drives a browser natively, as a user would, either locally or on a remote machine using the Selenium server, marks a leap forward in terms of browser automation.

Selenium WebDriver refers to both the language bindings and the implementations of the individual browser controlling code. This is commonly referred to as just *WebDriver*.

Selenium WebDriver is a [W3C Recommendation](#)

- WebDriver is designed as a simple and more concise programming interface.
- WebDriver is a compact object-oriented API.
- It drives the browser effectively.

2.1 - Getting started

If you are new to Selenium, we have a few resources that can help you get up to speed right away.

Selenium supports automation of all the major browsers in the market through the use of *WebDriver*. WebDriver is an API and protocol that defines a language-neutral interface for controlling the behaviour of web browsers. Each browser is backed by a specific WebDriver implementation, called a *driver*. The driver is the component responsible for delegating down to the browser, and handles communication to and from Selenium and the browser.

This separation is part of a conscious effort to have browser vendors take responsibility for the implementation for their browsers. Selenium makes use of these third party drivers where possible, but also provides its own drivers maintained by the project for the cases when this is not a reality.

The Selenium framework ties all of these pieces together through a user-facing interface that enables the different browser backends to be used transparently, enabling cross-browser and cross-platform automation.

Selenium setup is quite different from the setup of other commercial tools. Before you can start writing Selenium code, you have to install the language bindings libraries for your language of choice, the browser you want to use, and the driver for that browser.

Follow the links below to get up and going with Selenium WebDriver.

If you wish to start with a low-code/record and playback tool, please check [Selenium IDE](#)

Once you get things working, if you want to scale up your tests, check out the [Selenium Grid](#).

2.1.1 - Install a Selenium library

Setting up the Selenium library for your favourite programming language.

First you need to install the Selenium bindings for your automation project. The installation process for libraries depends on the language you choose to use. Make sure you check the [Selenium downloads page](#) to make sure you are using the latest version.

Requirements by language

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

Installation of Selenium libraries for Python can be done using pip:

```
pip install selenium
```

Alternatively you can download the [PyPI source archive](#) (selenium-x.x.x.tar.gz) and install it using *setup.py*.

```
python setup.py install
```

Installation of Selenium libraries for JavaScript can be done using npm:

```
npm install selenium-webdriver
```

Next Step

[Install the browser drivers](#)

2.1.2 - Install browser drivers

Setting up your system to allow a browser to be automated.

Through WebDriver, Selenium supports all major browsers on the market such as Chrome/Chromium, Firefox, Internet Explorer, Edge, Opera, and Safari. Where possible, WebDriver drives the browser using the browser's built-in support for automation.

Since all the driver implementations except for Internet Explorer are provided by the browser vendors themselves, they are not included in the standard Selenium distribution. This section explains the basic requirements for getting you started with the different browsers.

Read about more advanced options for starting a driver in our [driver configuration](#) documentation.

Quick Reference

Browser	Supported OS	Maintained by	Download	Issue Tracker
Chromium/Chrome	Windows/macOS/Linux	Google	Downloads	Issues
Firefox	Windows/macOS/Linux	Mozilla	Downloads	Issues
Edge	Windows/macOS	Microsoft	Downloads	Issues
Internet Explorer	Windows	Selenium Project	Downloads	Issues
Safari	macOS High Sierra and newer	Apple	Built in	Issues

Note: The Opera driver does not support w3c syntax, so we recommend using chromedriver to work with Opera. See the code example for [opening an Opera browser](#).

Three Ways to Use Drivers

1. Driver Management Software

Most machines automatically update the browser, but the driver does not. To make sure you get the correct driver for your browser, there are many third party libraries to assist you.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

1. Import [WebDriver Manager for Python](#)

```
from webdriver_manager.chrome import ChromeDriverManager
```

2. Use `install()` to get the location used by the manager and pass it into service class

```
service = Service(executable_path=ChromeDriverManager().install())
```

3. Use `Service` instance when initializing the driver:

```
driver = webdriver.Chrome(service=service)
```

[See full example on GitHub.](#)

2. The PATH Environment Variable

This option first requires manually downloading the driver (See [Quick Reference Section](#) for links).

This is a flexible option to change location of drivers without having to update your code, and will work on multiple machines without requiring that each machine put the drivers in the same place.

You can either place the drivers in a directory that is already listed in `PATH`, or you can place them in a directory and add it to `PATH`.

[Bash](#) [Zsh](#) [Windows](#)

To see what directories are already on `PATH`, open a Terminal and execute:

```
echo $PATH
```

If the location to your driver is not already in a directory listed, you can add a new directory to `PATH`:

```
echo 'export PATH=$PATH:/path/to/driver' >> ~/.bash_profile
source ~/.bash_profile
```

You can test if it has been added correctly by starting the driver:

```
chromedriver
```

If your `PATH` is configured correctly above, you will see some output relating to the startup of the driver:

```
Starting ChromeDriver 95.0.4638.54 (d31a821ec901f68d0d34ccdbaea45b4c86ce543e-refs/branch-heads/95.0@{#54})
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
```

You can regain control of your command prompt by pressing `Ctrl+C`

3. Hard Coded Location

Similar to Option 2 above, you need to manually download the driver (See [Quick Reference Section](#) for links). Specifying the location in the code itself has the advantage of not needing to figure out Environment Variables on your system, but has the drawback of making the code much less flexible.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
from selenium.webdriver.chrome.service import Service
from selenium import webdriver

service = Service(executable_path="/path/to/chromedriver")
driver = webdriver.Chrome(service=service)
```

Advanced Configuration

More information on how you can change the driver behavior can be found on the [Configuring driver parameters](#) page.

Next Step

[Open and close a browser](#)

2.1.3 - Open and close a browser with Selenium

Code examples for starting and stopping a session with each browser.

Once you have a [Selenium library installed](#), and your [desired browser driver](#), you can start and stop a session with a browser.

Typically, browsers are started with specific options that describe which capabilities the browser must support, and how the browser should behave during the session. Some capabilities are [shared by all browsers](#), and some will be specific to the browser being used. This page will show examples of starting a browser with the default capabilities.

After learning how to start a session, check out the next session on how to [write your first Selenium script](#)

Chrome

By default, Selenium 4 is compatible with Chrome v75 and greater. Note that the version of the Chrome browser and the version of chromedriver must match the major version.

In addition to the [shared capabilities](#), there are specific [Chrome capabilities](#) that can be used.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
options = ChromeOptions()
driver = webdriver.Chrome(options=options)

driver.quit()
```

Edge

Microsoft Edge is implemented with Chromium, with the earliest supported version of v79. Similar to Chrome, the major version number of edgedriver must match the major version of the Edge browser.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
options = EdgeOptions()
driver = webdriver.Edge(options=options)

driver.quit()
```

Firefox

Selenium 4 requires Firefox 78 or greater. It is recommended to always use the latest version of geckodriver.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
options = FirefoxOptions()
driver = webdriver.Firefox(options=options)

driver.quit()
```

Internet Explorer

The IE Driver is the only driver maintained by the Selenium Project directly. While binaries for both the 32-bit and 64-bit versions of Internet Explorer are available, there are some [limitations](#) with the 64-bit driver. As such it is recommended to use the 32-bit driver.

Legacy

The Selenium project aims to support the same releases that [Microsoft considers current](#). Older releases may work, but will not be supported. Note that Internet Explorer 11 will end support for certain operating systems, including Windows 10 on June 15, 2022.

It should be noted that as Internet Explorer preferences are saved against the logged-in user's account, some additional setup is required.

Additional information about using Internet Explorer can be found [on the Selenium wiki](#)

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
options = IEOPTIONS()
driver = webdriver.Ie(options=options)

driver.quit()
```

Compatibility Mode

Microsoft Edge can be used in IE compatibility mode using the IE Driver.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
options = IEOPTIONS()
options.attach_to_edge_chrome = True
options.edge_executable_path = "/path/to/edge/browser"
driver = webdriver.Ie(options=options)

driver.quit()
```

Opera

Since the opera driver does not set w3c as default value, but is based on Chromium, it is recommended to drive Opera browser with the chromedriver. Like all Chromium implementations, make sure that the browser version matches the driver version.

[Java](#)[Python](#)[CSharp](#)[Ruby](#)[JavaScript](#)[Kotlin](#)

```
options = ChromeOptions()
options.binary_location = "path/to/opera/browser"
driver = webdriver.Chrome(options=options)

driver.quit()
```

Safari

Desktop

Unlike Chromium and Firefox drivers, the safaridriver is installed with the Operating System. To enable automation on Safari, run the following command from the terminal:

```
safaridriver --enable
```

[Java](#)[Python](#)[CSharp](#)[Ruby](#)[JavaScript](#)[Kotlin](#)

```
driver = webdriver.Safari()

driver.quit()
```

Mobile

Those looking to automate Safari on iOS should look to the [Appium project](#).

Next Step

[Create your first Selenium script](#)

2.1.4 - Write your first Selenium script

Step-by-step instructions for constructing a Selenium script

Once you have [Selenium installed](#) and [Drivers installed](#), you're ready to write Selenium code.

Eight Basic Components

Everything Selenium does is send the browser commands to do something or send requests for information. Most of what you'll do with Selenium is a combination of these basic commands:

1. Start the session

For more details on starting a session read our documentation on [opening and closing a browser](#)

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver = new ChromeDriver();
```

 [Check code on GitHub](#)

2. Take action on browser

In this example we are [navigating](#) to a web page.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.get("https://duckduckgo.com/");
```

 [Check code on GitHub](#)

3. Request browser information

There are a bunch of types of [information about the browser](#) you can request, including window handles, browser size / position, cookies, alerts, etc.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
String title = driver.getTitle();
```

 [Check code on GitHub](#)

4. Establish Waiting Strategy

Synchronizing the code with the current state of the browser is one of the biggest challenges with Selenium, and doing it well is an advanced topic.

Essentially you want to make sure that the element is on the page before you attempt to locate it and the element is in an interactable state before you attempt to interact with it.

An implicit wait is rarely the best solution, but it's the easiest to demonstrate here, so we'll use it as a placeholder.

Read more about [Waiting strategies](#).

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));
```

 [Check code on GitHub](#)

5. Find an element

The majority of commands in most Selenium sessions are element related, and you can't interact with one without first [finding an element](#)

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement searchBox = driver.findElement(By.name("q"));
WebElement searchButton = driver.findElement(By.id("search_button_homepage"));
```

 [Check code on GitHub](#)

6. Take action on element

There are only a handful of [actions to take on an element](#), but you will use them frequently.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
search_box.sendKeys("Selenium")
search_button.click()
```

 [Check code on GitHub](#)

7. Request element information

Elements store a lot of [information that can be requested](#). Notice that we need to relocate the search box because the DOM has changed since we first located it.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
value = search_box.getAttribute("value")
```

 [Check code on GitHub](#)

8. End the session

This ends the driver process, which by default closes the browser as well. No more commands can be sent to this driver instance.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.quit()
```

 [Check code on GitHub](#)

Putting everything together

Let's combine these 8 things into a complete script with assertions that can be executed by a test runner.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service as ChromeService
from webdriver_manager.chrome import ChromeDriverManager

def test_eight_components():
    driver = webdriver.Chrome(service=ChromeService(executable_path=ChromeDriverManager().install()))

    driver.get("https://duckduckgo.com/")

    title = driver.title
    assert "DuckDuckGo" in title

    driver.implicitly_wait(10)

    search_box = driver.find_element(by=By.NAME, value="q")
    search_button = driver.find_element(by=By.ID, value="search_button_homepage")

    search_box.send_keys("Selenium")
    search_button.click()

    search_box = driver.find_element(by=By.NAME, value="q")
    value = search_box.get_attribute("value")
    assert value == "Selenium"

    driver.quit()
```

 [Check code on GitHub](#)

Next Steps

Take what you've learned and build out your Selenium code.

As you find more functionality that you need, read up on the rest of our [WebDriver documentation](#).

2.1.5 - Upgrade to Selenium 4

Are you still using Selenium 3? This guide will help you upgrade to the latest release!

Upgrading to Selenium 4 should be a painless process if you are using one of the officially supported languages (Ruby, JavaScript, C#, Python, and Java). There might be some cases where a few issues can happen, and this guide will help you to sort them out. We will go through the steps to upgrade your project dependencies and understand the major deprecations and changes the version upgrade brings.

These are the steps we will follow to upgrade to Selenium 4:

- Preparing our test code
- Upgrading dependencies
- Potential errors and deprecation messages

Note: while Selenium 3.x versions were being developed, support for the W3C WebDriver standard was implemented. Both this new protocol and the legacy JSON Wire Protocol were supported. Around version 3.11, Selenium code became compliant with the level W3C 1 specification. The W3C compliant code in the latest version of Selenium 3 will work as expected in Selenium 4.

Preparing our test code

Selenium 4 removes support for the legacy protocol and uses the W3C WebDriver standard by default under the hood. For most things, this implementation will not affect end users. The major exceptions are `Capabilities` and the `Actions` class.

Capabilities

If the test capabilities are not structured to be W3C compliant, may cause a session to not be started.

Here is the list of W3C WebDriver standard capabilities:

- `browserName`
- `browserVersion` (replaces `version`)
- `platformName` (replaces `platform`)
- `acceptInsecureCerts`
- `pageLoadStrategy`
- `proxy`
- `timeouts`
- `unhandledPromptBehavior`

An up-to-date list of standard capabilities can be found at [W3C WebDriver](#).

Any capability that is not contained in the list above, needs to include a vendor prefix. This applies to browser specific capabilities as well as cloud vendor specific capabilities. For example, if your cloud vendor uses `build` and `name` capabilities for your tests, you need to wrap them in a `cloud:options` block (check with your cloud vendor for the appropriate prefix).

Before

[Java](#) [JavaScript](#) [CSharp](#) [Ruby](#) [Python](#)

```
DesiredCapabilities caps = DesiredCapabilities.firefox();
caps.setCapability("platform", "Windows 10");
caps.setCapability("version", "92");
caps.setCapability("build", myTestBuild);
caps.setCapability("name", myTestName);
WebDriver driver = new RemoteWebDriver(new URL(cloudUrl), caps);
```

After

[Java](#) [JavaScript](#) [CSharp](#) [Ruby](#) [Python](#)

```
capabilities = {
    browserName: 'firefox',
    browserVersion: '92',
    platformName: 'Windows 10',
    'cloud:options': {
        build: myTestBuild,
        name: myTestName,
    }
}
```

```
from selenium.webdriver.firefox.options import Options as FirefoxOptions
options = FirefoxOptions()
options.browser_version = '92'
options.platform_name = 'Windows 10'
cloud_options = {}
cloud_options['build'] = my_test_build
cloud_options['name'] = my_test_name
options.set_capability('cloud:options', cloud_options)
driver = webdriver.Remote(cloud_url, options=options)
```

Find element(s) utility methods in Java

The utility methods to find elements in the Java bindings (`FindsBy` interfaces) have been removed as they were meant for internal use only. The following code samples explain this better.

Finding a single element with `findElement*`

Before	After
<pre>driver.findElementByName("ele") driver.findElementById("ele") driver.findElementByClassName("ele") driver.findElementByLinkText("ele") driver.findElementByPartialLinkText("ele") driver.findElementByTagName("ele") driver.findElementByXPath("ele")</pre>	<pre>driver.findElement(By.name("ele")) driver.findElement(By.id("ele")) driver.findElement(By.className("ele")) driver.findElement(By.linkText("ele")) driver.findElement(By.partialLinkText("ele")) driver.findElement(By.tagName("ele")) driver.findElement(By.xpath("ele"))</pre>

Finding a multiple elements with `findElements*`

Before	After

```
driver.findElementsByClassName  
driver.findElementsByCssSelect  
driver.findElementsById("elem  
driver.findElementsByLinkText  
driver.findElementsByName("el  
driver.findElementsByPartialL  
driver.findElementsByTagName(  
driver.findElementsByXPath("
```

```
driver.findElements(By.className  
driver.findElements(By.cssSel  
driver.findElements(By.id("el  
driver.findElements(By.linkTe  
driver.findElements(By.name("el  
driver.findElements(By.partialL  
driver.findElements(By.tagName  
driver.findElements(By.xpath("
```

Upgrading dependencies

Check the subsections below to install Selenium 4 and have your project dependencies upgraded.

Java

The process of upgrading Selenium depends on which build tool is being used. We will cover the most common ones for Java, which are [Maven](#) and [Gradle](#). The minimum Java version required is still 8.

Maven

Before	After
<pre><dependencies> <!-- more dependencies ... <dependency> <groupId>org.seleniumhq.s <artifactId>selenium-java <version>3.141.59</versio </dependency> <!-- more dependencies ... </dependencies></pre>	<pre><dependencies> <!-- more dependencies ... <dependency> <groupId>org.selenium <artifactId>selenium- <version>4.0.0</versi </dependency> <!-- more dependencies ... </dependencies></pre>

After making the change, you could execute `mvn clean compile` on the same directory where the `pom.xml` file is.

Gradle

Before	After

```
plugins {
    id 'java'
}
group 'org.example'
version '1.0-SNAPSHOT'
repositories {
    mavenCentral()
}
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
    implementation group: 'org.seleniumhq.selenium', name: 'selenium-java', version: '4.0.0'
}
test {
    useJUnitPlatform()
}
```

```
plugins {
    id 'java'
}
group 'org.example'
version '1.0-SNAPSHOT'
repositories {
    mavenCentral()
}
dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.0'
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.0'
    implementation group: 'org.seleniumhq.selenium', name: 'selenium-java', version: '4.0.0'
}
test {
    useJUnitPlatform()
}
```

After making the change, you could execute `./gradlew clean build` on the same directory where the `build.gradle` file is.

To check all the Java releases, you can head to [MVNRepository](#).

C#

The place to get updates for Selenium 4 in C# is [NuGet](#). Under the [Selenium.WebDriver](#) package you can get the instructions to update to the latest version. Inside of Visual Studio, through the NuGet Package Manager you can execute:

```
PM> Install-Package Selenium.WebDriver -Version 4.0.0
```

Python

The most important change to use Python is the minimum required version. Selenium 4 will require a minimum Python 3.7 or higher. More details can be found at the [Python Package Index](#). To upgrade from the command line, you can execute:

```
pip install selenium==4.0.0
```

Ruby

The update details for Selenium 4 can be seen at the [selenium-webdriver](#) gem in RubyGems. To install the latest version, you can execute:

```
gem install selenium-webdriver
```

To add it to your Gemfile:

```
gem 'selenium-webdriver', '~> 4.0.0'
```

JavaScript

The selenium-webdriver package can be found at the Node package manager, [npmjs](#). Selenium 4 can be found [here](#). To install it, you could either execute:

```
npm install selenium-webdriver
```

Or, update your package.json and run `npm install`:

```
{
  "name": "selenium-tests",
  "version": "1.0.0",
  "dependencies": {
    "selenium-webdriver": "^4.0.0"
  }
}
```

Potential errors and deprecation messages

Here is a set of code examples that will help to overcome the deprecation messages you might encounter after upgrading to Selenium 4.

Java

Waits and Timeout

The parameters received in Timeout have switched from expecting `(long time, TimeUnit unit)` to expect `(Duration duration)`.

Before	After
<pre>driver.manage().timeouts().im driver.manage().timeouts().se driver.manage().timeouts().pa</pre>	<pre>driver.manage().timeouts().im driver.manage().timeouts().sc driver.manage().timeouts().pa</pre>

Waits are also expecting different parameters now. `WebDriverWait` is now expecting a `Duration` instead of a `long` for timeout in seconds and milliseconds. The `withTimeout` and `pollingEvery` utility methods from `FluentWait` have switched from expecting `(long time, TimeUnit unit)` to expect `(Duration duration)`.

Before	After
<pre>new WebDriverWait(driver, 3) .until(ExpectedConditions.ele Wait<WebDriver> wait = new Fl .withTimeout(30, TimeUnit.S</pre>	<pre>new WebDriverWait(driver, Dur .until(ExpectedConditions.e Wait<WebDriver> wait = new .withTimeout(Duration.ofSec</pre>

```
.pollingEvery(5, TimeUnit.S
.ignoring(NoSuchElementException
```

```
.pollingEvery(Duration.ofSe
.ignoring(NoSuchElementException
```

Merging capabilities is no longer changing the calling object

It was possible to merge a different set of capabilities into another set, and it was mutating the calling object. Now, the result of the merge operation needs to be assigned.

Before

```
MutableCapabilities capability
capabilities.setCapability("p
FirefoxOptions options = new
options.setHeadless(true);
options.merge(capabilities);
```

As a result, the `options` object was getting modified.

After

```
MutableCapabilities capability
capabilities.setCapability("p
FirefoxOptions options = new
options.setHeadless(true);
options = options.merge(capab
```

The result of the `merge` call needs to be assigned to an object.

Firefox Legacy

Before GeckoDriver was around, the Selenium project had a driver implementation to automate Firefox (version <48). However, this implementation is not needed anymore as it does not work in recent versions of Firefox. To avoid major issues when upgrading to Selenium 4, the `setLegacy` option will be shown as deprecated. The recommendation is to stop using the old implementation and rely only on GeckoDriver. The following code will show the `setLegacy` line deprecated after upgrading.

```
FirefoxOptions options = new FirefoxOptions();
options.setLegacy(true);
```

BrowserType

The `BrowserType` interface has been around for a long time, however it is getting deprecated in favour of the new `Browser` interface.

Before

```
MutableCapabilities capability
capabilities.setCapability("b
capabilities.setCapability("b
```

After

```
MutableCapabilities capability
capabilities.setCapability("b
capabilities.setCapability("b
```

AddAdditionalCapability is deprecated

Instead of it, `AddAdditionalOption` is recommended. Here is an example showing this:

Before	After
<pre>var browserOptions = new Chro browserOptions.PlatformName = browserOptions.BrowserVersion var cloudOptions = new Dictio browserOptions.AddAdditionalC</pre>	<pre>var browserOptions = new Chro browserOptions.PlatformName = browserOptions.BrowserVersion var cloudOptions = new Dictio browserOptions.AddAdditionalO</pre>

Python

`executable_path` has been deprecated, please pass in a Service object

In Selenium 4, you'll need to set the driver's `executable_path` from a Service object to prevent deprecation warnings. (Or don't set the path and instead make sure that the driver you need is on the System PATH.)

Before	After
<pre>from selenium import webdrive options = webdriver.ChromeOpt options.add_experimental_opti options.add_experimental_opti driver = webdriver.Chrome(exe</pre>	<pre>from selenium import webdrive from selenium.webdriver.chrom options = webdriver.ChromeOpt options.add_experimental_opti options.add_experimental_opti service = ChromeService(execu driver = webdriver.Chrome(ser</pre>

Summary

We went through the major changes to be taken into consideration when upgrading to Selenium 4. Covering the different aspects to cover when test code is prepared for the upgrade, including suggestions on how to prevent potential issues that can show up when using the new version of Selenium. To finalize, we also covered a set of possible issues that you can bump into after upgrading, and we shared potential fixes for those issues.

This was originally posted at <https://saucelabs.com/resources/articles/how-to-upgrade-to-selenium-4>

2.2 - WebDriver Capabilities

2.2.1 - Shared capabilities

These capabilities are shared by all browsers.

In order to create a new session by Selenium WebDriver, the local end should provide the basic capabilities to the remote end. The remote end uses the same set of capabilities to create a session and describes the current session features.

WebDriver provides capabilities that each remote end will/should support the implementation. The following capabilities are supported by WebDriver:

browserName:

This capability is used to set the `browserName` for a given session. If the specified browser is not installed at the remote end, the session creation will fail.

browserVersion:

This capability is optional, this is used to set the available browser version at remote end. For Example, if ask for Chrome version 75 on a system that only has 80 installed, the session creation will fail.

pageLoadStrategy:

When navigating to a new page via URL, by default Selenium will wait until the Document's Ready State is "complete." The `document.readyState` property of a document describes the loading state of the current document. By default, WebDriver will hold off on completing a navigation method (e.g., `driver.navigate().get()`) until the document ready state is `complete`. This does not necessarily mean that the page has finished loading, especially for sites like Single Page Applications that use a lot of JavaScript to dynamically load content after the Ready State returns complete. Note also that this behavior does not apply to navigation that is a result of clicking an element or submitting a form.

If a page takes a long time to load as a result of downloading assets (e.g., images, css, js) that aren't important to the automation, you can change from the default parameter of `normal` to `eager` or `none` to speed up the session. This value applies to the entire session, so make sure that your [waiting strategy](#) is sufficient to minimize flakiness.

The page load strategy queries the [`document.readyState`](#) as described in the table below:

Strategy	State	Ready	Notes
normal	complete	Used by default, waits for all resources to download	
eager	interactive	DOM access is ready, but other resources like images may still be loading	
none	Any	Does not block WebDriver at all	

normal

This will make Selenium WebDriver to wait for the entire page is loaded. When set to `normal`, Selenium WebDriver waits until the [`load`](#) event fire is returned.

By default `normal` is set to browser if none is provided.

```
import org.openqa.selenium.PageLoadStrategy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.chrome.ChromeDriver;

public class pageLoadStrategy {
    public static void main(String[] args) {
        ChromeOptions chromeOptions = new ChromeOptions();
        chromeOptions.setPageLoadStrategy(PageLoadStrategy.NORMAL);
        WebDriver driver = new ChromeDriver(chromeOptions);
        try {
            // Navigate to Url
            driver.get("https://google.com");
        } finally {
            driver.quit();
        }
    }
}
```

eager

This will make Selenium WebDriver to wait until the initial HTML document has been completely loaded and parsed, and discards loading of stylesheets, images and subframes.

When set to **eager**, Selenium WebDriver waits until [DOMContentLoaded](#) event fire is returned.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.PageLoadStrategy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.chrome.ChromeDriver;

public class pageLoadStrategy {
    public static void main(String[] args) {
        ChromeOptions chromeOptions = new ChromeOptions();
        chromeOptions.setPageLoadStrategy(PageLoadStrategy.EAGER);
        WebDriver driver = new ChromeDriver(chromeOptions);
        try {
            // Navigate to Url
            driver.get("https://google.com");
        } finally {
            driver.quit();
        }
    }
}
```

none

When set to **none** Selenium WebDriver only waits until the initial page is downloaded.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```

import org.openqa.selenium.PageLoadStrategy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.chrome.ChromeDriver;

public class pageLoadStrategy {
    public static void main(String[] args) {
        ChromeOptions chromeOptions = new ChromeOptions();
        chromeOptions.setPageLoadStrategy(PageLoadStrategy.NONE);
        WebDriver driver = new ChromeDriver(chromeOptions);
        try {
            // Navigate to Url
            driver.get("https://google.com");
        } finally {
            driver.quit();
        }
    }
}

```

platformName

This identifies the operating system at the remote-end, fetching the `platformName` returns the OS name.

In cloud-based providers, setting `platformName` sets the OS at the remote-end.

acceptInsecureCerts

This capability checks whether an expired (or) invalid `TLS Certificate` is used while navigating during a session.

If the capability is set to `false`, an [insecure certificate error](#) will be returned as navigation encounters any domain certificate problems. If set to `true`, invalid certificate will be trusted by the browser.

All self-signed certificates will be trusted by this capability by default. Once set, `acceptInsecureCerts` capability will have an effect for the entire session.

timeouts

A WebDriver `session` is imposed with a certain `session timeout` interval, during which the user can control the behaviour of executing scripts or retrieving information from the browser.

Each session timeout is configured with combination of different `timeouts` as described below:

Script Timeout:

Specifies when to interrupt an executing script in a current browsing context. The default timeout **30,000** is imposed when a new session is created by WebDriver.

Page Load Timeout:

Specifies the time interval in which web page needs to be loaded in a current browsing context. The default timeout **300,000** is imposed when a new session is created by WebDriver. If page load limits a given/default time frame, the script will be stopped by *TimeoutException*.

Implicit Wait Timeout

This specifies the time to wait for the implicit element location strategy when locating elements. The default timeout **0** is imposed when a new session is created by WebDriver.

unhandledPromptBehavior

Specifies the state of current session's `user prompt handler`. Defaults to **dismiss and notify state**

User Prompt Handler

This defines what action must take when a user prompt encounters at the remote-end. This is defined by `unhandledPromptBehavior` capability and has the following states:

- dismiss
- accept
- dismiss and notify
- accept and notify
- ignore

setWindowRect

Indicates whether the remote end supports all of the [resizing and repositioning commands](#).

strictFileInteractivity

This new capability indicates if strict interactability checks should be applied to `input type=file` elements. As strict interactability checks are off by default, there is a change in behaviour when using `Element Send Keys` with hidden file upload controls.

proxy

A proxy server acts as an intermediary for requests between a client and a server. In simple, the traffic flows through the proxy server on its way to the address you requested and back.

A proxy server for automation scripts with Selenium could be helpful for:

- Capture network traffic
- Mock backend calls made by the website
- Access the required website under complex network topologies or strict corporate restrictions/policies.

If you are in a corporate environment, and a browser fails to connect to a URL, this is most likely because the environment needs a proxy to be accessed.

Selenium WebDriver provides a way to proxy settings:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.Proxy;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;

public class proxyTest {
    public static void main(String[] args) {
        Proxy proxy = new Proxy();
        proxy.setHttpProxy("<HOST:PORT>");
```

```
ChromeOptions options = new ChromeOptions();
options.setCapability("proxy", proxy);
WebDriver driver = new ChromeDriver(options);
driver.get("https://www.google.com/");
driver.manage().window().maximize();
driver.quit();
}
```

2.2.2 - Capabilities specific to Chromium browsers

These capabilities are specific to Chromium based browsers.

These Capabilities apply to:

- Chrome
- Chromium
- Edge

2.2.3 - Capabilities specific to Firefox browser

These capabilities are specific to Firefox.

Define Capabilities using `FirefoxOptions`

`FirefoxOptions` is the new way to define capabilities for the Firefox browser and should generally be used in preference to `DesiredCapabilities`.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
FirefoxOptions options = new FirefoxOptions();
options.addPreference("network.proxy.type", 0);
driver = new RemoteWebDriver(options);
```

Setting a custom profile

It is possible to create a custom profile for Firefox as demonstrated below.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
FirefoxProfile profile = new FirefoxProfile();
FirefoxOptions options = new FirefoxOptions();
options.setProfile(profile);
driver = new RemoteWebDriver(options);
```

2.2.4 - Capabilities specific to Internet Explorer browser

These capabilities are specific to Internet Explorer.

fileUploadDialogTimeout

In some environments, Internet Explorer may timeout when opening the File Upload dialog. IEDriver has a default timeout of 1000ms, but you can increase the timeout using the fileUploadDialogTimeout capability.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
InternetExplorerOptions options = new InternetExplorerOptions();
options.waitForUploadDialogUpTo(Duration.ofSeconds(2));
WebDriver driver = new RemoteWebDriver(options);
```

ensureCleanSession

When set to `true`, this capability clears the *Cache, Browser History and Cookies* for all running instances of InternetExplorer including those started manually or by the driver. By default, it is set to `false`.

Using this capability will cause performance drop while launching the browser, as the driver will wait until the cache gets cleared before launching the IE browser.

This capability accepts a Boolean value as parameter.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
InternetExplorerOptions options = new InternetExplorerOptions();
options.destructivelyEnsureCleanSession();
WebDriver driver = new RemoteWebDriver(options);
```

ignoreZoomSetting

InternetExplorer driver expects the browser zoom level to be 100%, else the driver will throw an exception. This default behaviour can be disabled by setting the `ignoreZoomSetting` to `true`.

This capability accepts a Boolean value as parameter.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
InternetExplorerOptions options = new InternetExplorerOptions();
options.ignoreZoomSettings();
WebDriver driver = new RemoteWebDriver(options);
```

ignoreProtectedModeSettings

Whether to skip the *Protected Mode* check while launching a new IE session.

If not set and *Protected Mode* settings are not same for all zones, an exception will be thrown by the driver.

If capability is set to `true`, tests may become flaky, unresponsive, or browsers may hang. However, this is still by far a second-best choice, and the first choice should *always* be to actually set the Protected Mode settings of each zone manually. If a user is using this property, only a “best effort” at support will be given.

This capability accepts a Boolean value as parameter.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
InternetExplorerOptions options = new InternetExplorerOptions();
options.introduceFlakinessByIgnoringSecurityDomains();
WebDriver driver = new RemoteWebDriver(options);
```

silent

When set to `true`, this capability suppresses the diagnostic output of the IEDriverServer.

This capability accepts a Boolean value as parameter.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
InternetExplorerOptions options = new InternetExplorerOptions();
options.setCapability("silent", true);
WebDriver driver = new InternetExplorerDriver(options);
```

Command-Line Options

Internet Explorer includes several command-line options that enable you to troubleshoot and configure the browser.

The following describes few supported command-line options

- `-private`: Used to start IE in private browsing mode. This works for IE 8 and later versions.
- `-k`: Starts Internet Explorer in kiosk mode. The browser opens in a maximized window that does not display the address bar, the navigation buttons, or the status bar.
- `-extoff`: Starts IE in no add-on mode. This option specifically used to troubleshoot problems with browser add-ons. Works in IE 7 and later versions.

Note: **forceCreateProcessApi** should be enabled in-order for command line arguments to work.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.Capabilities;
import org.openqa.selenium.ie.InternetExplorerDriver;
import org.openqa.selenium.ie.InternetExplorerOptions;
```

```
public class ieTest {
    public static void main(String[] args) {
        InternetExplorerOptions options = new InternetExplorerOptions();
        options.useCreateProcessApiToLaunchIe();
        options.addCommandSwitches("-k");
        InternetExplorerDriver driver = new InternetExplorerDriver(options);
        try {
            driver.get("https://google.com/ncr");
            Capabilities caps = driver.getCapabilities();
            System.out.println(caps);
        } finally {
            driver.quit();
        }
    }
}
```

forceCreateProcessApi

Forces launching Internet Explorer using the CreateProcess API. The default value is false.

For IE 8 and above, this option requires the “TabProcGrowth” registry value to be set to 0.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.Capabilities;
import org.openqa.selenium.ie.InternetExplorerDriver;
import org.openqa.selenium.ie.InternetExplorerOptions;

public class ieTest {
    public static void main(String[] args) {
        InternetExplorerOptions options = new InternetExplorerOptions();
        options.useCreateProcessApiToLaunchIe();
        InternetExplorerDriver driver = new InternetExplorerDriver(options);
        try {
            driver.get("https://google.com/ncr");
            Capabilities caps = driver.getCapabilities();
            System.out.println(caps);
        } finally {
            driver.quit();
        }
    }
}
```

2.2.5 - Capabilities specific to Safari browser

These capabilities are specific to Safari.

2.3 - Browser

Get browser information

Get title

You can read the current page title from the browser:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.getTitle();
```

Get current URL

You can read the current URL from the browser's address bar using:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.getCurrentUrl();
```

2.3.1 - Browser navigation

Navigate to

The first thing you will want to do after launching a browser is to open your website. This can be achieved in a single line:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Convenient  
driver.get("https://selenium.dev");  
  
//Longer way  
driver.navigate().to("https://selenium.dev");
```

Back

Pressing the browser's back button:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.navigate().back();
```

Forward

Pressing the browser's forward button:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.navigate().forward();
```

Refresh

Refresh the current page:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.navigate().refresh();
```

2.3.2 - JavaScript alerts, prompts and confirmations

WebDriver provides an API for working with the three types of native popup messages offered by JavaScript. These popups are styled by the browser and offer limited customisation.

Alerts

The simplest of these is referred to as an alert, which shows a custom message, and a single button which dismisses the alert, labelled in most browsers as OK. It can also be dismissed in most browsers by pressing the close button, but this will always do the same thing as the OK button. [See an example alert.](#)

WebDriver can get the text from the popup and accept or dismiss these alerts.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Click the link to activate the alert
driver.findElement(By.linkText("See an example alert")).click();

//Wait for the alert to be displayed and store it in a variable
Alert alert = wait.until(ExpectedConditions.alertIsPresent());

//Store the alert text in a variable
String text = alert.getText();

//Press the OK button
alert.accept();
```

Confirm

A confirm box is similar to an alert, except the user can also choose to cancel the message. [See a sample confirm.](#)

This example also shows a different approach to storing an alert:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Click the link to activate the alert
driver.findElement(By.linkText("See a sample confirm")).click();

//Wait for the alert to be displayed
wait.until(ExpectedConditions.alertIsPresent());

//Store the alert in a variable
Alert alert = driver.switchTo().alert();

//Store the alert in a variable for reuse
String text = alert.getText();

//Press the Cancel button
alert.dismiss();
```

Prompt

Prompts are similar to confirm boxes, except they also include a text input. Similar to working with form elements, you can use WebDriver's send keys to fill in a response. This will completely replace the placeholder text. Pressing the cancel button will not submit any text. [See a sample prompt.](#)

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Click the link to activate the alert
driver.findElement(By.linkText("See a sample prompt")).click();

//Wait for the alert to be displayed and store it in a variable
Alert alert = wait.until(ExpectedConditions.alertIsPresent());

//Type your message
alert.sendKeys("Selenium");

//Press the OK button
alert.accept();
```

2.3.3 - Working with cookies

A cookie is a small piece of data that is sent from a website and stored in your computer. Cookies are mostly used to recognise the user and load the stored information.

WebDriver API provides a way to interact with cookies with built-in methods:

Add Cookie

It is used to add a cookie to the current browsing context. Add Cookie only accepts a set of defined serializable JSON object. [Here](#) is the link to the list of accepted JSON key values

First of all, you need to be on the domain that the cookie will be valid for. If you are trying to preset cookies before you start interacting with a site and your homepage is large / takes a while to load an alternative is to find a smaller page on the site (typically the 404 page is small, e.g.

<http://example.com/some404page>)

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class addCookie {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("http://www.example.com");

            // Adds the cookie into current browser context
            driver.manage().addCookie(new Cookie("key", "value"));
        } finally {
            driver.quit();
        }
    }
}
```

Get Named Cookie

It returns the serialized cookie data matching with the cookie name among all associated cookies.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class getCookieNamed {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("http://www.example.com");
            driver.manage().addCookie(new Cookie("foo", "bar"));

            // Get cookie details with named cookie 'foo'
        }
    }
}
```

```
        Cookie cookie1 = driver.manage().getCookieNamed("foo");
        System.out.println(cookie1);
    } finally {
        driver.quit();
    }
}
```

Get All Cookies

It returns a ‘successful serialized cookie data’ for current browsing context. If browser is no longer available it returns error.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import java.util.Set;

public class getAllCookies {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("http://www.example.com");
            // Add few cookies
            driver.manage().addCookie(new Cookie("test1", "cookie1"));
            driver.manage().addCookie(new Cookie("test2", "cookie2"));

            // Get All available cookies
            Set<Cookie> cookies = driver.manage().getCookies();
            System.out.println(cookies);
        } finally {
            driver.quit();
        }
    }
}
```

Delete Cookie

It deletes the cookie data matching with the provided cookie name.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class deleteCookie {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
```

```

driver.get("http://www.example.com");
driver.manage().addCookie(new Cookie("test1", "cookie1"));
Cookie cookie1 = new Cookie("test2", "cookie2");
driver.manage().addCookie(cookie1);

// delete a cookie with name 'test1'
driver.manage().deleteCookieNamed("test1");

/*
Selenium Java bindings also provides a way to delete
cookie by passing cookie object of current browsing context
*/
driver.manage().deleteCookie(cookie1);
} finally {
    driver.quit();
}
}
}

```

Delete All Cookies

It deletes all the cookies of the current browsing context.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```

import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class deleteAllCookies {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("http://www.example.com");
            driver.manage().addCookie(new Cookie("test1", "cookie1"));
            driver.manage().addCookie(new Cookie("test2", "cookie2"));

            // deletes all cookies
            driver.manage().deleteAllCookies();
        } finally {
            driver.quit();
        }
    }
}

```

Same-Site Cookie Attribute

It allows a user to instruct browsers to control whether cookies are sent along with the request initiated by third party sites. It is introduced to prevent CSRF (Cross-Site Request Forgery) attacks.

Same-Site cookie attribute accepts two parameters as instructions

Strict:

When the sameSite attribute is set as **Strict**, the cookie will not be sent along with requests initiated by third party websites.

Lax:

When you set a cookie sameSite attribute to **Lax**, the cookie will be sent along with the GET request initiated by third party website.

Note: As of now this feature is landed in chrome(80+version), Firefox(79+version) and works with Selenium 4 and later versions.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class cookieTest {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("http://www.example.com");
            Cookie cookie = new Cookie.Builder("key", "value").sameSite("Strict").build();
            Cookie cookie1 = new Cookie.Builder("key", "value").sameSite("Lax").build();
            driver.manage().addCookie(cookie);
            driver.manage().addCookie(cookie1);
            System.out.println(cookie.getSameSite());
            System.out.println(cookie1.getSameSite());
        } finally {
            driver.quit();
        }
    }
}
```

2.3.4 - Working with iFrames and frames

Frames are a now deprecated means of building a site layout from multiple documents on the same domain. You are unlikely to work with them unless you are working with an pre HTML5 webapp. Iframes allow the insertion of a document from an entirely different domain, and are still commonly used.

If you need to work with frames or iframes, WebDriver allows you to work with them in the same way. Consider a button within an iframe. If we inspect the element using the browser development tools, we might see the following:

```
<div id="modal">
  <iframe id="buttonframe" name="myframe" src="https://seleniumhq.github.io">
    <button>Click here</button>
  </iframe>
</div>
```

If it was not for the iframe we would expect to click on the button using something like:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//This won't work
driver.findElement(By.tagName("button")).click();
```

However, if there are no buttons outside of the iframe, you might instead get a *no such element* error. This happens because Selenium is only aware of the elements in the top level document. To interact with the button, we will need to first switch to the frame, in a similar way to how we switch windows. WebDriver offers three ways of switching to a frame.

Using a WebElement

Switching using a WebElement is the most flexible option. You can find the frame using your preferred selector and switch to it.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Store the web element
WebElement iframe = driver.findElement(By.cssSelector("#modal>iframe"));

//Switch to the frame
driver.switchTo().frame(iframe);

//Now we can click the button
driver.findElement(By.tagName("button")).click();
```

Using a name or ID

If your frame or iframe has an id or name attribute, this can be used instead. If the name or ID is not unique on the page, then the first one found will be switched to.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Using the ID  
driver.switchTo().frame("buttonframe");  
  
//Or using the name instead  
driver.switchTo().frame("myframe");  
  
//Now we can click the button  
driver.findElement(By.tagName("button")).click();
```

Using an index

It is also possible to use the index of the frame, such as can be queried using `window.frames` in JavaScript.

[Java](#) [Ruby](#) [CSharp](#) [Python](#) [JavaScript](#) [Kotlin](#)

```
// Switches to the second frame  
driver.switchTo().frame(1);
```

Leaving a frame

To leave an iframe or frameset, switch back to the default content like so:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Return to the top level  
driver.switchTo().defaultContent();
```

2.3.5 - Working with windows and tabs

Windows and tabs

Get window handle

WebDriver does not make the distinction between windows and tabs. If your site opens a new tab or window, Selenium will let you work with it using a window handle. Each window has a unique identifier which remains persistent in a single session. You can get the window handle of the current window by using:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.getWindowHandle();
```

Switching windows or tabs

Clicking a link which opens in a [new window](#) will focus the new window or tab on screen, but WebDriver will not know which window the Operating System considers active. To work with the new window you will need to switch to it. If you have only two tabs or windows open, and you know which window you start with, by the process of elimination you can loop over both windows or tabs that WebDriver can see, and switch to the one which is not the original.

However, Selenium 4 provides a new api [NewWindow](#) which creates a new tab (or) new window and automatically switches to it.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Store the ID of the original window
String originalWindow = driver.getWindowHandle();

//Check we don't have other windows open already
assert driver.getWindowHandles().size() == 1;

//Click the link which opens in a new window
driver.findElement(By.linkText("new window")).click();

//Wait for the new window or tab
wait.until(numberOfWindowsToBe(2));

//Loop through until we find a new window handle
for (String windowHandle : driver.getWindowHandles()) {
    if(!originalWindow.contentEquals(windowHandle)) {
        driver.switchTo().window(windowHandle);
        break;
    }
}

//Wait for the new tab to finish loading content
wait.until(titleIs("Selenium documentation"));
```

Create new window (or) new tab and switch

Creates a new window (or) tab and will focus the new window or tab on screen. You don't need to switch to work with the new window (or) tab. If you have more than two windows (or) tabs opened other than the new window, you can loop over both windows or tabs that WebDriver can see, and switch to the one which is not the original.

Note: This feature works with Selenium 4 and later versions.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Opens a new tab and switches to new tab  
driver.switchTo().newWindow(WindowType.TAB);  
  
// Opens a new window and switches to new window  
driver.switchTo().newWindow(WindowType.WINDOW);
```

Closing a window or tab

When you are finished with a window or tab *and* it is not the last window or tab open in your browser, you should close it and switch back to the window you were using previously. Assuming you followed the code sample in the previous section you will have the previous window handle stored in a variable. Put this together and you will get:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Close the tab or window  
driver.close();  
  
//Switch back to the old tab or window  
driver.switchTo().window(originalWindow);
```

Forgetting to switch back to another window handle after closing a window will leave WebDriver executing on the now closed page, and will trigger a **No Such Window Exception**. You must switch back to a valid window handle in order to continue execution.

Quitting the browser at the end of a session

When you are finished with the browser session you should call quit, instead of close:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.quit();
```

- Quit will:
 - Close all the windows and tabs associated with that WebDriver session
 - Close the browser process
 - Close the background driver process
 - Notify Selenium Grid that the browser is no longer in use so it can be used by another session (if you are using Selenium Grid)

Failure to call quit will leave extra background processes and ports running on your machine which could cause you problems later.

Some test frameworks offer methods and annotations which you can hook into to tear down at the end of a test.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
/**  
 * Example using JUnit  
 * https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/AfterAll.html  
 */  
  
@AfterAll  
public static void tearDown() {  
    driver.quit();  
}
```

If not running WebDriver in a test context, you may consider using `try / finally` which is offered by most languages so that an exception will still clean up the WebDriver session.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
try {  
    //WebDriver code here...  
} finally {  
    driver.quit();  
}
```

Python's WebDriver now supports the python context manager, which when using the `with` keyword can automatically quit the driver at the end of execution.

```
with webdriver.Firefox() as driver:  
    # WebDriver code here...  
  
# WebDriver will automatically quit after indentation
```

Window management

Screen resolution can impact how your web application renders, so WebDriver provides mechanisms for moving and resizing the browser window.

Get window size

Fetches the size of the browser window in pixels.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//Access each dimension individually  
int width = driver.manage().window().getSize().getWidth();  
int height = driver.manage().window().getSize().getHeight();
```

```
//Or store the dimensions and query them later
Dimension size = driver.manage().window().getSize();
int width1 = size.getWidth();
int height1 = size.getHeight();
```

Set window size

Restores the window and sets the window size.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.manage().window().setSize(new Dimension(1024, 768));
```

Get window position

Fetches the coordinates of the top left coordinate of the browser window.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Access each dimension individually
int x = driver.manage().window().getPosition().getX();
int y = driver.manage().window().getPosition().getY();

// Or store the dimensions and query them later
Point position = driver.manage().window().getPosition();
int x1 = position.getX();
int y1 = position.getY();
```

Set window position

Moves the window to the chosen position.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Move the window to the top left of the primary monitor
driver.manage().window().setPosition(new Point(0, 0));
```

Maximize window

Enlarges the window. For most operating systems, the window will fill the screen, without blocking the operating system's own menus and toolbars.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.manage().window().maximize();
```

Minimize window

Minimizes the window of current browsing context. The exact behavior of this command is specific to individual window managers.

Minimize Window typically hides the window in the system tray.

Note: This feature works with Selenium 4 and later versions.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.manage().window().minimize();
```

Fullscreen window

Fills the entire screen, similar to pressing F11 in most browsers.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.manage().window().fullscreen();
```

TakeScreenshot

Used to capture screenshot for current browsing context. The WebDriver endpoint [Screenshot](#) returns screenshot which is encoded in Base64 format.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
from selenium import webdriver

driver = webdriver.Chrome()

driver.get("http://www.example.com")

# Returns and base64 encoded string into image
driver.save_screenshot('./image.png')

driver.quit()
```

TakeElementScreenshot

Used to capture screenshot of an element for current browsing context. The WebDriver endpoint [Screenshot](#) returns screenshot which is encoded in Base64 format.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```

import org.apache.commons.io.FileUtils;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import java.io.File;
import java.io.IOException;

public class SeleniumelementTakeScreenshot {
    public static void main(String args[]) throws IOException {
        WebDriver driver = new ChromeDriver();
        driver.get("https://www.example.com");
        WebElement element = driver.findElement(By.cssSelector("h1"));
        File scrFile = element.getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(scrFile, new File("./image.png"));
        driver.quit();
    }
}

```

Execute Script

Executes JavaScript code snippet in the current context of a selected frame or window.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```

//Creating the JavascriptExecutor interface object by Type casting
JavascriptExecutor js = (JavascriptExecutor)driver;
//Button Element
WebElement button =driver.findElement(By.name("btnLogin"));
//Executing JavaScript to click on element
js.executeScript("arguments[0].click();", button);
//Get return value from script
String text = (String) js.executeScript("return arguments[0].innerText", b
//Executing JavaScript directly
js.executeScript("console.log('hello world')");

```

Print Page

Prints the current page within the browser.

Note: This requires Chromium Browsers to be in headless mode

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```

import org.openqa.selenium.print.PrintOptions;

driver.get("https://www.selenium.dev");
printer = (PrintsPage) driver;

PrintOptions printOptions = new PrintOptions();
printOptions.setPageRanges("1-2");

Pdf pdf = printer.print(printOptions);

```

```
String content = pdf.getContent();
```

2.4 - Web elements

Identifying and working with element objects in the DOM.

The majority of most people's Selenium code involves working with web elements.

2.4.1 - Locator strategies

Ways to identify one or more specific elements in the DOM.

A locator is a way to identify elements on a page. It is the argument passed to the [Finding element](#) methods.

Check out our [encouraged test practices](#) for tips on [locators](#), including which to use when and why to declare locators separately from the finding methods.

Traditional Locators

Selenium provides support for these 8 traditional location strategies in WebDriver:

Locator	Description
class name	Locates elements whose class name contains the search value (compound class names are not permitted)
css selector	Locates elements matching a CSS selector
id	Locates elements whose ID attribute matches the search value
name	Locates elements whose NAME attribute matches the search value
link text	Locates anchor elements whose visible text matches the search value
partial link text	Locates anchor elements whose visible text contains the search value. If multiple elements are matching, only the first one will be selected.
tag name	Locates elements whose tag name matches the search value
xpath	Locates elements matching an XPath expression

Coding Help



Note: This section could use some updated code examples

of selecting elements using each locator strategy

Check our [contribution guidelines](#) and [code example formats](#) if you'd like to help.

Relative Locators

Selenium 4 introduces Relative Locators (previously called as *Friendly Locators*). These locators are helpful when it is not easy to construct a locator for the desired element, but easy to describe spatially where the element is in relation to an element that does have an easily constructed locator.

How it works

Selenium uses the JavaScript function [getBoundingClientRect\(\)](#) to determine the size and position of elements on the page, and can use this information to locate neighboring elements. find the relative elements.

Relative locator methods can take as the argument for the point of origin, either a previously located element reference, or another locator. In these examples we'll be using locators only, but you could swap the locator in the final method with an element object and it will work the same.

Let us consider the below example for understanding the relative locators.

The form consists of two input fields: one for 'Email Address' and one for 'Password'. Below the inputs are two buttons: 'Cancel' on the left and 'Submit' on the right. The entire form is enclosed in a light gray border.

Available relative locators

Above

If the email text field element is not easily identifiable for some reason, but the password text field element is, we can locate the text field element using the fact that it is an "input" element "above" the password element.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
By emailLocator = RelativeLocator.with(By.tagName("input")).above(By.id("password"))
```

Below

If the password text field element is not easily identifiable for some reason, but the email text field element is, we can locate the text field element using the fact that it is an "input" element "below" the email element.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
By passwordLocator = RelativeLocator.with(By.tagName("input")).below(By.id("email"))
```

Left of

If the cancel button is not easily identifiable for some reason, but the submit button element is, we can locate the cancel button element using the fact that it is a "button" element to the "left of" the submit element.

```
By cancelLocator = RelativeLocator.with(By.tagName("button")).toLeftOf(By.id("su
```

Right of

If the submit button is not easily identifiable for some reason, but the cancel button element is, we can locate the submit button element using the fact that it is a “button” element “to the right of” the cancel element.

```
By submitLocator = RelativeLocator.with(By.tagName("button")).toRightOf(By.id("c
```

Near

If the relative positioning is not obvious, or it varies based on window size, you can use the near method to identify an element that is at most 50px away from the provided locator. One great use case for this is to work with a form element that doesn’t have an easily constructed locator, but its associated input label element does.

```
By emailLocator = RelativeLocator.with(By.tagName("input")).near(By.id("lbl-email
```

Chaining relative locators

You can also chain locators if needed. Sometimes the element is most easily identified as being both above/below one element and right/left of another.

```
By submitLocator = RelativeLocator.with(By.tagName("button")).below(By.id("email
```

2.4.2 - Finding web elements

Locating the elements based on the provided locator values.

One of the most fundamental aspects of using Selenium is obtaining element references to work with. Selenium offers a number of built-in [locator strategies](#) to uniquely identify an element. There are many ways to use the locators in very advanced scenarios. For the purposes of this documentation, let's consider this HTML snippet:

```
<ol id="vegetables">
  <li class="potatoes">...
  <li class="onions">...
  <li class="tomatoes"><span>Tomato is a Vegetable</span>...
</ol>
<ul id="fruits">
  <li class="bananas">...
  <li class="apples">...
  <li class="tomatoes"><span>Tomato is a Fruit</span>...
</ul>
```

First matching element

Many locators will match multiple elements on the page. The singular find element method will return a reference to the first element found within a given context.

Evaluating entire DOM

When the find element method is called on the driver instance, it returns a reference to the first element in the DOM that matches with the provided locator. This value can be stored and used for future element actions. In our example HTML above, there are two elements that have a class name of "tomatoes" so this method will return the element in the "vegetables" list.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement vegetable = driver.findElement(By.className("tomatoes"));
```

Evaluating a subset of the DOM

Rather than finding a unique locator in the entire DOM, it is often useful to narrow the search to the scope of another located element. In the above example there are two elements with a class name of "tomatoes" and it is a little more challenging to get the reference for the second one.

One solution is to locate an element with a unique attribute that is an ancestor of the desired element and not an ancestor of the undesired element, then call find element on that object:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement fruits = driver.findElement(By.id("fruits"));
WebElement fruit = fruits.findElement(By.id("tomatoes"));
```

Java and C#

`WebDriver`, `WebElement` and `ShadowRoot` classes all implement a `SearchContext` interface, which is considered a *role-based interface*. Role-based interfaces allow you to determine whether a particular driver implementation supports a given feature. These interfaces are clearly defined and try to adhere to having only a single role of responsibility.

Optimized locator

A nested lookup might not be the most effective location strategy since it requires two separate commands to be issued to the browser.

To improve the performance slightly, we can use either CSS or XPath to find this element in a single command. See the [Locator strategy suggestions](#) in our [Encouraged test practices](#) section.

For this example, we'll use a CSS Selector:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement fruit = driver.findElement(By.cssSelector("#fruits .tomatoes"));
```

All matching elements

There are several use cases for needing to get references to all elements that match a locator, rather than just the first one. The plural `findElements` methods return a collection of element references. If there are no matches, an empty list is returned. In this case, references to all fruits and vegetable list items will be returned in a collection.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
List<WebElement> plants = driver.findElements(By.tagName("li"));
```

Get element

Often you get a collection of elements but want to work with a specific element, which means you need to iterate over the collection and identify the one you want.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
List<WebElement> elements = driver.findElements(By.tagName("li"));

for (WebElement element : elements) {
    System.out.println("Paragraph text:" + element.getText());
}
```

Find Elements From Element

It is used to find the list of matching child WebElements within the context of parent element. To achieve this, the parent WebElement is chained with 'findElements' to access child elements

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import java.util.List;

public class findElementsFromElement {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("https://example.com");

            // Get element with tag name 'div'
            WebElement element = driver.findElement(By.tagName("div"));

            // Get all the elements available with tag name 'p'
            List<WebElement> elements = element.findElements(By.tagName("p"));
            for (WebElement e : elements) {
                System.out.println(e.getText());
            }
        } finally {
            driver.quit();
        }
    }
}
```

Get Active Element

It is used to track (or) find DOM element which has the focus in the current browsing context.

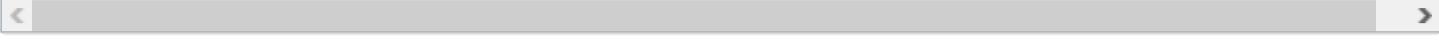
[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class activeElementTest {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            driver.get("http://www.google.com");
            driver.findElement(By.cssSelector("[name='q']")).sendKeys("webElement");

            // Get attribute of current active element
            String attr = driver.switchTo().activeElement().getAttribute("title");
        }
    }
}
```

```
    System.out.println(attr);
} finally {
    driver.quit();
}
}
```



2.4.3 - Interacting with web elements

A high-level instruction set for manipulating form controls.

There are only 5 basic commands that can be executed on an element:

- [click](#) (applies to any element)
- [send keys](#) (only applies to text fields and content editable elements)
- [clear](#) (only applies to text fields and content editable elements)
- submit (only applies to form elements)
- select (see [Select List Elements](#))

Additional validations

These methods are designed to closely emulate a user's experience, so, unlike the [Actions API](#), it attempts to perform two things before attempting the specified action.

1. If it determines the element is outside the viewport, it [scrolls the element into view](#), specifically it will align the bottom of the element with the bottom of the viewport.
2. It ensures the element is [interactable](#) before taking the action. This could mean that the scrolling was unsuccessful, or that the element is not otherwise displayed. Determining if an element is displayed on a page was too difficult to [define directly in the webdriver specification](#), so Selenium sends an execute command with a JavaScript atom that checks for things that would keep the element from being displayed. If it determines an element is not in the viewport, not displayed, not [keyboard-interactable](#), or not [pointer-interactable](#), it returns an [element not interactable](#) error.

Click

The [element click command](#) is executed on the [center of the element](#). If the center of the element is [obscured](#) for some reason, Selenium will return an [element click intercepted](#) error.

Send keys

The [element send keys command](#) types the provided keys into an [editable](#) element. Typically, this means an element is an input element of a form with a `text` type or an element with a `content-editable` attribute. If it is not editable, [an invalid element state](#) error is returned.

[Here](#) is the list of possible keystrokes that WebDriver Supports.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.By;
import org.openqa.selenium.Keys;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class HelloSelenium {
    public static void main(String[] args) {
        WebDriver driver = new FirefoxDriver();
        try {
            // Navigate to Url
            driver.get("https://google.com");

            // Enter text "q" and perform keyboard action "Enter"
            driver.findElement(By.name("q")).sendKeys("q" + Keys.ENTER);
        } finally {
            driver.quit();
        }
    }
}
```

Clear

The [element clear command](#) resets the content of an element. This requires an element to be [editable](#), and [resettable](#). Typically, this means an element is an input element of a form with a `text` type or an element with a `content-editable` attribute. If these conditions are not met, [an invalid element state](#) error is returned.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class clear {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        try {
            // Navigate to Url
            driver.get("https://www.google.com");
            // Store 'SearchInput' element
            WebElement searchInput = driver.findElement(By.name("q"));
            searchInput.sendKeys("selenium");
            // Clears the entered text
            searchInput.clear();
        } finally {
            driver.quit();
        }
    }
}
```

Submit

In Selenium 4 this is no longer implemented with a separate endpoint and functions by executing a script. As such, it is recommended not to use this method and to click the applicable form submission button instead.

2.4.4 - Information about web elements

What you can learn about an element.

There are a number of details you can query about a specific element.

Is Displayed

This method is used to check if the connected Element is displayed on a webpage. Returns a `Boolean` value, **True** if the connected element is displayed in the current browsing context else returns false.

This functionality is [mentioned in](#), but not defined by the w3c specification due to the [impossibility of covering all potential conditions](#). As such, Selenium cannot expect drivers to implement this functionality directly, and now relies on executing a large JavaScript function directly. This function makes many approximations about an element's nature and relationship in the tree to return a value.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Navigate to the url  
driver.get('https://www.google.com');  
  
// Get boolean value for is element display  
boolean isButtonVisible = driver.findElement(By.css("[name='login']")).isDisplay
```

Coding Help



Note: This section could use some updated code examples

for element displayedness

Check our [contribution guidelines](#) and [code example formats](#) if you'd like to help.

Is Enabled

This method is used to check if the connected Element is enabled or disabled on a webpage. Returns a boolean value, **True** if the connected element is **enabled** in the current browsing context else returns **false**.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//navigates to url  
driver.get("https://www.google.com/");  
  
//returns true if element is enabled else returns false  
boolean value = driver.findElement(By.name("btnK")).isEnabled();
```

Is Selected

This method determines if the referenced Element is *Selected* or not. This method is widely used on Check boxes, radio buttons, input elements, and option elements.

Returns a boolean value, **True** if referenced element is **selected** in the current browsing context else returns **false**.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//navigates to url
driver.get("https://the-internet.herokuapp.com/checkboxes");

//returns true if element is checked else returns false
boolean value = driver.findElement(By.cssSelector("input[type='checkbox']:first
```

Tag Name

It is used to fetch the TagName of the referenced Element which has the focus in the current browsing context.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
//navigates to url
driver.get("https://www.example.com");

//returns TagName of the element
String value = driver.findElement(By.cssSelector("h1")).getTagName();
```

Size and Position

It is used to fetch the dimensions and coordinates of the referenced element.

The fetched data body contain the following details:

- X-axis position from the top-left corner of the element
- y-axis position from the top-left corner of the element
- Height of the element
- Width of the element

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Navigate to url
driver.get("https://www.example.com");

// Returns height, width, x and y coordinates referenced element
Rectangle res = driver.findElement(By.cssSelector("h1")).getRect();

// Rectangle class provides getX, getY, getWidth, getHeight methods
System.out.println(res.getX());
```

Get CSS Value

Retrieves the value of specified computed style property of an element in the current browsing context.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Navigate to Url  
driver.get("https://www.example.com");  
  
// Retrieves the computed style property 'color' of linktext  
String cssValue = driver.findElement(By.linkText("More information...")).getCssV
```

Text Content

Retrieves the rendered text of the specified element.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Navigate to url  
driver.get("https://example.com");  
  
// Retrieves the text of the element  
String text = driver.findElement(By.cssSelector("h1")).getText();
```

Attributes and Properties

Attribute

DOM Attribute

DOM Property

2.4.5 - Working with select list elements

Select lists have special behaviors compared to other elements.

Select elements can require quite a bit of boilerplate code to automate. To reduce this, and make your tests cleaner, there is a `Select` class in the Selenium support package. To use it, you will need the following import statement:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.support.ui.Select;
```

You are then able to create a Select object using a WebElement that references a `<select>` element.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement selectElement = driver.findElement(By.id("selectElementID"));
Select selectObject = new Select(selectElement);
```

The Select object will now give you a series of commands that allow you to interact with a `<select>` element. First of all, there are different ways of selecting an option from the `<select>` element.

```
<select>
<option value=value1>Bread</option>
<option value=value2 selected>Milk</option>
<option value=value3>Cheese</option>
</select>
```

There are three ways to select the first option from the above element:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Select an <option> based upon the <select> element's internal index
selectObject.selectByIndex(1);

// Select an <option> based upon its value attribute
selectObject.selectByValue("value1");

// Select an <option> based upon its text
selectObject.selectByVisibleText("Bread");
```

You can then check which options are selected by using:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Return a List<WebElement> of options that have been selected
List<WebElement> allSelectedOptions = selectObject.getAllSelectedOptions();

// Return a WebElement referencing the first selection option found by walking down the tree
WebElement firstSelectedOption = selectObject.getFirstSelectedOption();
```

Or you may just be interested in what `<option>` elements the `<select>` element contains:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Return a List<WebElement> of options that the <select> element contains
List<WebElement> allAvailableOptions = selectObject.getOptions();
```

If you want to deselect any elements, you now have four options:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Deselect an <option> based upon the <select> element's internal index
selectObject.deselectByIndex(1);

// Deselect an <option> based upon its value attribute
selectObject.deselectByValue("value1");

// Deselect an <option> based upon its text
selectObject.deselectByVisibleText("Bread");

// Deselect all selected <option> elements
selectObject.deselectAll();
```

Finally, some `<select>` elements allow you to select more than one option. You can find out if your `<select>` element is one of these by using:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
Boolean doesThisAllowMultipleSelections = selectObject.isMultiple();
```

2.5 - Remote WebDriver

You can use WebDriver remotely the same way you would use it locally. The primary difference is that a remote WebDriver needs to be configured so that it can run your tests on a separate machine.

A remote WebDriver is composed of two pieces: a client and a server. The client is your WebDriver test and the server is simply a Java servlet, which can be hosted in any modern JEE app server.

To run a remote WebDriver client, we first need to connect to the RemoteWebDriver. We do this by pointing the URL to the address of the server running our tests. In order to customize our configuration, we set desired capabilities. Below is an example of instantiating a remote WebDriver object pointing to our remote web server, www.example.com, running our tests on Firefox.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
FirefoxOptions firefoxOptions = new FirefoxOptions();
WebDriver driver = new RemoteWebDriver(new URL("http://www.example.com"), firefoxOptions);
driver.get("http://www.google.com");
driver.quit();
```

To further customize our test configuration, we can add other desired capabilities.

Browser options

For example, suppose you wanted to run Chrome on Windows XP, using Chrome version 67:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setCapability("browserVersion", "67");
chromeOptions.setCapability("platformName", "Windows XP");
WebDriver driver = new RemoteWebDriver(new URL("http://www.example.com"), chromeOptions);
driver.get("http://www.google.com");
driver.quit();
```

Local file detector

The Local File Detector allows the transfer of files from the client machine to the remote server. For example, if a test needs to upload a file to a web application, a remote WebDriver can automatically transfer the file from the local machine to the remote web server during runtime. This allows the file to be uploaded from the remote machine running the test. It is not enabled by default and can be enabled in the following way:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.setFileDetector(new LocalFileDetector());
```

Once the above code is defined, you can upload a file in your test in the following way:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.get("http://sso.dev.saucelabs.com/test/guinea-file-upload");
WebElement upload = driver.findElement(By.id("myfile"));
upload.sendKeys("/Users/sso/the/local/path/to/darkbulb.jpg");
```

Tracing client requests

This feature is only available for Java client binding (Beta onwards). The Remote WebDriver client sends requests to the Selenium Grid server, which passes them to the WebDriver. Tracing should be enabled at the server and client-side to trace the HTTP requests end-to-end. Both ends should have a trace exporter setup pointing to the visualization framework. By default, tracing is enabled for both client and server. To set up the visualization framework Jaeger UI and Selenium Grid 4, please refer to [Tracing Setup](#) for the desired version.

For client-side setup, follow the steps below.

Add the required dependencies

Installation of external libraries for tracing exporter can be done using Maven. Add the *opentelemetry-exporter-jaeger* and *grpc-netty* dependency in your project pom.xml:

```
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-jaeger</artifactId>
    <version>1.0.0</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty</artifactId>
    <version>1.35.0</version>
</dependency>
```

Add/pass the required system properties while running the client

[Java](#)

```
System.setProperty("otel.traces.exporter", "jaeger");
System.setProperty("otel.exporter.jaeger.endpoint", "http://localhost:14250");
System.setProperty("otel.resource.attributes", "service.name=selenium-java-clien

ImmutableCapabilities capabilities = new ImmutableCapabilities("browserName", "c

WebDriver driver = new RemoteWebDriver(new URL("http://www.example.com"), capabi

driver.get("http://www.google.com");

driver.quit();
```

Please refer to [Tracing Setup](#) for more information on external dependencies versions required for the desired Selenium version.

More information can be found at:

- OpenTelemetry: <https://opentelemetry.io>
- Configuring OpenTelemetry: <https://github.com/open-telemetry/opentelemetry-java/tree/main/sdk-extensions/autoconfigure>
- Jaeger: <https://www.jaegertracing.io>
- [Selenium Grid Observability](#)

2.6 - Configuring driver parameters

We learned how to [install drivers](#) in the Getting Started section.

Selenium provides access to Service classes which are used to determine how the server is started

2.7 - Waits

WebDriver can generally be said to have a blocking API. Because it is an out-of-process library that *instructs* the browser what to do, and because the web platform has an intrinsically asynchronous nature, WebDriver does not track the active, real-time state of the DOM. This comes with some challenges that we will discuss here.

From experience, most intermittent issues that arise from use of Selenium and WebDriver are connected to *race conditions* that occur between the browser and the user's instructions. An example could be that the user instructs the browser to navigate to a page, then gets a **no such element** error when trying to find an element.

Consider the following document:

```
<!doctype html>
<meta charset=utf-8>
<title>Race Condition Example</title>

<script>
  var initialised = false;
  window.addEventListener("load", function() {
    var newElement = document.createElement("p");
    newElement.textContent = "Hello from JavaScript!";
    document.body.appendChild(newElement);
    initialised = true;
  });
</script>
```

The WebDriver instructions might look innocent enough:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
driver.get("file:///race_condition.html");
WebElement element = driver.findElement(By.tagName("p"));
assertEquals(element.getText(), "Hello from JavaScript!");
```

The issue here is that the default [page load strategy](#) used in WebDriver listens for the `document.readyState` to change to "complete" before returning from the call to `navigate`. Because the `p` element is added *after* the document has completed loading, this WebDriver script *might* be intermittent. It "might" be intermittent because no guarantees can be made about elements or events that trigger asynchronously without explicitly waiting—or blocking—on those events.

Fortunately, the normal instruction set available on the [WebElement](#) interface—such as `WebElement.click` and `WebElement.sendKeys`—are guaranteed to be synchronous, in that the function calls will not return (or the callback will not trigger in callback-style languages) until the command has been completed in the browser. The advanced user interaction APIs, [Keyboard](#) and [Mouse](#), are exceptions as they are explicitly intended as “do what I say” asynchronous commands.

Waiting is having the automated task execution elapse a certain amount of time before continuing with the next step.

To overcome the problem of race conditions between the browser and your WebDriver script, most Selenium clients ship with a `wait` package. When employing a wait, you are using what is commonly referred to as an [explicit wait](#).

Explicit wait

Explicit waits are available to Selenium clients for imperative, procedural languages. They allow your code to halt program execution, or freeze the thread, until the *condition* you pass it resolves. The condition is called with a certain frequency until the timeout of the wait is elapsed. This means that for as long as the condition returns a falsy value, it will keep trying and waiting.

Since explicit waits allow you to wait for a condition to occur, they make a good fit for synchronising the state between the browser and its DOM, and your WebDriver script.

To remedy our buggy instruction set from earlier, we could employ a wait to have the *findElement* call wait until the dynamically added element from the script has been added to the DOM:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebDriver driver = new ChromeDriver();
driver.get("https://google.com/ncr");
driver.findElement(By.name("q")).sendKeys("cheese" + Keys.ENTER);
// Initialize and wait till element(link) became clickable - timeout in 10 seconds
WebElement firstResult = new WebDriverWait(driver, Duration.ofSeconds(10))
    .until(ExpectedConditions.elementToBeClickable(By.xpath("//a/h3")));
// Print the first result
System.out.println(firstResult.getText());
```

We pass in the *condition* as a function reference that the *wait* will run repeatedly until its return value is truthy. A “truthful” return value is anything that evaluates to boolean true in the language at hand, such as a string, number, a boolean, an object (including a *WebElement*), or a populated (non-empty) sequence or list. That means an *empty list* evaluates to false. When the condition is truthful and the blocking wait is aborted, the return value from the condition becomes the return value of the wait.

With this knowledge, and because the wait utility ignores *no such element* errors by default, we can refactor our instructions to be more concise:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement foo = new WebDriverWait(driver, Duration.ofSeconds(3))
    .until(driver -> driver.findElement(By.name("q")));
assertEquals(foo.getText(), "Hello from JavaScript!");
```

In that example, we pass in an anonymous function (but we could also define it explicitly as we did earlier so it may be reused). The first and only argument that is passed to our condition is always a reference to our driver object, *WebDriver*. In a multi-threaded environment, you should be careful to operate on the driver reference passed in to the condition rather than the reference to the driver in the outer scope.

Because the wait will swallow *no such element* errors that are raised when the element is not found, the condition will retry until the element is found. Then it will take the return value, a *WebElement*, and pass it back through to our script.

If the condition fails, e.g. a truthful return value from the condition is never reached, the wait will throw/raise an error/exception called a *timeout error*.

Options

The wait condition can be customised to match your needs. Sometimes it is unnecessary to wait the full extent of the default timeout, as the penalty for not hitting a successful condition can be expensive.

The wait lets you pass in an argument to override the timeout:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
new WebDriverWait(driver, Duration.ofSeconds(3)).until(ExpectedConditions.elemen
```

Expected conditions

Because it is quite a common occurrence to have to synchronise the DOM and your instructions, most clients also come with a set of predefined *expected conditions*. As might be obvious by the name, they are conditions that are predefined for frequent wait operations.

The conditions available in the different language bindings vary, but this is a non-exhaustive list of a few:

- alert is present
- element exists
- element is visible
- title contains
- title is
- element staleness
- visible text

You can refer to the API documentation for each client binding to find an exhaustive list of expected conditions:

- Java's [org.openqa.selenium.support.ui.ExpectedConditions](#) class
- Python's [selenium.webdriver.support.expected_conditions](#) class
- .NET's [OpenQA.Selenium.Support.UI.ExpectedConditions](#) type
- JavaScript's [selenium-webdriver/lib/until](#) module

Implicit wait

There is a second type of wait that is distinct from [explicit wait](#) called *implicit wait*. By implicitly waiting, WebDriver polls the DOM for a certain duration when trying to find *any* element. This can be useful when certain elements on the webpage are not available immediately and need some time to load.

Implicit waiting for elements to appear is disabled by default and will need to be manually enabled on a per-session basis. Mixing [explicit waits](#) and implicit waits will cause unintended consequences, namely waits sleeping for the maximum time even if the element is available or condition is true.

Warning: Do not mix implicit and explicit waits. Doing so can cause unpredictable wait times. For example, setting an implicit wait of 10 seconds and an explicit wait of 15 seconds could cause a timeout to occur after 20 seconds.

An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0, meaning disabled. Once set, the implicit wait is set for the life of the session.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
driver.get("http://somedomain/url_that_delays_loading");
```

```
WebElement myDynamicElement = driver.findElement(By.id("myDynamicElement"));
```

FluentWait

FluentWait instance defines the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition.

Users may configure the wait to ignore specific types of exceptions whilst waiting, such as `NoSuchElementException` when searching for an element on the page.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Waiting 30 seconds for an element to be present on the page, checking
// for its presence once every 5 seconds.
Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
    .withTimeout(Duration.ofSeconds(30))
    .pollingEvery(Duration.ofSeconds(5))
    .ignoring(NoSuchElementException.class);

WebElement foo = wait.until(new Function<WebDriver, WebElement>() {
    public WebElement apply(WebDriver driver) {
        return driver.findElement(By.id("foo"));
    }
});
```

2.8 - Actions API

A low-level interface for providing virtualized device input actions to the web browser.

In addition to the high-level [element interactions](#), the [Actions API](#) provides granular control over exactly what designated input devices can do. Selenium provides an interface for 3 kinds of input sources: a key input for keyboard devices, a pointer input for a mouse, pen or touch devices, and wheel inputs for scroll wheel devices (introduced in Selenium 4.2). Selenium allows you to construct individual action commands assigned to specific inputs and chain them together and call the associated perform method to execute them all at once.

Action Builder

In the move from the legacy JSON Wire Protocol to the new W3C WebDriver Protocol, the low level building blocks of actions became especially detailed. It is extremely powerful, but each input device has a number of ways to use it and if you need to manage more than one device, you are responsible for ensuring proper synchronization between them.

Thankfully, you likely do not need to learn how to use the low level commands directly, since almost everything you might want to do has been given a convenience method that combines the lower level commands for you. These are all documented in [keyboard](#), [mouse](#), [pen](#), and [wheel](#) pages.

Pause

Pointer movements and Wheel scrolling allow the user to set a duration for the action, but sometimes you just need to wait a beat between actions for things to work correctly.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement clickable = driver.findElement(By.id("clickable"));
new Actions(driver)
    .moveToElement(clickable)
    .pause(Duration.ofSeconds(1))
    .clickAndHold()
    .pause(Duration.ofSeconds(1))
    .sendKeys("abc")
    .perform();
```

 [Check code on GitHub](#)

Release All Actions

An important thing to note is that the driver remembers the state of all the input items throughout a session. Even if you create a new instance of an actions class, the depressed keys and the location of the pointer will be in whatever state a previously performed action left them.

There is a special method to release all currently depressed keys and pointer buttons. This method is implemented differently in each of the languages because it does not get executed with the perform method.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
((RemoteWebDriver) driver).resetInputState();
```

 [Check code on GitHub](#)

2.8.1 - Keyboard actions

A representation of any key input device for interacting with a web page.

There are only 2 actions that can be accomplished with a keyboard: pressing down on a key, and releasing a pressed key. In addition to supporting ASCII characters, each keyboard key has a representation that can be pressed or released in designated sequences.

Keys

In addition to the keys represented by regular unicode, unicode values have been assigned to other keyboard keys for use with Selenium. Each language has its own way to reference these keys; the full list can be found [here](#).

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

Use the [Java Keys enum](#)

Key down

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
new Actions(driver)
    .keyDown(Keys.SHIFT)
    .sendKeys("a")
    .perform();
```

 [Check code on GitHub](#)

Key up

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
new Actions(driver)
    .keyDown(Keys.SHIFT)
    .sendKeys("a")
    .keyUp(Keys.SHIFT)
    .sendKeys("b")
    .perform();
```

 [Check code on GitHub](#)

Send keys

This is a convenience method in the Actions API that combines keyDown and keyUp commands in one action. Executing this command differs slightly from using the element method, but primarily this gets used when needing to type multiple characters in the middle of other actions.

Active Element

Java Python CSharp Ruby JavaScript Kotlin

```
new Actions(driver)
    .sendKeys("abc")
    .perform();
```

 Check code on GitHub

Designated Element

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement textField = driver.findElement(By.id("textInput"));
new Actions(driver)
    .sendKeys(textField, "Selenium!")
    .perform();
```

 [Check code on GitHub](#)

Copy and Paste

Here's an example of using all of the above methods to conduct a copy / paste action. Note that the key to use for this operation will be different depending on if it is a Mac OS or not. This code will end up with the text: SeleniumSelenium!

Java Python CSharp Ruby JavaScript Kotlin

```
Keys cmdCtrl = platformName.is(Platform.MAC) ? Keys.COMMAND : Keys.CONTR

WebElement textField = driver.findElement(By.id("textInput"));
new Actions(driver)
    .sendKeys(textField, "Selenium!")
    .sendKeys(Keys.ARROW_LEFT)
    .keyDown(Keys.SHIFT)
    .sendKeys(Keys.ARROW_UP)
    .keyUp(Keys.SHIFT)
    .keyDown(cmdCtrl)
    .sendKeys("xvv")
    .keyUp(cmdCtrl)
    .perform();
```

 Check code on GitHub

2.8.2 - Mouse actions

A representation of any pointer device for interacting with a web page.

There are only 3 actions that can be accomplished with a mouse: pressing down on a button, releasing a pressed button, and moving the mouse. Selenium provides convenience methods that combine these actions in the most common ways.

Click and hold

This method combines moving the mouse to the center of an element with pressing the left mouse button. This is useful for focusing a specific element:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement clickable = driver.findElement(By.id("clickable"));
new Actions(driver)
    .clickAndHold(clickable)
    .perform();
```

 [Check code on GitHub](#)

Click and release

This method combines moving to the center of an element with pressing and releasing the left mouse button. This is otherwise known as “clicking”:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement clickable = driver.findElement(By.id("click"));
new Actions(driver)
    .click(clickable)
    .perform();
```

 [Check code on GitHub](#)

Alternate Button Clicks

There are a total of 5 defined buttons for a Mouse:

- 0 — Left Button (the default)
- 1 — Middle Button (currently unsupported)
- 2 — Right Button
- 3 — X1 (Back) Button
- 4 — X2 (Forward) Button

Context Click

This method combines moving to the center of an element with pressing and releasing the right mouse button (button 2). This is otherwise known as “right-clicking”:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement clickable = driver.findElement(By.id("clickable"));
new Actions(driver)
    .contextClick(clickable)
    .perform();
```

 [Check code on GitHub](#)

Back Click

There is no convenience method for this, it is just pressing and releasing mouse button 3

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
PointerInput mouse = new PointerInput(PointerInput.Kind.MOUSE, "default"

Sequence actions = new Sequence(mouse, 0)
    .addAction(mouse.createPointerDown(PointerInput.MouseButton.BACK))
    .addAction(mouse.createPointerUp(PointerInput.MouseButton.BACK))

((RemoteWebDriver) driver).perform(Collections.singletonList(actions));
```

 [Check code on GitHub](#)

Forward Click

There is no convenience method for this, it is just pressing and releasing mouse button 4

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
PointerInput mouse = new PointerInput(PointerInput.Kind.MOUSE, "default"

Sequence actions = new Sequence(mouse, 0)
    .addAction(mouse.createPointerDown(PointerInput.MouseButton.FORWARD))
    .addAction(mouse.createPointerUp(PointerInput.MouseButton.FORWARD))

((RemoteWebDriver) driver).perform(Collections.singletonList(actions));
```

 [Check code on GitHub](#)

Double click

This method combines moving to the center of an element with pressing and releasing the left mouse button twice.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement clickable = driver.findElement(By.id("clickable"));
new Actions(driver)
    .doubleClick(clickable)
    .perform();
```

 [Check code on GitHub](#)

Move to element

This method moves the mouse to the in-view center point of the element. This is otherwise known as “hovering.” Note that the element must be in the viewport or else the command will error.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement hoverable = driver.findElement(By.id("hover"));
new Actions(driver)
    .moveToElement(hoverable)
    .perform();
```

 [Check code on GitHub](#)

Move by offset

These methods first move the mouse to the designated origin and then by the number of pixels in the provided offset. Note that the position of the mouse must be in the viewport or else the command will error.

Offset from Element (Top Left Origin)

This method moves the mouse to the in-view center point of the element then attempts to move to the upper left corner of the element and then moves by the provided offset.

This will be removed as an option in Selenium 4.3, and only an offset from center of the element will be supported. As of Selenium 4.2, this is the default behavior for Ruby, .NET and Python in order to be backwards compatible with previous versions of Selenium. This approach does not work correctly when the element is not entirely inside the viewport.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

Not Implemented in Selenium 4

Offset from Element (Center Origin)

This method moves to the in-view center point of the element, then moves the mouse by the provided offset

This is the default behavior in Java as of Selenium 4.0, and will be the default for the remaining languages as of Selenium 4.3.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement tracker = driver.findElement(By.id("mouse-tracker"));
new Actions(driver)
    .moveToElement(tracker, 8, 0)
```

 [Check code on GitHub](#)

Offset from Viewport

This method moves the mouse from the upper left corner of the current viewport by the provided offset.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
PointerInput mouse = new PointerInput(PointerInput.Kind.MOUSE, "default"

Sequence actions = new Sequence(mouse, 0)
    .addAction(mouse.createPointerMove(Duration.ZERO, PointerInput.O

((RemoteWebDriver) driver).perform(Collections.singletonList(actions));
```

 [Check code on GitHub](#)

Offset from Current Pointer Location

This method moves the mouse from its current position by the offset provided by the user. If the mouse has not previously been moved, the position will be in the upper left corner of the viewport. Note that the pointer position does not change when the page is scrolled.

Note that the first argument X specifies to move right when positive, while the second argument Y specifies to move down when positive. So `moveByOffset(30, -10)` moves right 30 and up 10 from the current mouse position.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
new Actions(driver)
    .moveByOffset(13, 15)
    .perform();
```

 [Check code on GitHub](#)

Drag and Drop on Element

This method firstly performs a click-and-hold on the source element, moves to the location of the target element and then releases the mouse.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement draggable = driver.findElement(By.id("draggable"));
WebElement droppable = driver.findElement(By.id("droppable"));
new Actions(driver)
    .dragAndDrop(draggable, droppable)
    .perform();
```

 [Check code on GitHub](#)

Drag and Drop by Offset

This method firstly performs a click-and-hold on the source element, moves to the given offset and then releases the mouse.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement draggable = driver.findElement(By.id("draggable"));
Rectangle start = draggable.getRect();
Rectangle finish = driver.findElement(By.id("droppable")).getRect();
new Actions(driver)
    .dragAndDropBy(draggable, finish.getX() - start.getX(), finish.g
    .perform();
```

 [Check code on GitHub](#)

2.8.3 - Pen actions

A representation of a pen stylus kind of pointer input for interacting with a web page.

Chromium Only

A Pen is a type of pointer input that has most of the same behavior as a mouse, but can also have event properties unique to a stylus. Additionally, while a mouse has 5 buttons, a pen has 3 equivalent button states:

- 0 — Touch Contact (the default; equivalent to a left click)
- 2 — Barrel Button (equivalent to a right click)
- 5 — Eraser Button (currently unsupported by drivers)

Using a Pen

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

Selenium v4.2

```
WebElement pointerArea = driver.findElement(By.id("pointerArea"));
new Actions(driver)
    .setActivePointer(PointerInput.Kind.PEN, "default pen")
    .moveToElement(pointerArea)
    .clickAndHold()
    .moveByOffset(2, 2)
    .release()
    .perform();
```

 [Check code on GitHub](#)

Adding Pointer Event Attributes

Selenium v4.2

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement pointerArea = driver.findElement(By.id("pointerArea"));
PointerInput pen = new PointerInput(PointerInput.Kind.PEN, "default pen")
PointerInput.PointerEventProperties eventProperties = PointerInput.event
    .setTiltX(-72)
    .setTiltY(9)
    .setTwist(86);
PointerInput-Origin origin = PointerInput-Origin.fromElement(pointerArea)

Sequence actionListPen = new Sequence(pen, 0)
    .addAction(pen.createPointerMove(Duration.ZERO, origin, 0, 0))
    .addAction(pen.createPointerDown(0))
    .addAction(pen.createPointerMove(Duration.ZERO, origin, 2, 2, ev))
    .addAction(pen.createPointerUp(0));

((RemoteWebDriver) driver).perform(Collections.singletonList(actionListP
```

 [Check code on GitHub](#)

2.8.4 - Scroll wheel actions

A representation of a scroll wheel input device for interacting with a web page.

[Selenium v4.2](#)

[Chromium Only](#)

There are 5 scenarios for scrolling on a page.

Scroll to element

This is the most common scenario. Unlike traditional click and send keys methods, the actions class does not automatically scroll the target element into view, so this method will need to be used if elements are not already inside the viewport.

This method takes a web element as the sole argument.

Regardless of whether the element is above or below the current viewscreen, the viewport will be scrolled so the bottom of the element is at the bottom of the screen.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement iframe = driver.findElement(By.tagName("iframe"));
new Actions(driver)
    .scrollToElement(iframe)
    .perform();
```

 [Check code on GitHub](#)

Scroll by given amount

This is the second most common scenario for scrolling. Pass in an delta x and a delta y value for how much to scroll in the right and down directions. Negative values represent left and up, respectively.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
WebElement footer = driver.findElement(By.tagName("footer"));
int deltaY = footer.getRect().y;
new Actions(driver)
    .scrollByAmount(0, deltaY)
    .perform();
```

 [Check code on GitHub](#)

Scroll from an element by a given amount

This scenario is effectively a combination of the above two methods.

To execute this use the “Scroll From” method, which takes 3 arguments. The first represents the origination point, which we designate as the element, and the second two are the delta x and delta y values.

If the element is out of the viewport, it will be scrolled to the bottom of the screen, then the page will be scrolled by the provided delta x and delta y values.

```
WebElement iframe = driver.findElement(By.tagName("iframe"));
WheelInput.ScrollOrigin scrollOrigin = WheelInput.ScrollOrigin.fromElement(iframe);
new Actions(driver)
    .scrollFromOrigin(scrollOrigin, 0, 200)
    .perform();
```

[Check code on GitHub](#)

Scroll from an element with an offset

This scenario is used when you need to scroll only a portion of the screen, and it is outside the viewport. Or is inside the viewport and the portion of the screen that must be scrolled is a known offset away from a specific element.

This uses the “Scroll From” method again, and in addition to specifying the element, an offset is specified to indicate the origin point of the scroll. The offset is calculated from the center of the provided element.

If the element is out of the viewport, it first will be scrolled to the bottom of the screen, then the origin of the scroll will be determined by adding the offset to the coordinates of the center of the element, and finally the page will be scrolled by the provided delta x and delta y values.

Note that if the offset from the center of the element falls outside of the viewport, it will result in an exception.

```
WebElement footer = driver.findElement(By.tagName("footer"));
WheelInput.ScrollOrigin scrollOrigin = WheelInput.ScrollOrigin.fromElement(footer);
new Actions(driver)
    .scrollFromOrigin(scrollOrigin, 0, 200)
    .perform();
```

[Check code on GitHub](#)

Scroll from a offset of origin (element) by given amount

The final scenario is used when you need to scroll only a portion of the screen, and it is already inside the viewport.

This uses the “Scroll From” method again, but the viewport is designated instead of an element. An offset is specified from the upper left corner of the current viewport. After the origin point is determined, the page will be scrolled by the provided delta x and delta y values.

Note that if the offset from the upper left corner of the viewport falls outside of the screen, it will result in an exception.

[Java](#)[Python](#)[CSharp](#)[Ruby](#)[JavaScript](#)[Kotlin](#)

```
WheelInput.ScrollOrigin scroll0rigin = WheelInput.ScrollOrigin.fromViewp  
new Actions(driver)  
    .scrollFromOrigin(scroll0rigin, 0, 200)  
    .perform();
```

 [Check code on GitHub](#)

2.9 - BiDirectional functionality

Selenium is working with browser vendors to create the [WebDriver BiDirectional Protocol](#) as a means to provide a stable, cross-browser API that uses the bidirectional functionality useful for both browser automation generally and testing specifically. Before now, users seeking this functionality have had to rely on with all of its frustrations and limitations.

The traditional WebDriver model of strict request/response commands will be supplemented with the ability to stream events from the user agent to the controlling software via WebSockets, better matching the evented nature of the browser DOM.

As it is not a good idea to tie your tests to a specific version of any browser, the Selenium project recommends using WebDriver BiDi wherever possible.

However, until the specification is complete there are many useful things that CDP (Chrome DevTools Protocol) offers. To help keep your tests independent and portable, Selenium offers some useful helper classes as well. At the moment, they use the CDP, but soon it could be done using WebDriver BiDi.

2.9.1 - BiDirectional API

The following list of APIs will be growing as the Selenium project works through supporting real world use cases. If there is additional functionality you'd like to see, please raise a [feature request](#).

Register Basic Auth

Some applications make use of browser authentication to secure pages. With Selenium, you can automate the input of basic auth credentials whenever they arise.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
Predicate<URI> uriPredicate = uri -> uri.getHost().contains("your-domain.com");

((HasAuthentication) driver).register(uriPredicate, UsernameAndPassword.of("admin",
driver.get("https://your-domain.com/login"));
```

Mutation Observation

Mutation Observation is the ability to capture events via WebDriver BiDi when there are DOM mutations on a specific element in the DOM.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
ChromeDriver driver = new ChromeDriver();

AtomicReference<DomMutationEvent> seen = new AtomicReference<>();
CountDownLatch latch = new CountDownLatch(1);
((HasLogEvents) driver).onLogEvent(domMutation -> {
    seen.set(domMutation);
    latch.countDown();
});

driver.get("https://www.google.com");
WebElement span = driver.findElement(By.cssSelector("span"));

((JavascriptExecutor) driver).executeScript("arguments[0].setAttribute('cheese',",
    assertThat(latch.await(10, SECONDS), is(true));
    assertThat(seen.get().getAttributeName(), is("cheese")));
    assertThat(seen.get().getCurrentValue(), is("gouda"));

driver.quit();
```

Listen to `console.log` events

Listen to the `console.log` events and register callbacks to process the event.

```
ChromeDriver driver = new ChromeDriver();
DevTools devTools = driver.getDevTools();
devTools.createSession();
devTools.send(Log.enable());
devTools.addListener(Log.entryAdded(),
    logEntry -> {
        System.out.println("log: "+logEntry.getText());
        System.out.println("level: "+logEntry.getLevel())
    });
driver.get("http://the-internet.herokuapp.com/broken_images");
// Check the terminal output for the browser console messages.
driver.quit();
```

Listen to JS Exceptions

Listen to the JS Exceptions and register callbacks to process the exception details.

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.devtools.DevTools;

public void jsExceptionsExample() {
    ChromeDriver driver = new ChromeDriver();
    DevTools devTools = driver.getDevTools();
    devTools.createSession();

    List<JavascriptException> jsExceptionsList = new ArrayList<>();
    Consumer<JavascriptException> addEntry = jsExceptionsList::add;
    devTools.getDomains().events().addJavascriptExceptionListener(addEntry);

    driver.get("<your site url>");

    WebElement link2click = driver.findElement(By.linkText("<your link text>"));
    ((JavascriptExecutor) driver).executeScript("arguments[0].setAttribute(arguments[1], arguments[2])", link2click, "onclick", "throw new Error('Hello, world!')");
    link2click.click();

    for (JavascriptException jsException : jsExceptionsList) {
        System.out.println("JS exception message: " + jsException.getMessage());
        System.out.println("JS exception system information: " + jsException.getSystemInformation());
        jsException.printStackTrace();
    }
}
```

Network Interception

If you want to capture network events coming into the browser and you want manipulate them you are able to do it with the following examples.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.devtools.HasDevTools;
import org.openqa.selenium.devtools.NetworkInterceptor;
import org.openqa.selenium.remote.http.Contents;
import org.openqa.selenium.remote.http.Filter;
import org.openqa.selenium.remote.http.HttpResponse;
import org.openqa.selenium.remote.http.Route;

NetworkInterceptor interceptor = new NetworkInterceptor(
    driver,
    Route.matching(req -> true)
        .to(() -> req -> new HttpResponse()
            .setStatus(200)
            .addHeader("Content-Type", MediaType.HTML_UTF_8.toString())
            .setContent(utf8String("Creamy, delicious cheese!"))));

driver.get("https://example-sausages-site.com");

String source = driver.getPageSource();

assertThat(source).contains("delicious cheese!");
```

2.9.2 - RemoteWebDriver BiDirectional API

The following examples demonstrate how to leverage BiDi APIs with [Remote WebDriver](#).

Register Basic Auth

Some applications make use of browser authentication to secure pages. With Selenium, you can automate the input of basic auth credentials whenever they arise.

Java

```
@Test
public void testBasicAuth() {
    AtomicReference<DevTools> devToolsAtomicReference = new AtomicReference<>();

    driver = new Augmenter().addDriverAugmentation("chrome",
                                                HasAuthentication.class,
                                                (caps, exec) -> (whenThisMatched, response) -> {
                                                    devToolsAtomicReference.get()
                                                        .createSessionIfThereIsNone(response);
                                                    devToolsAtomicReference.get()
                                                        .network()
                                                        .addAuthHandler(whenThisMatched, response)
                                                        .useTheseCredentials(response);
                                                });
}

}).augment(driver);
```

 [Check code on GitHub](#)

Mutation Observation

Mutation Observation is the ability to capture events via WebDriver BiDi when there are DOM mutations on a specific element in the DOM.

Java

```
AtomicReference<WebDriver> augmentedDriver = new AtomicReference<>();
CountDownLatch latch = new CountDownLatch(1);

Augmenter augmenter = new Augmenter();

driver = augmenter.
    addDriverAugmentation("chrome", HasLogEvents.class, (caps, exec) -> new Handler<LogEvent>() {
        @Override
        public <X> void onLogEvent(EventType<X> kind) {
            kind.initializeListener(augmentedDriver.get());
        }
    }).augment(driver);

DevTools devTools = ((HasDevTools) driver).getDevTools();
```

```
devTools.createSession();

if (driver instanceof HasLogEvents) {
    augmentedDriver.set(driver);
} else {
    throw new Exception("Not an instance of HasLogEvents");
}

((HasLogEvents) driver).onLogEvent(domMutation -> {
```

 [Check code on GitHub](#)

Listen to `console.log` events

Listen to the `console.log` events and register callbacks to process the event.

[Java](#)

```
}
```

```
@Test
public void testConsoleLogListener() throws InterruptedException {
    CountDownLatch latch = new CountDownLatch(4);
    driver = new Augmenter().augment(driver);
    DevTools devTools = ((HasDevTools) driver).getDevTools();
    devTools.createSession();

    // Use as per Devtools version
    devTools.send(org.openqa.selenium.devtools.v85.runtime.Runtime.enable());
    devTools.send(Log.enable());

    // https://chromedevtools.github.io/devtools-protocol/tot/Console/ states that
    // Depending on the implementation, events from either of the domains can be

    devTools.addListener(Log.entryAdded(),
        logEntry -> {
            System.out.println("log: " + logEntry.getText());
            System.out.println("level: " + logEntry.getLevel());
            latch.countDown();
        });
}
```

 [Check code on GitHub](#)

Actions causing JS exceptions

[Java](#)

```
}
```

```
@Test
public void testJsExceptionListener() throws Exception {
    driver = new Augmenter().augment(driver);
    DevTools devTools = ((HasDevTools) driver).getDevTools();
    devTools.createSession();
```

Network Interception

If you want to capture network events coming into the browser and you want manipulate them you are able to do it with the following examples.

Java

```
futureJsExc.get(5, TimeUnit.SECONDS);
Assertions.assertEquals(1, jsExceptionsList.size());
}

@Test
public void testNetworkInterceptor() {
    driver = new Augmenter().augment(driver);
    DevTools devTools = ((HasDevTools) driver).getDevTools();
    devTools.createSession();

    // Intercept and change response if the request uri contains "google"
    try (NetworkInterceptor interceptor = new NetworkInterceptor(
        driver,
        Route.matching(req -> req.getUri().contains("google")))
```

 [Check code on GitHub](#)

2.9.3 - Chrome DevTools

While Selenium 4 provides direct access to the Chrome DevTools Protocol (CDP), it is highly encouraged that you use the [WebDriver Bidi APIs](#) instead.

Many browsers provide “DevTools” – a set of tools that are integrated with the browser that developers can use to debug web apps and explore the performance of their pages. Google Chrome’s DevTools make use of a protocol called the Chrome DevTools Protocol (or “CDP” for short). As the name suggests, this is not designed for testing, nor to have a stable API, so functionality is highly dependent on the version of the browser.

WebDriver Bidi is the next generation of the W3C WebDriver protocol and aims to provide a stable API implemented by all browsers, but it's not yet complete. Until it is, Selenium provides access to the CDP for those browsers that implement it (such as Google Chrome, or Microsoft Edge, and Firefox), allowing you to enhance your tests in interesting ways. Some examples of what you can do with it are given below.

Emulate Geo Location

Some applications have different features and functionalities across different locations. Automating such applications is difficult because it is hard to emulate the geo-locations in the browser using Selenium. But with the help of Devtools, we can easily emulate them. Below code snippet demonstrates that.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
ChromeDriver driver = new ChromeDriver();
DevTools devTools = driver.getDevTools();
devTools.createSession();
devTools.send(Emulation.setGeolocationOverride(Optional.of(52.5043),
                                                Optional.of(13.4501),
                                                Optional.of(1)));
driver.get("https://my-location.org/");
driver.quit();
```

Emulate Geo Location with the Remote WebDriver:

Java Python CSharp Ruby JavaScript Kotlin

```
Optional.of(1))));

driver.get("https://my-location.org/");
driver.quit();
```

Override Device Mode

Using Selenium's integration with CDP, one can override the current device mode and simulate a new mode. Width, height, mobile, and deviceScaleFactor are required parameters. Optional parameters include scale, screenWidth, screenHeight, positionX, positionY, dontSetVisible, screenOrientation, viewport, and displayFeature.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
ChromeDriver driver = new ChromeDriver();
DevTools devTools = driver.getDevTools();
devTools.createSession();
// iPhone 11 Pro dimensions
devTools.send(Emulation.setDeviceMetricsOverride(375,
                                                812,
                                                50,
                                                true,
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty(),
                                                Optional.empty()));

driver.get("https://selenium.dev/");
driver.quit();
```

Collect Performance Metrics

Collect various performance metrics while navigating the application.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.devtools.DevTools;

public void performanceMetricsExample() {
    ChromeDriver driver = new ChromeDriver();
    DevTools devTools = driver.getDevTools();
    devTools.createSession();
    devTools.send(Performance.enable(Optional.empty()));
    List<Metric> metricList = devTools.send(Performance.getMetrics());

    driver.get("https://google.com");
    driver.quit();
```

```
for(Metric m : metricList) {  
    System.out.println(m.getName() + " = " + m.getValue());  
}  
}
```

2.10 - Additional features

Set of packages and functionalities to simplify automation with Selenium.

2.10.1 - Working With Colors

You will occasionally want to validate the colour of something as part of your tests; the problem is that colour definitions on the web are not constant. Would it not be nice if there was an easy way to compare a HEX representation of a colour with a RGB representation of a colour, or a RGBA representation of a colour with a HSLA representation of a colour?

Worry not. There is a solution: the *Color* class!

First of all, you will need to import the class:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import org.openqa.selenium.support.Color;
```

You can now start creating colour objects. Every colour object will need to be created from a string representation of your colour. Supported colour representations are:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
private final Color HEX_COLOUR = Color.fromString("#2F7ED8");
private final Color RGB_COLOUR = Color.fromString("rgb(255, 255, 255)");
private final Color RGB_COLOUR = Color.fromString("rgb(40%, 20%, 40%)");
private final Color RGBA_COLOUR = Color.fromString("rgba(255, 255, 255, 0.5)");
private final Color RGBA_COLOUR = Color.fromString("rgba(40%, 20%, 40%, 0.5)");
private final Color HSL_COLOUR = Color.fromString("hsl(100, 0%, 50%)");
private final Color HSLA_COLOUR = Color.fromString("hsla(100, 0%, 50%, 0.5)");
```

The Color class also supports all of the base colour definitions specified in <http://www.w3.org/TR/css3-color/#html4>.

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
private final Color BLACK = Color.fromString("black");
private final Color CHOCOLATE = Color.fromString("chocolate");
private final Color HOTPINK = Color.fromString("hotpink");
```

Sometimes browsers will return a colour value of “transparent” if no colour has been set on an element. The Color class also supports this:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
private final Color TRANSPARENT = Color.fromString("transparent");
```

You can now safely query an element to get its colour/background colour knowing that any response will be correctly parsed and converted into a valid Color object:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
Color loginButtonColour = Color.fromString(driver.findElement(By.id("login")).getBackground());  
Color loginButtonBackgroundColour = Color.fromString(driver.findElement(By.id("login")).getBackground());
```

You can then directly compare colour objects:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
assert loginButtonBackgroundColour.equals(HOTPINK);
```

Or you can convert the colour into one of the following formats and perform a static validation:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
assert loginButtonBackgroundColour.asHex().equals("#ff69b4");  
assert loginButtonBackgroundColour.asRgba().equals("rgba(255, 105, 180, 1)");  
assert loginButtonBackgroundColour.asRgb().equals("rgb(255, 105, 180)");
```

Colours are no longer a problem.

2.10.2 - File Upload

The file upload dialog could be handled using Selenium, when the input element is of type file. An example of it, could be found on this web page- <https://the-internet.herokuapp.com/upload> We will require to have a file available with us, which we need to upload. The code to upload the file for different programming languages will be as follows -

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
import java.util.concurrent.TimeUnit;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import io.github.bonigarcia.wdm.WebDriverManager;
class fileUploadDoc{
    public static void main(String[] args) {
        WebDriverManager.chromedriver().setup();
        WebDriver driver = new ChromeDriver();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        driver.get("https://the-internet.herokuapp.com/upload");
        //we want to import selenium-snapshot file.
        driver.findElement(By.id("file-upload")).sendKeys("selenium-snapshot.jpg");
        driver.findElement(By.id("file-submit")).submit();
        if(driver.getPageSource().contains("File Uploaded!")) {
            System.out.println("file uploaded");
        }
        else{
            System.out.println("file not uploaded");
        }
        driver.quit();
    }
}
```

So the above example code helps to understand how we can upload a file using Selenium.

2.10.3 - ThreadGuard

This class is only available in the Java Binding

ThreadGuard checks that a driver is called only from the same thread that created it. Threading issues especially when running tests in Parallel may have mysterious and hard to diagnose errors. Using this wrapper prevents this category of errors and will raise an exception when it happens.

The following example simulate a clash of threads:

```
public class DriverClash {  
    //thread main (id 1) created this driver  
    private WebDriver protectedDriver = ThreadGuard.protect(new ChromeDriver());  
  
    static {  
        System.setProperty("webdriver.chrome.driver", "<Set path to your Chromedriver");  
    }  
  
    //Thread-1 (id 24) is calling the same driver causing the clash to happen  
    Runnable r1 = () -> {protectedDriver.get("https://selenium.dev");};  
    Thread thr1 = new Thread(r1);  
  
    void runThreads(){  
        thr1.start();  
    }  
  
    public static void main(String[] args) {  
        new DriverClash().runThreads();  
    }  
}
```

The result shown below:

```
Exception in thread "Thread-1" org.openqa.selenium.WebDriverException:  
Thread safety error; this instance of WebDriver was constructed  
on thread main (id 1) and is being accessed by thread Thread-1 (id 24)  
This is not permitted and *will* cause undefined behaviour
```

As seen in the example:

- protectedDriver Will be created in Main thread
- We use Java Runnable to spin up a new process and a new Thread to run the process
- Both Thread will clash because the Main Thread does not have protectedDriver in it's memory.
- ThreadGuard.protect will throw an exception.

Note:

This does not replace the need for using ThreadLocal to manage drivers when running parallel.

3 - Selenium Grid 4

Want to run tests in parallel across multiple machines? Then, Grid is for you.

Selenium Grid allows the execution of WebDriver scripts on remote machines (virtual or real) by routing commands sent by the client to remote browser instances. It aims to provide an easy way to run tests in parallel on multiple machines.

Selenium Grid allows us to run tests in parallel on multiple machines, and to manage different browser versions and browser configurations centrally (instead of in each individual test).

Selenium Grid is not a silver bullet. It solves a subset of common delegation and distribution problems, but will for example not manage your infrastructure, and might not suit your specific needs.

Purposes and main functionalities

- Central entry point for all tests
- Management and control of the nodes / environment where the browsers run
- Scaling
- Running tests in parallel
- Cross-platform testing
- Load balancing

Selenium Grid 4

Grid 4 takes advantage of a number of new technologies in order to facilitate scaling up while allowing local execution.

Selenium Grid 4 is a fresh implementation and does not share the codebase the previous version had.

To get all the details of Grid 4 components, understand how it works, and how to set up your own, please browse thorough the following sections.

3.1 - Getting started with Selenium Grid

Instructions, step by step, showing how to run a simple Selenium Grid.

Grid roles

Several [components](#) compose a Selenium Grid. Depending on your needs, you can start each one of them on its own, or a few at the same time by using a Grid role.

Standalone

Standalone is the union of all components, and to the user's eyes, they are executed as one. A fully functional Grid of one is available after starting it in the Standalone mode.

Standalone is also the easiest mode to spin up a Selenium Grid. By default, the server will be listening on `http://localhost:4444`, and that's the URL you should point your `RemoteWebDriver` tests. The server will detect the available drivers that it can use from the System PATH.

```
java -jar selenium-server-<version>.jar standalone
```

Hub and Node(s)

It enables the classic Hub & Node(s) setup. These roles are suitable for small and middle-sized Grids.

Hub

A Hub is the union of the following components:

- Router
- Distributor
- Session Map
- New Session Queue
- Event Bus

```
java -jar selenium-server-<version>.jar hub
```

By default, the server will be listening on `http://localhost:4444`, and that's the URL you should point your `RemoteWebDriver` tests.

Node(s)

One or more Nodes can be started in this setup, and the server will detect the available drivers that it can use from the System PATH.

```
java -jar selenium-server-<version>.jar node
```

Distributed

On Distributed mode, each component needs to be started on its own. This setup is more suitable for large Grids.

The startup order of the components is not important, however, we recommend following these steps when starting a distributed Grid.

1. Event Bus: serves as a communication path to other Grid components in subsequent steps.

```
java -jar selenium-server-<version>.jar event-bus
```

2. Session Map: responsible for mapping session IDs to the Node where the session is running.

```
java -jar selenium-server-<version>.jar sessions
```

3. New Session Queue: adds the new session request to a queue, then the distributor processes it.

```
java -jar selenium-server-<version>.jar sessionqueue
```

4. Distributor: Nodes register to it, and assigns a Node for a session request.

```
java -jar selenium-server-<version>.jar distributor --sessions http://localhost:
```

5. Router: the Grid entrypoint, in charge of redirecting requests to the right component.

```
java -jar selenium-server-<version>.jar router --sessions http://localhost:5556
```

6. Node(s)

```
java -jar selenium-server-<version>.jar node
```

Running tests

To run tests after starting successfully Selenium Grid, you can use a `RemoteWebDriver`. Head to the [RemoteWebDriver section](#) for more details.

Tests metadata

You can add metadata to your tests and consume it via [GraphQL](#) or visualize parts of it through the Selenium Grid UI. Metadata can be added by prefixing the metadata with `se:`.

Here is a quick example in Java showing how to do that.

```
ChromeOptions chromeOptions = new ChromeOptions();
chromeOptions.setCapability("browserVersion", "100");
chromeOptions.setCapability("platformName", "Windows");
// Showing a test name instead of the session id in the Grid UI
chromeOptions.setCapability("se:name", "My simple test");
// Other type of metadata can be seen in the Grid UI by clicking on the
// session info or via GraphQL
chromeOptions.setCapability("se:sampleMetadata", "Sample metadata value");
WebDriver driver = new RemoteWebDriver(new URL("http://gridUrl:4444"), chromeOpt
```

```
driver.get("http://www.google.com");
driver.quit();
```

Querying Selenium Grid

After starting a Grid, there are mainly two ways of querying its status, through the Grid UI or via an API call.

The Grid UI can be reached by opening your preferred browser and heading to <http://localhost:4444>.

API calls can be done through the <http://localhost:4444/status> endpoint or using [GraphQL](#)

For simplicity, all command examples shown in this page assume that components are running locally. More detailed examples and usages can be found in the [Configuring Components](#) section.

Warning

Selenium Grid must be protected from external access using appropriate firewall permissions.

Failure to protect your Grid could result in one or more of the following occurring:

- You provide open access to your Grid infrastructure
- You allow third parties to access internal web applications and files
- You allow third parties to run custom binaries

See this blog post on [Detectify](#), which gives a good overview of how a publicly exposed Grid could be misused: [Don't Leave your Grid Wide Open](#)

3.2 - When to Use a Grid

Is Grid the right tool for you?

Generally speaking, there's two reasons why you might want to use 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.

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 Grid to support 4 different machines (VMs or separate physical 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. For large test suites, and long-running test suite such as those performing large amounts of data-validation, this can be a significant time-saver. Some test suites can take hours to run. Another reason to boost the time spent running the suite is to shorten the turnaround time for test results after developers check-in code for the AUT. Increasingly software teams practicing Agile software development want test feedback as immediately as possible as opposed to wait overnight for an overnight test pass.

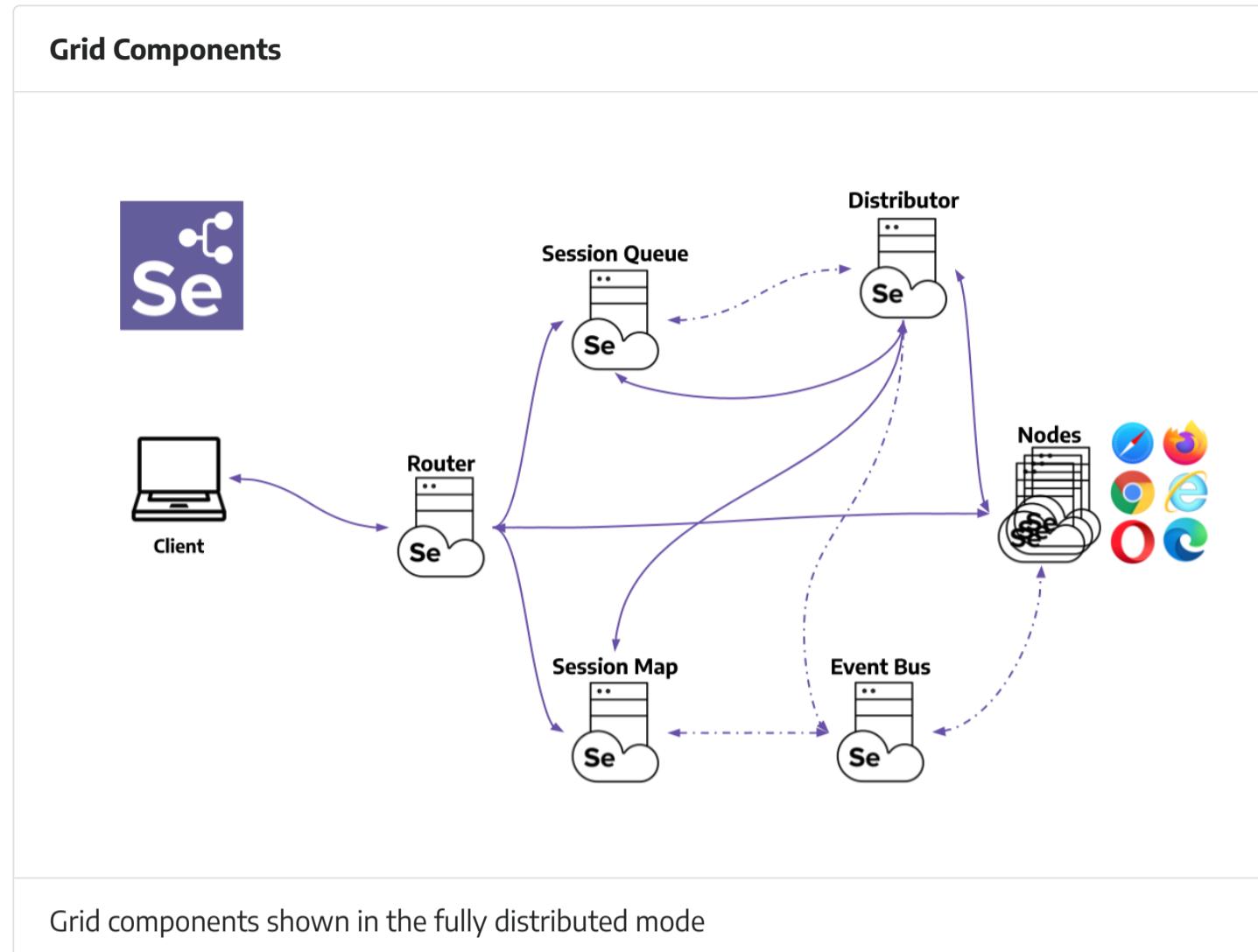
Grid is also used to support running tests against multiple runtime environments, specifically, against different browsers at the same time. For example, a 'grid' of virtual machines can be setup with each supporting a different browser that the application to be tested must support. So, machine 1 has Internet Explorer 8, machine 2, Internet Explorer 9, machine 3 the latest Chrome, and machine 4 the latest Firefox. When the test suite is run, Selenium-Grid receives each test-browser combination and assigns each test to run against its required browser.

In addition, one can have a grid of all the same browser, type and version. For instance, one could have a grid of 4 machines each running 3 instances of Firefox 70, allowing for a 'server-farm' (in a sense) of available Firefox instances. When the suite runs, each test is passed to Grid which assigns the test to the next available Firefox instance. In this manner one gets test pass where conceivably 12 tests are all running at the same time in parallel, significantly reducing the time required to complete a test pass.

Grid is very flexible. These two examples can be combined to allow multiple instances of each browser type and version. A configuration such as this would provide both, parallel execution for fast test pass completion and support for multiple browser types and versions simultaneously.

3.3 - Selenium Grid Components

Check the different Grid components to understand how to use them.



Router

The Router takes care of forwarding the request to the correct component.

It is the entry point of the Grid, all external requests will be received by it. The Router behaves differently depending on the request. If it is a new session request, the Router will add it to the New Session Queue. The Distributor regularly checks if there is a free slot. If so, the first matching request is removed from the New Session Queue. The Router will receive the event and poll the New Session Queue to get the new session request. If the request belongs to an existing session, the Router will send the session id to the Session Map, and the Session Map will return the Node where the session is running. After this, the Router will forward the request to the Node.

The Router aims to balance the load in the Grid by sending the requests to the component that is able to handle them better, without overloading any component that is not needed in the process.

Distributor

The Distributor is aware of all the Nodes and their capabilities. Its main role is to receive a new session request and find a suitable Node where the session can be created. After the session is created, the Distributor stores in the Session Map the relation between the session id and Node where the session is being executed.

Node

A Node can be present several times in a Grid. Each Node takes care of managing the slots for the available browsers of the machine where it is running.

The Node registers itself to the Distributor through the Event Bus, and its configuration is sent as part of the registration message.

By default, the Node auto-registers all browser drivers available on the path of the machine where it runs. It also creates one slot per available CPU for Chromium based browsers and Firefox. For Safari and Internet Explorer, only one slot is created. Through a specific configuration, it can run sessions in Docker containers or relay commands. You can see more configuration details in [setting up your own grid](#).

A Node only executes the received commands, it does not evaluate, make judgments, or control anything. The machines where the Node is running does not need to have the same operating system as the other components. For example, A Windows Node might have the capability of offering Internet Explorer as a browser option, whereas this would not be possible on Linux or Mac.

Session Map

The Session Map is a data store that keeps the information of the session id and the Node where the session is running. It serves as a support for the Router in the process of forwarding a request to the Node. The Router will ask the Session Map for the Node associated to a session id.

New Session Queue

New Session Queue holds all the new session requests in a FIFO order. It has configurable parameters for setting the request timeout and request retry interval.

The Router adds the new session request to the New Session Queue and waits for the response. The New Session Queue regularly checks if any request in the queue has timed out, if so the request is rejected and removed immediately.

The Distributor regularly checks if a slot is available. If so, the Distributor requests the New Session Queue for the first matching request. The Distributor then attempts to create a new session.

Once the requested capabilities match the capabilities of any of the free Node slots, the Distributor attempts to get the available slot. If all the slots are busy, the Distributor will ask the queue to add the request to the front of the queue. If request times out while retrying or adding to the front of the queue it is rejected.

After a session is created successfully, the Distributor sends the session information to the New Session Queue. The New Session Queue sends the response back to the client.

Event Bus

The Event Bus serves as a communication path between the Nodes, Distributor, New Session Queue, and Session Map. The Grid does most of its internal communication through messages, avoiding expensive HTTP calls. When starting the Grid in its fully distributed mode, the Event Bus is the first component that should be started.

Running your own Grid

Looking forward to using all these components and run your own Grid? Head to the [“Getting Started”](#) section to understand how to put all these pieces together.

3.4 - Configuration of Components

Here you can see how each Grid component can be configured individually based on common configuration values and component-specific configuration values.

3.4.1 - Configuration help

Get information about all the available options to configure Grid.

The help commands display information based on the current code implementation. Hence, it will provide accurate information in case the documentation is not updated. It is the easiest way to learn about Grid 4 configuration for any new version.

Info Command

The info command provides detailed docs on the following topics:

- Configuring Selenium
- Security
- Session Map setup
- Tracing

Config help

Quick config help and overview is provided by running:

```
java -jar selenium-server-<version>.jar info config
```

Security

To get details on setting up the Grid servers for secure communication and node registration:

```
java -jar selenium-server-<version>.jar info security
```

Session Map setup

By default, Grid uses a local session map to store session information. Grid supports additional storage options like Redis and JDBC - SQL supported databases. To set up different session storage, use the following command to get setup steps:

```
java -jar selenium-server-<version>.jar info sessionmap
```

Setting up tracing with OpenTelemetry and Jaeger

By default, tracing is enabled. To export traces and visualize them via Jaeger, use the following command for instructions:

```
java -jar selenium-server-<version>.jar info tracing
```

List the Selenium Grid commands

```
java -jar selenium-server-<version>.jar --config-help
```

It will show all the available commands and description for each one.

Component help commands

Pass –help config option after the Selenium role to get component-specific config information.

Standalone

```
java -jar selenium-server-<version>.jar standalone --help
```

Hub

```
java -jar selenium-server-<version>.jar hub --help
```

Sessions

```
java -jar selenium-server-<version>.jar sessions --help
```

New Session Queue

```
java -jar selenium-server-<version>.jar sessionqueue --help
```

Distributor

```
java -jar selenium-server-<version>.jar distributor --help
```

Router

```
java -jar selenium-server-<version>.jar router --help
```

Node

```
java -jar selenium-server-<version>.jar node --help
```

3.4.2 - CLI options in the Selenium Grid

All Grid components configuration CLI options in detail.

Different sections are available to configure a Grid. Each section has options can be configured through command line arguments.

A complete description of the component to section mapping can be seen below.

Note that this documentation could be outdated if an option was modified or added but has not been documented yet. In case you bump into this situation, please check the "[Config help](#)" section and feel free to send us a pull request updating this page.

Sections

	Standalone	Hub	Node	Distributor	Router	Sessions	SessionQueue
Distributor	✓	✓		✓	✓		
Docker	✓		✓				
Events		✓	✓	✓		✓	✓
Logging	✓	✓	✓	✓	✓	✓	✓
Network	✓	✓			✓		
Node	✓		✓				
Router	✓	✓			✓		
Relay	✓		✓				
Server	✓	✓	✓	✓	✓	✓	✓
SessionQueue	✓	✓		✓	✓		✓
Sessions				✓	✓	✓	

Distributor

Option	Type	Value/Example
--grid-model	string	org.openqa.selenium.grid.distributor.GridModel

Option	Type	Value/Example
--healthcheck-interval	int	120
--distributor-uri	uri	http://localhost:5553
--distributor-host	string	localhost
--distributor-implementation	string	org.openqa.selenium.grid.distributor.local.LocalDistributor
--distributor-port	int	5553
--reject-unsupported-caps	boolean	false
--slot-matcher	string	org.openqa.selenium.grid.data.DefaultSlotMatcher

Option	Type	Value/Example
--slot-selector	string	org.openqa.selenium.grid.distributor.selector.DefaultSlotSelector



Docker

Option	Type	Value/Example	Description
--docker-assets-path	string	/opt/selenium/assets	Absolute path where assets will be stored
--docker-caps	string[]	selenium/standalone-firefox:latest '{"browserName": "firefox"}'	Docker configs which map image name to stereotype capabilities (example ` -D selenium/standalone-firefox:latest '{"browserName": "firefox"}'`)
--docker-devices	string[]	/dev/kvm:/dev/kvm	Exposes devices to a container. Each device mapping declaration must have at least the path of the device in both host and container separated by a colon like in this example: /device/path/in/host:/device/path/in/container
--docker-host	string	localhost	Host name where the Docker daemon is running
--docker-port	int	2375	Port where the Docker daemon is running
--docker-url	string	http://localhost:2375	URL for connecting to the Docker daemon
--docker-video-image	string	selenium/video:latest	Docker image to be used when video recording is enabled

Events

Option	Type	Value/Example	Description

Option	Type	Value/Example	Description
--bind-bus	boolean	false	Whether the connection string should be bound or connected. When true, the component will be bound to the Event Bus (as in the EventBus will also be started by the component, typically by the Distributor and the Hub). When false, the component will connect to the Event Bus.
--events-implementation	string	org.openqa.selenium.events.zeromq.ZeroMqEventBus	Full class name of non-default event bus implementation
--publish-events	string	tcp://*:4442	Connection string for publishing events to the event bus
--subscribe-events	string	tcp://*:4443	Connection string for subscribing to events from the event bus

[◀](#) [▶](#)

Logging

Option	Type	Value/Example	Description
--http-logs	boolean	false	Enable http logging. Tracing should be enabled.
--log-encoding	string	UTF-8	Log encoding
--log	string	Windows path example : '\path\to\file\gridlog.log' or 'C:\path\path\to\file\gridlog.log'	File to write out logs. Ensure the operating system's file path.
		Linux/Unix/MacOS path example : '/path/to/file/gridlog.log'	

Option	Type	Value/Example	Description
--log-level	string	"INFO"	Log level. Default logging level is here https://docs.oracle.com/javase/7/d
--plain-logs	boolean	true	Use plain log lines
--structured-logs	boolean	false	Use structured logs
--tracing	boolean	true	Enable trace collection
--log-timestamp-format	string	HH:mm:ss.SSS	Allows the configure log timestamp

Network

Option	Type	Value/Example	Description
--relax-checks	boolean	false	Relax checks on origin header and content type of incoming requests, in contravention of strict W3C spec compliance.

Node

Option	Type	Value/Example	Description
--detect-drivers	boolean	true	Autodetect drivers from the current environment and add them to the Node.
--driver-configuration	string[]	display-name="Firefox Nightly" max-sessions=2 webdriver-path="/usr/local/bin/geckodriver" stereotype='{"browserName": "firefox", "browserVersion": "86", "moz:firefoxOptions": {"binary": "/Applications/Firefox Nightly.app/Contents/MacOS/firefox-bin"}'}	List of configuration files for supported drivers; recommend providing a toml configuration file to improve detection.
--driver-factory	string[]	org.openqa.selenium.example.LynxDriverFactory '{"browserName": "lynx"}'	Mapping of qualified driver factories to browser configurations that matches the browser name.
--driver-implementation	string[]	"firefox"	Drivers that have been checked against the browser will skip autoconfiguration.

Option	Type	Value/Example	Description
--node-implementation	string	"org.openqa.selenium.grid.node.local.LocalNodeFactory"	Full class name of the node implementation used to create sessions.
--grid-url	string	https://grid.example.com	Public URL of the grid hub as a whole, including the address or the RDP port.
--heartbeat-period	int	60	How often will the heartbeat message be sent to the Distribution Center to inform it that the host is up.
--max-sessions	int	8	Maximum number of concurrent sessions. Default value is 8.
--override-max-sessions	boolean	false	The # of processes recommended by the browser processor flag to tell the distribution center the value to override the stability limit might suggest host configuration resources.
--register-cycle	int	10	How often the Node will register itself for the first time with the Distribution Center.
--register-period	int	120	How long will the Node register with the Distribution Center. After this time, if the Node is still active, it will not register again.

Option	Type	Value/Example	Description
--session-timeout	int	300	Let X be timeout. The Node will automatically end session if it had any last X seconds. It will release other terminals.
--vnc-env-var	string	START_XVFB	Environment variable to check to determine if VNC stream is active or not.
--no-vnc-port	int	7900	If VNC is running on the port, the port will be released locally so no other node can be connected.
--drain-after-session-count	int	1	Drain after the Node has terminated sessions. If the Node is executing a command in an environment in Kubernetes, the higher the value, the longer it enables the command to run.
--hub	string	http://localhost:4444	The address in a Hub configuration. It can be a hostname, address, which can be a URL (http://), the --g flag, the same events ('tcp://host:port') and --s events ('tcp://host:port'). If hostnames and port numbers are used, the URL but the events remain the same. One of these may be set by setting the flags. If it has a protocol (https) it is used to connect to the hub.

Option	Type	Value/Example	Description
--enable-cdp	boolean	true	Enable Cross-Domain Protocol (CDP) support. A CDP endpoint will be available at /wd/hub/session/<session_id>/cdp. If false, disable CDP support. Default: true.

Relay

Option	Type	Value/Example	Description
--service-url	string	http://localhost:4723	URL for connecting to the service that supports WebDriver commands like an Appium server or a cloud service.
--service-host	string	localhost	Host name where the service that supports WebDriver commands is running
--service-port	int	4723	Port where the service that supports WebDriver commands is running
--service-status-endpoint	string	/status	Optional, endpoint to query the WebDriver service status, an HTTP 200 response is expected
--service-configuration	string[]	max-sessions=2 stereotype='{"browserName": "safari", "platformName": "iOS", "appium:platformVersion": "14.5"}'	Configuration for the service where calls will be relayed to. It is recommended to provide this type of configuration through a toml config file to improve readability.

Router

Option	Type	Value/Example	Description
--password	string	myStrongPassword	Password clients must use to connect to the server. Both this and the username need to be set in order to be used.
--username	string	admin	User name clients must use to connect to the server. Both this and the password need to be set in order to be used.

Server

Option	Type	Value/Example	Description
--allow-cors	boolean	true	Whether the Selenium server should allow web browser connections from any host

Option	Type	Value/Example	Description
--host	string	localhost	Server IP or hostname: usually determined automatically.
--bind-host	boolean	true	Whether the server should bind to the host address/name, or only use it to report its reachable url. Helpful in complex network topologies where the server cannot report itself with the current IP/hostname but rather an external IP or hostname (e.g. inside a Docker container)
--https-certificate	path	/path/to/cert.pem	Server certificate for https. Get more detailed information by running "java -jar selenium-server.jar info security"
--https-private-key	path	/path/to/key.pkcs8	Private key for https. Get more detailed information by running "java -jar selenium-server.jar info security"
--max-threads	int	24	Maximum number of listener threads. Default value is: (available processors) * 3.
--port	int	4444	Port to listen on. There is no default as this parameter is used by different components, for example, Router/Hub/Standalone will use 4444 and Node will use 5555.

SessionQueue

Option	Type	Value/Example	Description
--sessionqueue	uri	http://localhost:1237	Address of the session queue server.
-sessionqueue-host	string	localhost	Host on which the session queue server is listening.
--sessionqueue-port	int	1234	Port on which the session queue server is listening.
--session-request-timeout	int	300	Timeout in seconds. A new incoming session request is added to the queue. Requests sitting in the queue for longer than the configured time will timeout.
--session-retry-interval	int	5	Retry interval in seconds. If all slots are busy, new session request will be retried after the given interval.

Sessions

Option	Type	Value/Example	Description
--sessions	uri	http://localhost:1234	Address of the session map server.

Option	Type	Value/Example	Description
--sessions-host	string	localhost	Host on which the session map server is listening.
--sessions-port	int	1234	Port on which the session map server is listening.

Configuration examples

All the options mentioned above can be used when starting the Grid components. They are a good way of exploring the Grid options, and trying out values to find a suitable configuration.

We recommend the use of [Toml files](#) to configure a Grid. Configuration files improve readability, and you can also check them in source control.

When needed, you can combine a Toml file configuration with CLI arguments.

Command-line flags

To pass config options as command-line flags, identify the valid options for the component and follow the template below.

```
java -jar selenium-server-<version>.jar <component> --<option> value
```

Standalone, setting max sessions and main port

```
java -jar selenium-server-<version>.jar standalone --max-sessions 4 --port 4444
```

Hub, setting a new session request timeout, a main port, and disabling tracing

```
java -jar selenium-server-<version>.jar hub --session-request-timeout 500 --port 3333 --trac
```

Node, with 4 max sessions, with debug(fine) log, 7777 as port, and only with Firefox and Edge

```
java -jar selenium-server-<version>.jar node --max-sessions 4 --log-level "fine" --port 7777
```

Distributor, setting Session Map server url, Session Queue server url, and disabling bus

```
java -jar selenium-server-<version>.jar distributor --sessions http://localhost:5556 --sessi
```

Setting custom capabilities for matching specific Nodes

Important: Custom capabilities need to be set in the configuration in all Nodes. They also need to be included always in every session request.

Start the Hub

```
java -jar selenium-server-<version>.jar hub
```

Start the Node A with custom cap set to **true**

```
java -jar selenium-server-<version>.jar node --detect-drivers false --driver-configuration c
```

Start the Node B with custom cap set to **false**

```
java -jar selenium-server-<version>.jar node --detect-drivers false --driver-configuration c
```

Matching Node A

```
ChromeOptions options = new ChromeOptions();
options.setCapability("gsg:customcap", true);
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444"), options);
driver.get("https://selenium.dev");
driver.quit();
```

Set the custom capability to `false` in order to match the Node B.

3.4.3 - TOML configuration options

Grid configuration examples using Toml files.

All the options shown in [CLI options](#) can be configured through a [TOML](#) file. This page shows configuration examples for the different Grid components.

Note that this documentation could be outdated if an option was modified or added but has not been documented yet. In case you bump into this situation, please check the “[Config help](#)” section and feel free to send us a pull request updating this page.

Overview

Selenium Grid uses [TOML](#) format for config files. The config file consists of sections and each section has options and its respective value(s).

Refer to the [TOML documentation](#) for detailed usage guidance. In case of parsing errors, validate the config using [TOML linter](#).

The general configuration structure has the following pattern:

```
[section1]
option1="value"

[section2]
option2=["value1", "value2"]
option3=true
```

Below are some examples of Grid components configured with a Toml file, the component can be started in the following way:

```
java -jar selenium-server-<version>.jar <component> --config /path/to/file/<file-name>.toml
```

Standalone

A Standalone server, running on port 4449, and a new session request timeout of 500 seconds.

```
[server]
port = 4449

[sessionqueue]
session-request-timeout = 500
```

Specific browsers and a limit of max sessions

A Standalone server or a Node which only has Firefox and Chrome enabled by default.

```
[node]
drivers = ["chrome", "firefox"]
max-sessions = 3
```

Configuring and customising drivers

Standalone or Node server with customised drivers, which allows things like having Firefox Beta or Nightly, and having different browser versions.

```
[node]
detect-drivers = false
[[node.driver-configuration]]
max-sessions = 100
display-name = "Firefox Nightly"
stereotype = "{\"browserName\": \"firefox\", \"browserVersion\": \"93\", \"platf
[[node.driver-configuration]]
display-name = "Chrome Beta"
stereotype = "{\"browserName\": \"chrome\", \"browserVersion\": \"94\", \"platfo
[[node.driver-configuration]]
display-name = "Chrome Dev"
stereotype = "{\"browserName\": \"chrome\", \"browserVersion\": \"95\", \"platfo
webdriver-executable = '/path/to/chromedriver/95/chromedriver'
```

Standalone or Node with Docker

A Standalone or Node server that is able to run each new session in a Docker container. Disabling drivers detection, having maximum 2 concurrent sessions. Stereotypes configured need to be mapped to a Docker image, and the Docker daemon needs to be exposed via http/tcp. In addition, it is possible to define which device files, accessible on the host, will be available in containers through the `devices` property. Refer to the [docker](#) documentation for more information about how docker device mapping works.

```
[node]
detect-drivers = false
max-sessions = 2

[docker]
configs = [
    "selenium/standalone-chrome:93.0", "{\"browserName\": \"chrome\", \"browserv
    "selenium/standalone-firefox:92.0", "{\"browserName\": \"firefox\", \"brows
]
# Optionally define all device files that should be mapped to docker containers
#devices = [
#    "/dev/kvm:/dev/kvm"
#]
url = "http://localhost:2375"
video-image = "selenium/video:latest"
```

Relaying commands to a service endpoint that supports WebDriver

It is useful to connect an external service that supports WebDriver to Selenium Grid. An example of such service could be a cloud provider or an Appium server. In this way, Grid can enable more coverage to platforms and versions not present locally.

The following is an example of connecting an Appium server to Grid.

```
[server]
port = 5555
```

```

[node]
detect-drivers = false

[relay]
# Default Appium server endpoint
url = "http://localhost:4723/wd/hub"
status-endpoint = "/status"
# Stereotypes supported by the service
configs = [
    "1", "{\"browserName\": \"chrome\", \"platformName\": \"android\", \"appium:pl"
]

```

Basic auth enabled

It is possible to protect a Grid with basic auth by configuring the Router/Hub/Standalone with a username and password. This user/password combination will be needed when loading the Grid UI or starting a new session.

```

[router]
username = "admin"
password = "myStrongPassword"

```

Here is a Java example showing how to start a session using the configured user and password.

```

URL gridUrl = new URL("http://admin:myStrongPassword@localhost:4444");
RemoteWebDriver webDriver = new RemoteWebDriver(gridUrl, new ChromeOptions());

```

Setting custom capabilities for matching specific Nodes

Important: Custom capabilities need to be set in the configuration in all Nodes. They also need to be included always in every session request.

```

[node]
detect-drivers = false

[[node.driver-configuration]]
display-name = "firefox"
stereotype = '{"browserName": "firefox", "platformName": "macOS", "browserVersion": "96", "max-sessions": 5}

```

Here is a Java example showing how to match that Node

```

FirefoxOptions options = new FirefoxOptions();
options.setCapability("networkname:applicationName", "node_1");
options.setCapability("nodename:applicationName", "app_1");
options.setBrowserVersion("96");
options.setPlatformName("macOS");
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444"), options);
driver.get("https://selenium.dev");
driver.quit();

```

3.5 - Grid architecture

The Grid is designed as a set of components that all fulfill a role in maintaining the Grid. It can seem quite complicated, but hopefully this document can help clear up any confusion.

The Key Components

The main components of the Grid are:

Event Bus

Used for sending messages which may be received asynchronously between the other components.

Session Queue

Maintains a list of incoming sessions which have yet to be assigned to a Node by the Distributor.

Distributor

Responsible for maintaining a model of the available locations in the Grid where a session may run (known as "slots") and taking any incoming [new session](#) requests and assigning them to a slot.

Node

Runs a [WebDriver session](#). Each session is assigned to a slot, and each node has one or more slots.

Session Map

Maintains a mapping between the [session ID](#) and the address of the Node the session is running on.

Router

Acts as the front-end of the Grid. This is the only part of the Grid which _may_ be exposed to the wider Web (though we strongly caution against it). This routes incoming requests to either the New Session Queue or the Node on which the session is running.

While discussing the Grid, there are some other useful concepts to keep in mind:

- A **slot** is the place where a session can run.
- Each slot has a **stereotype**. This is the minimal set of capabilities that a [new session](#) session request must match before the Distributor will send that request to the Node owning the slot.
- The **Grid Model** is how the Distributor tracks the state of the Grid. As the name suggests, this may sometimes fall out of sync with reality (perhaps because the Distributor has only just started). It is used in preference to querying each Node so that the Distributor can quickly assign a slot to a New Session request.

Synchronous and Asynchronous Calls

There are two main communication mechanisms used within the Grid:

1. Synchronous “REST-ish” JSON over HTTP requests.
2. Asynchronous events sent to the Event Bus.

How do we pick which communication mechanism to use? After all, we could model the entire Grid in an event-based way, and it would work out just fine.

The answer is that if the action being performed is synchronous (eg. most WebDriver calls), or if missing the response would be problematic, the Grid uses a synchronous call. If, instead, we want to broadcast information to anyone who's interested, or if missing the response doesn't matter, then we prefer to use the event bus.

One interesting thing to note is that the async calls are more decoupled from their listeners than the synchronous calls are.

Start Up Sequence and Dependencies Between Components

Although the Grid is designed to allow components to start up in any order, conceptually the order in which components starts is:

1. The Event Bus and Session Map start first. These have no other dependencies, not even on each other, and so are safe to start in parallel.
2. The Session Queue starts next.
3. It is now possible to start the Distributor. This will periodically connect to the Session Queue and poll for jobs, though this polling might be initiated either by an event (that a New Session has been added to the queue) or at regular intervals.
4. The Router(s) can be started. New Session requests will be directed to the Session Queue, and the Distributor will attempt to find a slot to run the session on.
5. We are now able to start a Node. See below for details about how the Node is registered with the Grid. Once registration is complete, the Grid is ready to serve traffic.

You can picture the dependencies between components this way, where a “” indicates that there is a synchronous dependency between the components.

	Event Bus	Distributor	Node	Router	Session Map	Session Queue
Event Bus	X					
Distributor	<input checked="" type="checkbox"/>	X	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>
Node	<input checked="" type="checkbox"/>		X			
Router		<input checked="" type="checkbox"/>	X	<input checked="" type="checkbox"/>		
Session Map					X	
Session Queue	<input checked="" type="checkbox"/>					X

Node Registration

The process of registering a new Node to the Grid is lightweight.

1. When the Node starts, it should emit a “heart beat” event on a regular basis. This heartbeat contains the [node status](#).
2. The Distributor listens for the heart beat events. When it sees one, it attempts to `GET` the `/status` endpoint of the Node. It is from this information that the Grid is set up.

The Distributor will use the same `/status` endpoint to check the Node on a regular basis, but the Node should continue sending heart beat events even after started so that a Distributor without a persistent store of the Grid state can be restarted and will (eventually) be up to date and correct.

The Node Status Object

The Node Status is a JSON blob with the following fields:

Name	Type	Description
availability	string	A string which is one of <code>up</code> , <code>draining</code> , or <code>down</code> . The important one is <code>draining</code> , which indicates that no new sessions should be sent to the Node, and once the last session on it closes, the Node will exit or restart.
externalUrl	string	The URI that the other components in the Grid should connect to.
lastSessionCreated	integer	The epoch timestamp of when the last session was created on this Node. The Distributor will attempt to send new sessions to the Node that has been idle longest if all other things are equal.

Name	Type	Description
maxSessionCount	integer	Although a session count can be inferred by counting the number of available slots, this integer value is used to determine the maximum number of sessions that should be running simultaneously on the Node before it is considered “full”.
nodeId	string	A UUID used to identify this instance of the Node.
osInfo	object	An object with <code>arch</code> , <code>name</code> , and <code>version</code> fields. This is used by the Grid UI and the GraphQL queries.
slots	array	An array of Slot objects (described below)
version	string	The version of the Node (for Selenium, this will match the Selenium version number)

It is recommended to put values in all fields.

The Slot Object

The Slot object represents a single slot within a Node. A “slot” is where a single session may be run. It is possible that a Node will have more slots than it can run concurrently. For example, a node may be able to run up 10 sessions, but they could be any combination of Chrome, Edge, or Firefox; in this case, the Node would indicate a “max session count” of 10, and then also say it has 10 slots for Chrome, 10 for Edge, and 10 for Firefox.

Name	Type	Description
id	string	UUID to refer to the slot
lastStarted	string	When the slot last had a session started, in ISO-8601 format
stereotype	object	The minimal set of capabilities this slot will match against. A minimal example is <code>{"browserName": "firefox"}</code>
session	object	The Session object (see below)

The Session Object

This represents a running session within a slot

Name	Type	Description
capabilities	object	The actual capabilities provided by the session. Will match the return value from the new session command
startTime	string	The start time of the session in ISO-8601 format
stereotype	object	The minimal set of capabilities this slot will match against. A minimal example is <code>{"browserName": "firefox"}</code>
uri	string	The URI used by the Node to communicate with the session

3.6 - Advanced features of Selenium

To get all the details of the advanced features, understand how it works, and how to set up your own, please browse thorough the following sections.

3.6.1 - Observability in Selenium Grid

Table of Contents

- [Selenium Grid](#)
- [Observability](#)
 - [Distributed tracing](#)
 - [Event logging](#)
- [Grid Observability](#)
 - [Visualizing Traces](#)
 - [Leveraging event logs](#)
- [References](#)

Selenium Grid

Grid aids in scaling and distributing tests by executing tests on various browser and operating system combinations.

Observability

Observability has three pillars: traces, metrics and logs. Since Selenium Grid 4 is designed to be fully distributed, observability will make it easier to understand and debug the internals.

Distributed tracing

A single request or transaction spans multiple services and components. Tracing tracks the request lifecycle as each service executes the request. It is useful in debugging in an error scenario. Some key terms used in tracing context are:

Trace Tracing allows one to trace a request through multiple services, starting from its origin to its final destination. This request's journey helps in debugging, monitoring the end-to-end flow, and identifying failures. A trace depicts the end-to-end request flow. Each trace has a unique id as its identifier.

Span Each trace is made up of timed operations called spans. A span has a start and end time and it represents operations done by a service. The granularity of span depends on how it is instrumented. Each span has a unique identifier. All spans within a trace have the same trace id.

Span Attributes Span attributes are key-value pairs which provide additional information about each span.

Events Events are timed-stamped logs within a span. They provide additional context to the existing spans. Events also contain key-value pairs as event attributes.

Event logging

Logging is essential to debug an application. Logging is often done in a human-readable format. But for machines to search and analyze the logs, it has to have a well-defined format. Structured logging is a common practice of recording logs consistently in a fixed format. It commonly contains fields like:

- Timestamp
- Logging level
- Logger class
- Log message (This is further broken down into fields relevant to the operation where the log was recorded)

Logs and events are closely related. Events encapsulate all the possible information available to do a single unit of work. Logs are essentially subsets of an event. At the crux, both aid in debugging. Refer following resources for detailed understanding:

1. <https://www.honeycomb.io/blog/how-are-structured-logs-different-from-events/>
2. <https://charity.wtf/2019/02/05/logs-vs-structured-events/>

Grid Observability

Selenium server is instrumented with tracing using OpenTelemetry. Every request to the server is traced from start to end. Each trace consists of a series of spans as a request is executed within the server. Most spans in the Selenium server consist of two events:

1. Normal event - records all information about a unit of work and marks successful completion of the work.
2. Error event - records all information till the error occurs and then records the error information. Marks an exception event.

Running Selenium server

1. [Standalone](#)
2. [Hub and Node](#)
3. [Fully Distributed](#)
4. [Docker](#)

Visualizing Traces

All spans, events and their respective attributes are part of a trace. Tracing works while running the server in all of the above-mentioned modes.

By default, tracing is enabled in the Selenium server. Selenium server exports the traces via two exporters:

1. Console - Logs all traces and their included spans at FINE level. By default, Selenium server prints logs at INFO level and above. The **log-level** flag can be used to pass a logging level of choice while running the Selenium Grid jar/s.

```
java -jar selenium-server-4.0.0-<selenium-version>.jar standalone --log-level FI
```

2. Jaeger UI - OpenTelemetry provides the APIs and SDKs to instrument traces in the code. Whereas Jaeger is a tracing backend, that aids in collecting the tracing telemetry data and providing querying, filtering and visualizing features for the data.

Detailed instructions of visualizing traces using Jaeger UI can be obtained by running the command :

```
java -jar selenium-server-4.0.0-<selenium-version>.jar info tracing
```

[A very good example and scripts to run the server and send traces to Jaeger](#)

Leveraging event logs

Tracing has to be enabled for event logging as well, even if one does not wish to export traces to visualize them.

By default, tracing is enabled. No additional parameters need to be passed to see logs on the console. All events within a span are logged at FINE level. Error events are logged at WARN level.

All event logs have the following fields :

Field	Field value	Description
Event time	eventId	Timestamp of the event record in epoch nanoseconds.
Trace Id	traceId	Each trace is uniquely identified by a trace id.
Span Id	spanId	Each span within a trace is uniquely identified by a span id.
Span Kind	spanKind	Span kind is a property of span indicating the type of span. It helps in understanding the nature of the unit of work done by the Span.
Event name	eventName	This maps to the log message.
Event attributes	eventAttributes	This forms the crux of the event logs, based on the operation executed, it has JSON formatted key-value pairs. This also includes a handler class attribute, to show the logger class.

Sample log

```
FINE [LoggingOptions$1.lambda$export$1] - {
  "traceId": "fc8aef1d44b3cc8bc09eb8e581c4a8eb",
  "spanId": "b7d3b9865d3ddd45",
  "spanKind": "INTERNAL",
  "eventTime": 1597819675128886121,
  "eventName": "Session request execution complete",
  "attributes": {
    "http.status_code": 200,
    "http.handler_class": "org.openqa.selenium.grid.router.HandleSession",
    "http.url": "\u002fsession\u002fdd35257f104bb43fdfb06242953f4c85",
    "http.method": "DELETE",
    "session.id": "dd35257f104bb43fdfb06242953f4c85"
  }
}
```

In addition to the above fields, based on [OpenTelemetry specification](#) error logs consist of :

Field	Field value	Description
Exception type	exception.type	The class name of the exception.
Exception message	exception.message	Reason for the exception.
Exception stacktrace	exception.stacktrace	Prints the call stack at the point of time when the exception was thrown. Helps in understanding the origin of the exception.

Sample error log

```
WARN [LoggingOptions$1.lambda$export$1] - {
    "traceId": "7efa5ea57e02f89cdf8de586fe09f564",
    "spanId": "914df6bc9a1f6e2b",
    "spanKind": "INTERNAL",
    "eventTime": 1597820253450580272,
    "eventName": "exception",
    "attributes": {
        "exception.type": "org.openqa.selenium.ScriptTimeoutException",
        "exception.message": "Unable to execute request: java.sql.SQLSyntaxErrorException: Table 'testdb.t1' doesn't exist",
        "exception.stacktrace": "org.openqa.selenium.ScriptTimeoutException: java.sql.SQLSyntaxErrorException: Table 'testdb.t1' doesn't exist\n        at org.openqa.selenium.grid.distributor.remote.RemoteDistributor$1.call(RemoteDistributor.java:110)\n        at org.openqa.selenium.grid.distributor.remote.RemoteDistributor$1.call(RemoteDistributor.java:107)\n        at java.util.concurrent.FutureTask.run(FutureTask.java:266)\n        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)\n        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)\n        at java.lang.Thread.run(Thread.java:748)\nCaused by: java.sql.SQLSyntaxErrorException: Table 'testdb.t1' doesn't exist\n        at com.mysql.cj.jdbc.exceptions.SQLError.createSQLException(SQLError.java:110)\n        at com.mysql.cj.jdbc.exceptions.SQLExceptionsMapping.translateException(SQLExceptionsMapping.java:112)\n        at com.mysql.cj.jdbc.ClientPreparedStatement.executeInternal(ClientPreparedStatement.java:903)\n        at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1019)\n        at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdateInternal(ClientPreparedStatement.java:1004)\n        at com.mysql.cj.jdbc.ClientPreparedStatement.executeUpdate(ClientPreparedStatement.java:982)\n        at org.openqa.selenium.grid.distributor.remote.RemoteDistributor$1.call(RemoteDistributor.java:107)\n        ... 4 more\n    }
}
```

Note: Logs are pretty printed above for readability. Pretty printing for logs is turned off in Selenium server.

The steps above should set you up for seeing traces and logs.

References

1. [Understanding Tracing](#)
2. [OpenTelemetry Tracing API Specification](#)
3. [Selenium Wiki](#)
4. [Structured logs vs events](#)
5. [Jaeger framework](#)

3.6.2 - GraphQL query support

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. It gives users the power to ask for exactly what they need and nothing more.

Enums

Enums represent possible sets of values for a field.

For example, the `Node` object has a field called `status`. The state is an enum (specifically, of type `Status`) because it may be `UP`, `DRAINING` or `UNAVAILABLE`.

Scalars

Scalars are primitive values: `Int`, `Float`, `String`, `Boolean`, or `ID`.

When calling the GraphQL API, you must specify nested subfield until you return only scalars.

Structure of the Schema

The structure of grid schema is as follows:

```
{
  session(id: "<session-id>") : {
    id,
    capabilities,
    startTime,
    uri,
    nodeId,
    nodeUri,
    sessionDurationMillis
    slot : {
      id,
      stereotype,
      lastStarted
    }
  }
  grid: {
    uri,
    totalSlots,
    nodeCount,
    maxSession,
    sessionCount,
    version,
    sessionQueueSize
  }
  sessionsInfo: {
    sessionQueueRequests,
    sessions: [
      {
        id,
        capabilities,
        startTime,
        uri,
        nodeId,
        nodeUri,
        sessionDurationMillis
        slot : {

```

```

        id,
        stereotype,
        lastStarted
    }
]
}
nodesInfo: {
    nodes : [
        {
            id,
            uri,
            status,
            maxSession,
            slotCount,
            sessions: [
                {
                    id,
                    capabilities,
                    startTime,
                    uri,
                    nodeId,
                    nodeUri,
                    sessionDurationMillis
                    slot : {
                        id,
                        stereotype,
                        lastStarted
                    }
                }
            ],
            sessionCount,
            stereotypes,
            version,
            osInfo: {
                arch,
                name,
                version
            }
        }
    ]
}
}

```

Querying GraphQL

The best way to query GraphQL is by using `curl` requests. The query is interpreted as JSON. Ensure double quotes are properly escaped to avoid unexpected errors. GraphQL allows you to fetch only the data that you want, nothing more nothing less.

Some of the example GraphQL queries are given below. You can build your own queries as you like.

Querying the number of `maxSession` and `sessionCount` in the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{ grid { maxSession, sessionCount } }"}'
```

Generally on local machine the <LINK_TO_GRAPHQL_ENDPOINT> would be <http://localhost:4444/graphql>

Querying all details for session, node and the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":"{ grid { uri,
```

Query for getting the current session count in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":"{ grid { sess
```

Query for getting the max session count in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":"{ grid { maxS
```

Query for getting all session details for all nodes in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":"{ sessionsIn
```

Query to get slot information for all sessions in each Node in the Grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query":"{ sessionsIn
```

Query to get session information for a given session:

```
curl -X POST -H "Content-Type: application/json" --data '{"query":"{ session (id
```

Querying the capabilities of each node in the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{ nodesInfo
```

Querying the status of each node in the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{ nodesInfo
```

Querying the URI of each node and the grid :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{ nodesInfo
```

Query for getting the current requests in the New Session Queue:

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{ sessionsInfo
```

Query for getting the New Session Queue size :

```
curl -X POST -H "Content-Type: application/json" --data '{"query": "{ grid { sess
```

3.6.3 - Grid endpoints

Grid

Grid Status

Grid status provides the current state of the Grid. It consists of details about every registered Node. For every Node, the status includes information regarding Node availability, sessions, and slots.

```
CURL GET 'http://localhost:4444/status'
```

In the Standalone mode, the Grid URL is the Standalone server address.

In the Hub-Node mode, the Grid URL is the Hub server address.

In the fully distributed mode, the Grid URL is the Router server address.

Default URL for all the above modes is http://localhost:4444.

Distributor

Remove Node

To remove the Node from the Grid, use the cURL command enlisted below. It does not stop any ongoing session running on that Node. The Node continues running as it is unless explicitly killed. The Distributor is no longer aware of the Node and hence any matching new session request will not be forwarded to that Node.

In the Standalone mode, the Distributor URL is the Standalone server address.

In the Hub-Node mode, the Distributor URL is the Hub server address.

```
CURL --request DELETE 'http://localhost:4444/se/grid/distributor/node/<node-id>'
```

In the fully distributed mode, the URL is the Distributor server address.

```
CURL --request DELETE 'http://localhost:5553/se/grid/distributor/node/<node-id>'
```

If no registration secret has been configured while setting up the Grid, then use

```
CURL --request DELETE 'http://<Distributor-URL>/se/grid/distributor/node/<node-i
```

Drain Node

Node drain command is for graceful node shutdown. Draining a Node stops the Node after all the ongoing sessions are complete. However, it does not accept any new session requests.

In the Standalone mode, the Distributor URL is the Standalone server address.

In the Hub-Node mode, the Distributor URL is the Hub server address.

```
curl --request POST 'http://localhost:4444/se/grid/distributor/node/<node-id>/dr
```

In the fully distributed mode, the URL is the Distributor server address.

```
curl --request POST 'http://localhost:5553/se/grid/distributor/node/<node-id>/dr
```

If no registration secret has been configured while setting up the Grid, then use

```
curl --request POST 'http://<Distributor-URL>/se/grid/distributor/node/<node-id>/dr
```

Node

The endpoints in this section are applicable for Hub-Node mode and fully distributed Grid mode where the Node runs independently. The default Node URL is `http://localhost:5555` in case of one Node. In case of multiple Nodes, use [Grid status](#) to get all Node details and locate the Node address.

Status

The Node status is essentially a health-check for the Node. Distributor pings the node status at regular intervals and updates the Grid Model accordingly. The status includes information regarding availability, sessions, and slots.

```
curl --request GET 'http://localhost:5555/status'
```

Drain

Distributor passes the [drain](#) command to the appropriate node identified by the node-id. To drain the Node directly, use the curl command enlisted below. Both endpoints are valid and produce the same result. Drain finishes the ongoing sessions before stopping the Node.

```
curl --request POST 'http://localhost:5555/se/grid/node/drain' --header 'X-REGIS
```

If no registration secret has been configured while setting up the Grid, then use

```
curl --request POST 'http://<node-URL>/se/grid/node/drain' --header 'X-REGISTRAT
```

Check session owner

To check if a session belongs to a Node, use the curl command enlisted below.

```
curl --request GET 'http://localhost:5555/se/grid/node/owner/<session-id>' --hea
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request GET 'http://<node-URL>/se/grid/node/owner/<session-id>' --header
```

It will return true if the session belongs to the Node else it will return false.

Delete session

Deleting the session terminates the WebDriver session, quits the driver and removes it from the active sessions map. Any request using the removed session-id or reusing the driver instance will throw an error.

```
cURL --request DELETE 'http://localhost:5555/se/grid/node/session/<session-id>'
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request DELETE 'http://<node-URL>/se/grid/node/session/<session-id>' --he
```

New Session Queue

Clear New Session Queue

New Session Request Queue holds the new session requests. To clear the queue, use the cURL command enlisted below. Clearing the queue rejects all the requests in the queue. For each such request, the server returns an error response to the respective client. The result of the clear command is the total number of deleted requests.

In the Standalone mode, the Queue URL is the Standalone server address.

In the Hub-Node mode, the Queue URL is the Hub server address.

```
cURL --request DELETE 'http://localhost:4444/se/grid/newsessionqueue/queue' --he
```

In the fully distributed mode, the Queue URL is New Session Queue server address.

```
cURL --request DELETE 'http://localhost:5559/se/grid/newsessionqueue/queue' --he
```

If no registration secret has been configured while setting up the Grid, then use

```
cURL --request DELETE 'http://<URL>/se/grid/newsessionqueue/queue' --header 'X-R
```

Get New Session Queue Requests

New Session Request Queue holds the new session requests. To get the current requests in the queue, use the cURL command enlisted below. The response returns the total number of requests in the queue and the request payloads.

In the Standalone mode, the Queue URL is the Standalone server address.

In the Hub-Node mode, the Queue URL is the Hub server address.

```
curl --request GET 'http://localhost:4444/se/grid/newsessionqueue/queue'
```

In the fully distributed mode, the Queue URL is New Session Queue server address.

```
curl --request GET 'http://localhost:5559/se/grid/newsessionqueue/queue'
```

4 - IE Driver Server

The Internet Explorer Driver is a standalone server that implements the WebDriver specification.

This documentation previously located [on the wiki](#)

The `InternetExplorerDriver` is a standalone server which implements WebDriver's wire protocol. This driver has been tested with IE 11, and on Windows 10. It might work with older versions of IE and Windows, but this is not supported.

The driver supports running 32-bit and 64-bit versions of the browser. The choice of how to determine which "bit-ness" to use in launching the browser depends on which version of the `IEDriverServer.exe` is launched. If the 32-bit version of `IEDriverServer.exe` is launched, the 32-bit version of IE will be launched. Similarly, if the 64-bit version of `IEDriverServer.exe` is launched, the 64-bit version of IE will be launched.

Installing

You do not need to run an installer before using the `InternetExplorerDriver`, though some configuration is required. The standalone server executable must be downloaded from the [Downloads](#) page and placed in your [PATH](#).

Pros

- Runs in a real browser and supports JavaScript

Cons

- Obviously the `InternetExplorerDriver` will only work on Windows!
- Comparatively slow (though still pretty snappy :)

Command-Line Switches

As a standalone executable, the behavior of the IE driver can be modified through various command-line arguments. To set the value of these command-line arguments, you should consult the documentation for the language binding you are using. The command line switches supported are described in the table below. All `-<switch>`, `--<switch>` and `/<switch>` are supported.

Switch	Meaning
<code>-port= <portNumber></code>	Specifies the port on which the HTTP server of the IE driver will listen for commands from language bindings. Defaults to 5555.
<code>-host= <hostAdapterIPAddress></code>	Specifies the IP address of the host adapter on which the HTTP server of the IE driver will listen for commands from language bindings. Defaults to 127.0.0.1.
<code>-log-level= <logLevel></code>	Specifies the level at which logging messages are output. Valid values are FATAL, ERROR, WARN, INFO, DEBUG, and TRACE. Defaults to FATAL.
<code>-log-file= <logFile></code>	Specifies the full path and file name of the log file. Defaults to <code>stdout</code> .

Switch	Meaning
-extract-path= <path>	Specifies the full path to the directory used to extract supporting files used by the server. Defaults to the TEMP directory if not specified.
-silent	Suppresses diagnostic output when the server is started.

Important System Properties

The following system properties (read using `System.getProperty()` and set using `System.setProperty()` in Java code or the “`-DpropertyName=value`” command line flag) are used by the InternetExplorerDriver :

Property	What it means
<code>webdriver.ie.driver</code>	The location of the IE driver binary.
<code>webdriver.ie.driver.host</code>	Specifies the IP address of the host adapter on which the IE driver will listen.
<code>webdriver.ie.driver.loglevel</code>	Specifies the level at which logging messages are output. Valid values are FATAL, ERROR, WARN, INFO, DEBUG, and TRACE. Defaults to FATAL.
<code>webdriver.ie.driver.logfile</code>	Specifies the full path and file name of the log file.
<code>webdriver.ie.driver.silent</code>	Suppresses diagnostic output when the IE driver is started.
<code>webdriver.ie.driver.extractpath</code>	Specifies the full path to the directory used to extract supporting files used by the server. Defaults to the TEMP directory if not specified.

Required Configuration

- The `IEDriverServer` executable must be [downloaded](#) and placed in your [PATH](#).
- On IE 7 or higher on Windows Vista, Windows 7, or Windows 10, you must set the Protected Mode settings for each zone to be the same value. The value can be on or off, as long as it is the same for every zone. To set the Protected Mode settings, choose “Internet Options...” from the Tools menu, and click on the Security tab. For each zone, there will be a check box at the bottom of the tab labeled “Enable Protected Mode”.
- Additionally, “Enhanced Protected Mode” must be disabled for IE 10 and higher. This option is found in the Advanced tab of the Internet Options dialog.
- The browser zoom level must be set to 100% so that the native mouse events can be set to the correct coordinates.
- For Windows 10, you also need to set “Change the size of text, apps, and other items” to 100% in display settings.
- For IE 11 *only*, you will need to set a registry entry on the target computer so that the driver can maintain a connection to the instance of Internet Explorer it creates. For 32-bit Windows installations, the key you must examine in the registry editor is
`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\Main\FeatureControl\FEATURE_BFCACHE`. For 64-bit Windows installations, the key is
`HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Internet Explorer\Main\FeatureControl\FEATURE_BFCACHE`. Please note that the `FEATURE_BFCACHE` subkey may or may not be present, and should be created if it is not present. **Important:** Inside this key, create a DWORD value named `iexplore.exe` with the value of 0.

Native Events and Internet Explorer

As the `InternetExplorerDriver` is Windows-only, it attempts to use so-called “native”, or OS-level events to perform mouse and keyboard operations in the browser. This is in contrast to using simulated JavaScript events for the same operations. The advantage of using native events is that it does not rely on the JavaScript sandbox, and it ensures proper JavaScript event propagation within the browser. However, there are currently some issues with mouse events when the IE browser window does not have focus, and when attempting to hover over elements.

Browser Focus

The challenge is that IE itself appears to not fully respect the Windows messages we send the IE browser window (`WM_MOUSEDOWN` and `WM_MOUSEUP`) if the window doesn’t have the focus. Specifically, the element being clicked on will receive a focus window around it, but the click will not be processed by the element. Arguably, we shouldn’t be sending messages at all; rather, we should be using the `SendInput()` API, but that API explicitly requires the window to have the focus. We have two conflicting goals with the WebDriver project.

First, we strive to emulate the user as closely as possible. This means using native events rather than simulating the events using JavaScript.

Second, we want to not require focus of the browser window being automated. This means that just forcing the browser window to the foreground is suboptimal.

An additional consideration is the possibility of multiple IE instances running under multiple WebDriver instances, which means any such “bring the window to the foreground” solution will have to be wrapped in some sort of synchronizing construct (mutex?) within the IE driver’s C++ code. Even so, this code will still be subject to race conditions, if, for example, the user brings another window to the foreground between the driver bringing IE to the foreground and executing the native event.

The discussion around the requirements of the driver and how to prioritize these two conflicting goals is ongoing. The current prevailing wisdom is to prioritize the former over the latter, and document that your machine will be unavailable for other tasks when using the IE driver. However, that decision is far from finalized, and the code to implement it is likely to be rather complicated.

Hovering Over Elements

When you attempt to hover over elements, and your physical mouse cursor is within the boundaries of the IE browser window, the hover will not work. More specifically, the hover will appear to work for a fraction of a second, and then the element will revert back to its previous state. The prevailing theory why this occurs is that IE is doing hit-testing of some sort during its event loop, which causes it to respond to the physical mouse position when the physical cursor is within the window bounds. The WebDriver development team has been unable to discover a workaround for this behavior of IE.

Clicking `<option>` Elements or Submitting Forms and `alert()`

There are two places where the IE driver does not interact with elements using native events. This is in clicking `<option>` elements within a `<select>` element. Under normal circumstances, the IE driver calculates where to click based on the position and size of the element, typically as returned by the JavaScript `getBoundingClientRect()` method. However, for `<option>` elements, `getBoundingClientRect()` returns a rectangle with zero position and zero size. The IE driver handles this one scenario by using the `click()` Automation Atom, which essentially sets the `.selected` property of the element and simulates the `onChange` event in JavaScript. However, this means that if the `onChange` event of the `<select>` element contains JavaScript code that calls `alert()`, `confirm()` or `prompt()`, calling WebElement’s `click()` method will hang until the modal dialog is manually dismissed. There is no known workaround for this behavior using only WebDriver code.

Similarly, there are some scenarios when submitting an HTML form via WebElement’s `submit()` method may have the same effect. This can happen if the driver calls the JavaScript `submit()` function on the form, and there is an `onSubmit` event handler that calls the JavaScript `alert()`, `confirm()`, or `prompt()` functions.

This restriction is filed as issue 3508 (on Google Code).

Multiple instances of InternetExplorerDriver

With the creation of the `IEDriverServer.exe`, it should be possible to create and use multiple simultaneous instances of the `InternetExplorerDriver`. However, this functionality is largely untested, and there may be issues with cookies, window focus, and the like. If you attempt to use multiple instances of the IE driver, and run into such issues, consider using the `RemoteWebDriver` and virtual machines.

There are 2 solutions for problem with cookies (and another session items) shared between multiple instances of InternetExplorer.

The first is to start your InternetExplorer in private mode. After that InternetExplorer will be started with clean session data and will not save changed session data at quitting. To do so you need to pass 2 specific capabilities to driver: `ie.forceCreateProcessApi` with `true` value and `ie.browserCommandLineSwitches` with `-private` value. Be note that it will work only for InternetExplorer 8 and newer, and Windows Registry

`HKLM_CURRENT_USER\Software\Microsoft\Internet Explorer\Main` path should contain key `TabProcGrowth` with `0` value.

The second is to clean session during InternetExplorer starting. For this you need to pass specific `ie.ensureCleanSession` capability with `true` value to driver. This clears the cache for all running instances of InternetExplorer, including those started manually.

Running IEDriverServer.exe Remotely

The HTTP server started by the `IEDriverServer.exe` sets an access control list to only accept connections from the local machine, and disallows incoming connections from remote machines. At present, this cannot be changed without modifying the source code to the `IEDriverServer.exe`. To run the Internet Explorer driver on a remote machine, use the Java standalone remote server in connection with your language binding's equivalent of `RemoteWebDriver`.

Running IEDriverServer.exe Under a Windows Service

Attempting to use IEDriverServer.exe as part of a Windows Service application is expressly unsupported. Service processes, and processes spawned by them, have much different requirements than those executing in a regular user context. `IEDriverServer.exe` is explicitly untested in that environment, and includes Windows API calls that are documented to be prohibited to be used in service processes. While it may be possible to get the IE driver to work while running under a service process, users encountering problems in that environment will need to seek out their own solutions.

4.1 - Internet Explorer Driver Internals

More detailed information on the IE Driver.

Client Code Into the Driver

We use the W3C WebDriver protocol to communicate with a local instance of an HTTP server. This greatly simplifies the implementation of the language-specific code, and minimizes the number of entry points into the C++ DLL that must be called using a native-code interop technology such as [JNA](#), [ctypes](#), [pinvoke](#) or [DL](#).

Memory Management

The IE driver utilizes the Active Template Library (ATL) to take advantage of its implementation of smart pointers to COM objects. This makes reference counting and cleanup of COM objects much easier.

Why Do We Require Protected Mode Settings Changes?

IE 7 on Windows Vista introduced the concept of Protected Mode, which allows for some measure of protection to the underlying Windows OS when browsing. The problem is that when you manipulate an instance of IE via COM, and you navigate to a page that would cause a transition into or out of Protected Mode, IE requires that another browser session be created. This will orphan the COM object of the previous session, not allowing you to control it any longer.

In IE 7, this will usually manifest itself as a new top-level browser window; in IE 8, a new IExplore.exe process will be created, but it will usually (not always!) seamlessly attach it to the existing IE top-level frame window. Any browser automation framework that drives IE externally (as opposed to using a WebBrowser control) will run into these problems.

In order to work around that problem, we dictate that to work with IE, all zones must have the same Protected Mode setting. As long as it's on for all zones, or off for all zones, we can prevent the transitions to different Protected Mode zones that would invalidate our browser object. It also allows users to continue to run with UAC turned on, and to run securely in the browser if they set Protected Mode "on" for all zones.

In earlier releases of the IE driver, if the user's Protected Mode settings were not correctly set, we would launch IE, and the process would simply hang until the HTTP request timed out. This was suboptimal, as it gave no indication what needed to be set. Erring on the side of caution, we do not modify the user's Protected Mode settings. Current versions, however check that the Protected Mode settings are properly set, and will return an error response if they are not.

Keyboard and Mouse Input

Key files: [interactions.cpp](#)

There are two ways that we could simulate keyboard and mouse input. The first way, which is used in parts of webdriver, is to synthesize events on the DOM. This has a number of drawbacks, since each browser (and version of a browser) has its own unique quirks; to model each of these is a demanding task, and impossible to get completely right (for example, it's hard to tell what `window.selection` should be and this is a read-only property on some browsers) The alternative approach is to synthesize keyboard and mouse input at the OS level, ideally without stealing focus from the user (who tends to be doing other things on their computer as long-running webdriver tests run)

The code for doing this is in [interactions.cpp](#). The key thing to note here is that we use PostMessages to push window events on to the message queue of the IE instance. Typing, in particular, is interesting: we only send the "keydown" and "keyup" messages. The "keypress" event is created if necessary by IE's internal event processing. Because the key press event is not always generated (for example, not

every character is printable, and if the default event bubbling is cancelled, listeners don't see the key press event) we send a "probe" event in after the key down. Once we see that this has been processed, we know that the key press event is on the stack of events to be processed, and that it is safe to send the key up event. If this was not done, it is possible for events to fire in the wrong order, which is definitely sub-optimal.

Working On the InternetExplorerDriver

Currently, there are tests that will run for the InternetExplorerDriver in all languages (Java, C#, Python, and Ruby), so you should be able to test your changes to the native code no matter what language you're comfortable working in from the client side. For working on the C++ code, you'll need Visual Studio 2010 Professional or higher. Unfortunately, the C++ code of the driver uses ATL to ease the pain of working with COM objects, and ATL is not supplied with Visual C++ 2010 Express Edition. If you're using Eclipse, the process for making and testing modifications is:

1. Edit the C++ code in VS.
2. Build the code to ensure that it compiles
3. Do a complete rebuild when you are ready to run a test. This will cause the created DLL to be copied to the right place to allow its use in Eclipse
4. Load Eclipse (or some other IDE, such as Idea)
5. Edit the `SingleTestSuite` so that it is `usingDriver(IE)`
6. Create a JUnit run configuration that uses the "webdriver-internet-explorer" project. If you don't do this, the test won't work at all, and there will be a somewhat cryptic error message on the console.

Once the basic setup is done, you can start working on the code pretty quickly. You can attach to the process you execute your code from using Visual Studio (from the Debug menu, select Attach to Process...).

5 - Selenium IDE

The Selenium IDE is a browser extension that records and plays back a user's actions.

Selenium's Integrated Development Environment ([Selenium IDE](#)) is an easy-to-use browser extension that records a user's actions in the browser using existing Selenium commands, with parameters defined by the context of each element. It provides an excellent way to learn Selenium syntax. It's available for Google Chrome, Mozilla Firefox, and Microsoft Edge.

For more information, visit the complete [Selenium IDE Documentation](#)

6 - Test Practices

Some guidelines and recommendations on testing from the Selenium project.

A note on “Best Practices”: We’ve intentionally avoided the phrase “Best Practices” in this documentation. No one approach works for all situations. We prefer the idea of “Guidelines and Recommendations”. We encourage you to read through these and thoughtfully decide what approaches will work for you in your particular environment.

Functional testing is difficult to get right for many reasons. As if application state, complexity, and dependencies do not make testing difficult enough, dealing with browsers (especially with cross-browser incompatibilities) makes writing good tests a challenge.

Selenium provides tools to make functional user interaction easier, but does not help you write well-architected test suites. In this chapter we offer advice, guidelines, and recommendations on how to approach functional web page automation.

This chapter records software design patterns popular amongst many of the users of Selenium that have proven successful over the years.

6.1 - Design patterns and development strategies

(previously located: <https://github.com/SeleniumHQ/selenium/wiki/Bot-Style-Tests>)

Overview

Over time, projects tend to accumulate large numbers of tests. As the total number of tests increases, it becomes harder to make changes to the codebase — a single “simple” change may cause numerous tests to fail, even though the application still works properly. Sometimes these problems are unavoidable, but when they do occur you want to be up and running again as quickly as possible. The following design patterns and strategies have been used before with WebDriver to help make tests easier to write and maintain. They may help you too.

[DomainDrivenDesign](#): Express your tests in the language of the end-user of the app. [PageObjects](#): A simple abstraction of the UI of your web app. LoadableComponent: Modeling PageObjects as components. BotStyleTests: Using a command-based approach to automating tests, rather than the object-based approach that PageObjects encourage

Loadable Component

What Is It?

The LoadableComponent is a base class that aims to make writing PageObjects less painful. It does this by providing a standard way of ensuring that pages are loaded and providing hooks to make debugging the failure of a page to load easier. You can use it to help reduce the amount of boilerplate code in your tests, which in turn makes maintaining your tests less tiresome.

There is currently an implementation in Java that ships as part of Selenium 2, but the approach used is simple enough to be implemented in any language.

Simple Usage

As an example of a UI that we'd like to model, take a look at the [new issue](#) page. From the point of view of a test author, this offers the service of being able to file a new issue. A basic Page Object would look like:

```

package com.example.webdriver;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class EditIssue {

    private final WebDriver driver;

    public EditIssue(WebDriver driver) {
        this.driver = driver;
    }

    public void setSummary(String summary) {
        WebElement field = driver.findElement(By.name("summary"));
        clearAndType(field, summary);
    }

    public void enterDescription(String description) {
        WebElement field = driver.findElement(By.name("comment"));
        clearAndType(field, description);
    }

    public IssueList submit() {
        driver.findElement(By.id("submit")).click();
        return new IssueList(driver);
    }

    private void clearAndType(WebElement field, String text) {
        field.clear();
        field.sendKeys(text);
    }
}

```

In order to turn this into a LoadableComponent, all we need to do is to set that as the base type:

```

public class EditIssue extends LoadableComponent<EditIssue> {
    // rest of class ignored for now
}

```

This signature looks a little unusual, but all it means is that this class represents a LoadableComponent that loads the EditIssue page.

By extending this base class, we need to implement two new methods:

```

@Override
protected void load() {
    driver.get("https://github.com/SeleniumHQ/selenium/issues/new");
}

@Override
protected void isLoaded() throws Error {
    String url = driver.getCurrentUrl();
    assertTrue("Not on the issue entry page: " + url, url.endsWith("/new"));
}

```

The `load` method is used to navigate to the page, whilst the `isLoaded` method is used to determine whether we are on the right page. Although the method looks like it should return a boolean, instead it performs a series of assertions using JUnit's Assert class. There can be as few or as many assertions as you like. By using these assertions it's possible to give users of the class clear information that can be used to debug tests.

With a little rework, our PageObject looks like:

```

package com.example.webdriver;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

import static junit.framework.Assert.assertTrue;

public class EditIssue extends LoadableComponent<EditIssue> {

    private final WebDriver driver;

    // By default the PageFactory will locate elements with the same name or id
    // as the field. Since the summary element has a name attribute of "summary"
    // we don't need any additional annotations.
    private WebElement summary;

    // Same with the submit element, which has the ID "submit"
    private WebElement submit;

    // But we'd prefer a different name in our code than "comment", so we use the
    // FindBy annotation to tell the PageFactory how to locate the element.
    @FindBy(name = "comment") private WebElement description;

    public EditIssue(WebDriver driver) {
        this.driver = driver;

        // This call sets the WebElement fields.
        PageFactory.initElements(driver, this);
    }

    @Override
    protected void load() {
        driver.get("https://github.com/SeleniumHQ/selenium/issues/new");
    }

    @Override
    protected void isLoaded() throws Error {
        String url = driver.getCurrentUrl();
        assertTrue("Not on the issue entry page: " + url, url.endsWith("/new"));
    }

    public void setSummary(String issueSummary) {
        clearAndType(summary, issueSummary);
    }

    public void enterDescription(String issueDescription) {
        clearAndType(description, issueDescription);
    }

    public IssueList submit() {
        submit.click();
        return new IssueList(driver);
    }

    private void clearAndType(WebElement field, String text) {
        field.clear();
        field.sendKeys(text);
    }
}

```

That doesn't seem to have bought us much, right? One thing it has done is encapsulate the information about how to navigate to the page into the page itself, meaning that this information's not scattered through the code base. It also means that we can do this in our tests:

```
EditIssue page = new EditIssue(driver).get();
```

This call will cause the driver to navigate to the page if that's necessary.

Nested Components

LoadableComponents start to become more useful when they are used in conjunction with other LoadableComponents. Using our example, we could view the “edit issue” page as a component within a project’s website (after all, we access it via a tab on that site). You also need to be logged in to file an issue. We could model this as a tree of nested components:

```
+ ProjectPage
+---+ SecuredPage
    +---+ EditIssue
```

What would this look like in code? For a start, each logical component would have its own class. The “load” method in each of them would “get” the parent. The end result, in addition to the EditIssue class above is:

ProjectPage.java:

```
package com.example.webdriver;

import org.openqa.selenium.WebDriver;

import static org.junit.Assert.assertTrue;

public class ProjectPage extends LoadableComponent<ProjectPage> {

    private final WebDriver driver;
    private final String projectName;

    public ProjectPage(WebDriver driver, String projectName) {
        this.driver = driver;
        this.projectName = projectName;
    }

    @Override
    protected void load() {
        driver.get("http://" + projectName + ".googlecode.com/");
    }

    @Override
    protected void isLoaded() throws Error {
        String url = driver.getCurrentUrl();

        assertTrue(url.contains(projectName));
    }
}
```

and SecuredPage.java:

```

package com.example.webdriver;

import org.openqa.selenium.By;
import org.openqa.selenium.NoSuchElementException;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

import static org.junit.Assert.fail;

public class SecuredPage extends LoadableComponent<SecuredPage> {

    private final WebDriver driver;
    private final LoadableComponent<?> parent;
    private final String username;
    private final String password;

    public SecuredPage(WebDriver driver, LoadableComponent<?> parent, String username, String
        this.driver = driver;
        this.parent = parent;
        this.username = username;
        this.password = password;
    }

    @Override
    protected void load() {
        parent.get();

        String originalUrl = driver.getCurrentUrl();

        // Sign in
        driver.get("https://www.google.com/accounts/ServiceLogin?service=code");
        driver.findElement(By.name("Email")).sendKeys(username);
        WebElement passwordField = driver.findElement(By.name("Passwd"));
        passwordField.sendKeys(password);
        passwordField.submit();

        // Now return to the original URL
        driver.get(originalUrl);
    }

    @Override
    protected void isLoaded() throws Error {
        // If you're signed in, you have the option of picking a different login.
        // Let's check for the presence of that.

        try {
            WebElement div = driver.findElement(By.id("multilogin-dropdown"));
        } catch (NoSuchElementException e) {
            fail("Cannot locate user name link");
        }
    }
}

```

The “load” method in EditIssue now looks like:

```

@Override
protected void load() {
    securedPage.get();

    driver.get("https://github.com/SeleniumHQ/selenium/issues/new");
}

```

This shows that the components are all “nested” within each other. A call to `get()` in EditIssue will cause all its dependencies to load too. The example usage:

```

public class FooTest {
    private EditIssue editIssue;

    @Before
    public void prepareComponents() {
        WebDriver driver = new FirefoxDriver();

        ProjectPage project = new ProjectPage(driver, "selenium");
        SecuredPage securedPage = new SecuredPage(driver, project, "example", "top secret");
        editIssue = new EditIssue(driver, securedPage);
    }

    @Test
    public void demonstrateNestedLoadableComponents() {
        editIssue.get();

        editIssue.setSummary("Summary");
        editIssue.enterDescription("This is an example");
    }
}

```

If you're using a library such as [Guiceberry](#) in your tests, the preamble of setting up the PageObjects can be omitted leading to nice, clear, readable tests.

Bot Pattern

(previously located: <https://github.com/SeleniumHQ/selenium/wiki/Bot-Style-Tests>)

Although PageObjects are a useful way of reducing duplication in your tests, it's not always a pattern that teams feel comfortable following. An alternative approach is to follow a more "command-like" style of testing.

A "bot" is an action-oriented abstraction over the raw Selenium APIs. This means that if you find that commands aren't doing the Right Thing for your app, it's easy to change them. As an example:

```

public class ActionBot {
    private final WebDriver driver;

    public ActionBot(WebDriver driver) {
        this.driver = driver;
    }

    public void click(By locator) {
        driver.findElement(locator).click();
    }

    public void submit(By locator) {
        driver.findElement(locator).submit();
    }

    /**
     * Type something into an input field. WebDriver doesn't normally clear these
     * before typing, so this method does that first. It also sends a return key
     * to move the focus out of the element.
     */
    public void type(By locator, String text) {
        WebElement element = driver.findElement(locator);
        element.clear();
        element.sendKeys(text + "\n");
    }
}

```

Once these abstractions have been built and duplication in your tests identified, it's possible to layer PageObjects on top of bots.

6.2 - Overview of Test Automation

First, start by asking yourself whether or not you really need to use a browser. Odds are that, at some point, if you are working on a complex web application, you will need to open a browser and actually test it.

Functional end-user tests such as Selenium tests are expensive to run, however. Furthermore, they typically require substantial infrastructure to be in place to be run effectively. It is a good rule to always ask yourself if what you want to test can be done using more lightweight test approaches such as unit tests or with a lower-level approach.

Once you have made the determination that you are in the web browser testing business, and you have your Selenium environment ready to begin writing tests, you will generally perform some combination of three steps:

- Set up the data
- Perform a discrete set of actions
- Evaluate the results

You will want to keep these steps as short as possible; one or two operations should be enough most of the time. Browser automation has the reputation of being “flaky”, but in reality, that is because users frequently demand too much of it. In later chapters, we will return to techniques you can use to mitigate apparent intermittent problems in tests, in particular on how to [overcome race conditions](#) between the browser and WebDriver.

By keeping your tests short and using the web browser only when you have absolutely no alternative, you can have many tests with minimal flake.

A distinct advantage of Selenium tests is their inherent ability to test all components of the application, from backend to frontend, from a user’s perspective. So in other words, whilst functional tests may be expensive to run, they also encompass large business-critical portions at one time.

Testing requirements

As mentioned before, Selenium tests can be expensive to run. To what extent depends on the browser you are running the tests against, but historically browsers’ behaviour has varied so much that it has often been a stated goal to cross-test against multiple browsers.

Selenium allows you to run the same instructions against multiple browsers on multiple operating systems, but the enumeration of all the possible browsers, their different versions, and the many operating systems they run on will quickly become a non-trivial undertaking.

Let’s start with an example

Larry has written a web site which allows users to order their custom unicorns.

The general workflow (what we will call the “happy path”) is something like this:

- Create an account
- Configure the unicorn
- Add it to the shopping cart
- Check out and pay
- Give feedback about the unicorn

It would be tempting to write one grand Selenium script to perform all these operations—many will try.

Resist the temptation! Doing so will result in a test that a) takes a long time, b) will be subject to some common issues around page rendering timing issues, and c) is such that if it fails, it will not give you a concise, “glanceable” method for diagnosing what went wrong.

The preferred strategy for testing this scenario would be to break it down to a series of independent, speedy tests, each of which has one “reason” to exist.

Let us pretend you want to test the second step: Configuring your unicorn. It will perform the following actions:

- Create an account
- Configure a unicorn

Note that we are skipping the rest of these steps, we will test the rest of the workflow in other small, discrete test cases after we are done with this one.

To start, you need to create an account. Here you have some choices to make:

- Do you want to use an existing account?
- Do you want to create a new account?
- Are there any special properties of such a user that need to be taken into account before configuration begins?

Regardless of how you answer this question, the solution is to make it part of the “set up the data” portion of the test. If Larry has exposed an API that enables you (or anyone) to create and update user accounts, be sure to use that to answer this question. If possible, you want to launch the browser only after you have a user “in hand”, whose credentials you can just log in with.

If each test for each workflow begins with the creation of a user account, many seconds will be added to the execution of each test. Calling an API and talking to a database are quick, “headless” operations that don’t require the expensive process of opening a browser, navigating to the right pages, clicking and waiting for the forms to be submitted, etc.

Ideally, you can address this set-up phase in one line of code, which will execute before any browser is launched:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

```
// Create a user who has read-only permissions--they can configure a unicorn,
// but they do not have payment information set up, nor do they have
// administrative privileges. At the time the user is created, its email
// address and password are randomly generated--you don't even need to
// know them.
User user = UserFactory.createCommonUser(); //This method is defined elsewhere.

// Log in as this user.
// Logging in on this site takes you to your personal "My Account" page, so the
// AccountPage object is returned by the loginAs method, allowing you to then
// perform actions from the AccountPage.
AccountPage accountPage = loginAs(user.getEmail(), user.getPassword());
```

As you can imagine, the `UserFactory` can be extended to provide methods such as `createAdminUser()`, and `createUserWithPayment()`. The point is, these two lines of code do not distract you from the ultimate purpose of this test: configuring a unicorn.

The intricacies of the [Page Object model](#) will be discussed in later chapters, but we will introduce the concept here:

Your tests should be composed of actions, performed from the user’s point of view, within the context of pages in the site. These pages are stored as objects, which will contain specific information about how the web page is composed and how actions are performed– very little of which should concern you as a tester.

What kind of unicorn do you want? You might want pink, but not necessarily. Purple has been quite popular lately. Does she need sunglasses? Star tattoos? These choices, while difficult, are your primary concern as a tester– you need to ensure that your order fulfillment center sends out the right unicorn to the right person, and that starts with these choices.

Notice that nowhere in that paragraph do we talk about buttons, fields, drop-downs, radio buttons, or web forms. **Neither should your tests!** You want to write your code like the user trying to solve their problem. Here is one way of doing this (continuing from the previous example):

```
// The Unicorn is a top-level Object--it has attributes, which are set here.  
// This only stores the values; it does not fill out any web forms or interact  
// with the browser in any way.  
Unicorn sparkles = new Unicorn("Sparkles", UnicornColors.PURPLE, UnicornAccessors.  
  
// Since we are already "on" the account page, we have to use it to get to the  
// actual place where you configure unicorns. Calling the "Add Unicorn" method  
// takes us there.  
AddUnicornPage addUnicornPage = accountPage.addUnicorn();  
  
// Now that we're on the AddUnicornPage, we will pass the "sparkles" object to  
// its createUnicorn() method. This method will take Sparkles' attributes,  
// fill out the form, and click submit.  
UnicornConfirmationPage unicornConfirmationPage = addUnicornPage.createUnicorn(s
```

Now that you have configured your unicorn, you need to move on to step 3: making sure it actually worked.

```
// The exists() method from UnicornConfirmationPage will take the Sparkles  
// object--a specification of the attributes you want to see, and compare  
// them with the fields on the page.  
Assert.assertTrue("Sparkles should have been created, with all attributes intact",
```

Note that the tester still has not done anything but talk about unicorns in this code– no buttons, no locators, no browser controls. This method of *modelling* the application allows you to keep these test-level commands in place and unchanging, even if Larry decides next week that he no longer likes Ruby-on-Rails and decides to re-implement the entire site in the latest Haskell bindings with a Fortran front-end.

Your page objects will require some small maintenance in order to conform to the site redesign, but these tests will remain the same. Taking this basic design, you will want to keep going through your workflows with the fewest browser-facing steps possible. Your next workflow will involve adding a unicorn to the shopping cart. You will probably want many iterations of this test in order to make sure the cart is keeping its state properly: Is there more than one unicorn in the cart before you start? How many can fit in the shopping cart? If you create more than one with the same name and/or features, will it break? Will it only keep the existing one or will it add another?

Each time you move through the workflow, you want to try to avoid having to create an account, login as the user, and configure the unicorn. Ideally, you will be able to create an account and pre-configure a unicorn via the API or database. Then all you have to do is log in as the user, locate Sparkles, and add her to the cart.

To automate or not to automate?

Is automation always advantageous? When should one decide to automate test cases?

It is not always advantageous to automate test cases. There are times when manual testing may be more appropriate. For instance, if the application's user interface will change considerably in the near future, then any automation might need to be rewritten anyway. Also, sometimes there simply is not

enough time to build test automation. For the short term, manual testing may be more effective. If an application has a very tight deadline, there is currently no test automation available, and it's imperative that the testing gets done within that time frame, then manual testing is the best solution.

6.3 - Types of Testing

Acceptance testing

This type of testing is done to determine if a feature or system meets the customer expectations and requirements. This type of testing generally involves the customer's cooperation or feedback, being a validation activity that answers the question:

Are we building the **right** product?

For web applications, the automation of this testing can be done directly with Selenium by simulating user expected behaviour. This simulation could be done by record/playback or through the different supported languages as explained in this documentation. Note: Acceptance testing is a subtype of **functional testing**, which some people might also refer to.

Functional testing

This type of testing is done to determine if a feature or system functions properly without issues. It checks the system at different levels to ensure that all scenarios are covered and that the system does *what it's supposed to do*. It's a verification activity that answers the question:

Are we building the product **right?**

This generally includes: the tests work without errors (404, exceptions...), in a usable way (correct redirections), in an accessible way and matching its specifications (see **acceptance testing** above).

For web applications, the automation of this testing can be done directly with Selenium by simulating expected returns.

This simulation could be done by record/playback or through the different supported languages as explained in this documentation.

Performance testing

As its name indicates, performance tests are done to measure how well an application is performing.

There are two main sub-types for performance testing:

Load testing

Load testing is done to verify how well the application works under different defined loads (usually a particular number of users connected at once).

Stress testing

Stress testing is done to verify how well the application works under stress (or above the maximum supported load).

Generally, performance tests are done by executing some Selenium written tests simulating different users hitting a particular function on the web app and retrieving some meaningful measurements.

This is generally done by other tools that retrieve the metrics. One such tool is **JMeter**.

For a web application, details to measure include throughput, latency, data loss, individual component loading times...

Note 1: All browsers have a performance tab in their developers' tools section (accessible by pressing F12)

Note 2: is a subtype of **non-functional testing** as this is generally measured per system and not per function/feature.

Regression testing

This testing is generally done after a change, fix or feature addition.

To ensure that the change has not broken any of the existing functionality, some already executed tests are executed again.

The set of re-executed tests can be full or partial and can include several different types, depending on the application and development team.

Test driven development (TDD)

Rather than a test type *per se*, TDD is an iterative development methodology in which tests drive the design of a feature.

Each cycle starts by creating a set of unit tests that the feature should eventually pass (they should fail their first time executed).

After this, development takes place to make the tests pass. The tests are executed again, starting another cycle and this process continues until all tests are passing.

This aims to speed up the development of an application based on the fact that defects are less costly the earlier they are found.

Behavior-driven development (BDD)

BDD is also an iterative development methodology based on the above TDD, in which the goal is to involve all the parties in the development of an application.

Each cycle starts by creating some specifications (which should fail). Then create the failing unit tests (which should also fail) and then do the development.

This cycle is repeated until all types of tests are passing.

In order to do so, a specification language is used. It should be understandable by all parties and simple, standard and explicit. Most tools use **Gherkin** as this language.

The goal is to be able to detect even more errors than TDD, by targeting potential acceptance errors too and make communication between parties smoother.

A set of tools are currently available to write the specifications and match them with code functions, such as **Cucumber** or **SpecFlow**.

A set of tools are built on top of Selenium to make this process even faster by directly transforming the BDD specifications into executable code. Some of these are **JBehave, Capybara and Robot Framework**.

6.4 - Encouraged behaviors

Some guidelines and recommendations on testing from the Selenium project.

A note on “Best Practices”: We’ve intentionally avoided the phrase “Best Practices” in this documentation. No one approach works for all situations. We prefer the idea of “Guidelines and Recommendations”. We encourage you to read through these and thoughtfully decide what approaches will work for you in your particular environment.

Functional testing is difficult to get right for many reasons. As if application state, complexity, and dependencies do not make testing difficult enough, dealing with browsers (especially with cross-browser incompatibilities) makes writing good tests a challenge.

Selenium provides tools to make functional user interaction easier, but does not help you write well-architected test suites. In this chapter we offer advice, guidelines, and recommendations on how to approach functional web page automation.

This chapter records software design patterns popular amongst many of the users of Selenium that have proven successful over the years.

6.4.1 - Page object models

Note: this page has merged contents from multiple sources, including the [Selenium wiki](#)

Overview

Within your web app's UI there are areas that your tests interact with. A Page Object simply models these as objects within the test code. This reduces the amount of duplicated code and means that if the UI changes, the fix need only be applied in one place.

Page Object is a Design Pattern which has become popular in test automation for enhancing test maintenance and reducing code duplication. A page object is an object-oriented class that serves as an interface to a page of your AUT. The tests then use the methods of this page object class whenever they need to interact with the UI of that page. The benefit is that if the UI changes for the page, the tests themselves don't need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place.

Advantages

- There is a clean separation between test code and page specific code such as locators (or their use if you're using a UI Map) and layout.
- There is a single repository for the services or operations offered by the page rather than having these services scattered throughout the tests.

In both cases this allows any modifications required due to UI changes to all be made in one place. Useful information on this technique can be found on numerous blogs as this 'test design pattern' is becoming widely used. We encourage the reader who wishes to know more to search the internet for blogs on this subject. Many have written on this design pattern and can provide useful tips beyond the scope of this user guide. To get you started, though, we'll illustrate page objects with a simple example.

Examples

First, consider an example, typical of test automation, that does not use a page object:

```
/**
 * Tests login feature
 */
public class Login {

    public void testLogin() {
        // fill login data on sign-in page
        driver.findElement(By.name("user_name")).sendKeys("userName");
        driver.findElement(By.name("password")).sendKeys("my supersecret password");
        driver.findElement(By.name("sign-in")).click();

        // verify h1 tag is "Hello userName" after login
        driver.findElement(By.tagName("h1")).isDisplayed();
        assertThat(driver.findElement(By.tagName("h1")).getText(), is("Hello userName"));
    }
}
```

There are two problems with this approach.

- There is no separation between the test method and the AUT's locators (IDs in this example); both are intertwined in a single method. If the AUT's UI changes its identifiers, layout, or how a login is input and processed, the test itself must change.
- The ID-locators would be spread in multiple tests, in all tests that had to use this login page.

Applying the page object techniques, this example could be rewritten like this in the following example of a page object for a Sign-in page.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

/**
 * Page Object encapsulates the Sign-in page.
 */
public class SignInPage {
    protected WebDriver driver;

    // <input name="user_name" type="text" value="">
    private By usernameBy = By.name("user_name");
    // <input name="password" type="password" value="">
    private By passwordBy = By.name("password");
    // <input name="sign_in" type="submit" value="SignIn">
    private By signinBy = By.name("sign_in");

    public SignInPage(WebDriver driver){
        this.driver = driver;
    }

    /**
     * Login as valid user
     *
     * @param userName
     * @param password
     * @return HomePage object
     */
    public HomePage loginValidUser(String userName, String password) {
        driver.findElement(usernameBy).sendKeys(userName);
        driver.findElement(passwordBy).sendKeys(password);
        driver.findElement(signinBy).click();
        return new HomePage(driver);
    }
}
```

and page object for a Home page could look like this.

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

/**
 * Page Object encapsulates the Home Page
 */
public class HomePage {
    protected WebDriver driver;

    // <h1>Hello userName</h1>
    private By messageBy = By.tagName("h1");

    public HomePage(WebDriver driver){
        this.driver = driver;
        if (!driver.getTitle().equals("Home Page of logged in user")) {
            throw new IllegalStateException("This is not Home Page of logged in user,"
                " current page is: " + driver.getCurrentUrl());
        }
    }
}
```

```

/**
 * Get message (h1 tag)
 *
 * @return String message text
 */
public String getMessageText() {
    return driver.findElement(messageBy).getText();
}

public HomePage manageProfile() {
    // Page encapsulation to manage profile functionality
    return new HomePage(driver);
}
/* More methods offering the services represented by Home Page
of Logged User. These methods in turn might return more Page Objects
for example click on Compose mail button could return ComposeMail class object
}

```

So now, the login test would use these two page objects as follows.

```

/**
 * Tests login feature
 */
public class TestLogin {

    @Test
    public void testLogin() {
        SignInPage signInPage = new SignInPage(driver);
        HomePage homePage = signInPage.loginValidUser("userName", "password");
        assertThat(homePage.getMessageText(), is("Hello userName"));
    }
}

```

There is a lot of flexibility in how the page objects may be designed, but there are a few basic rules for getting the desired maintainability of your test code.

Page objects themselves should never make verifications or assertions. This is part of your test and should always be within the test's code, never in a page object. The page object will contain the representation of the page, and the services the page provides via methods but no code related to what is being tested should be within the page object.

There is one, single, verification which can, and should, be within the page object and that is to verify that the page, and possibly critical elements on the page, were loaded correctly. This verification should be done while instantiating the page object. In the examples above, both the SignInPage and HomePage constructors check that the expected page is available and ready for requests from the test.

A page object does not necessarily need to represent all the parts of a page itself. The same principles used for page objects can be used to create “Page *Component* Objects” that represent discrete chunks of the page and can be included in page objects. These component objects can provide references to the elements inside those discrete chunks, and methods to leverage the functionality provided by them. You can even nest component objects inside other component objects for more complex pages. If a page in the AUT has multiple components, or common components used throughout the site (e.g. a navigation bar), then it may improve maintainability and reduce code duplication.

There are other design patterns that also may be used in testing. Some use a Page Factory for instantiating their page objects. Discussing all of these is beyond the scope of this user guide. Here, we merely want to introduce the concepts to make the reader aware of some of the things that can be

done. As was mentioned earlier, many have blogged on this topic and we encourage the reader to search for blogs on these topics.

Implementation Notes

PageObjects can be thought of as facing in two directions simultaneously. Facing towards the developer of a test, they represent the **services** offered by a particular page. Facing away from the developer, they should be the only thing that has a deep knowledge of the structure of the HTML of a page (or part of a page). It's simplest to think of the methods on a Page Object as offering the "services" that a page offers rather than exposing the details and mechanics of the page. As an example, think of the inbox of any web-based email system. Amongst the services that it offers are typically the ability to compose a new email, to choose to read a single email, and to list the subject lines of the emails in the inbox. How these are implemented shouldn't matter to the test.

Because we're encouraging the developer of a test to try and think about the services that they're interacting with rather than the implementation, PageObjects should seldom expose the underlying WebDriver instance. To facilitate this, methods on the PageObject should return other PageObjects. This means that we can effectively model the user's journey through our application. It also means that should the way that pages relate to one another change (like when the login page asks the user to change their password the first time they log into a service, when it previously didn't do that) simply changing the appropriate method's signature will cause the tests to fail to compile. Put another way, we can tell which tests would fail without needing to run them when we change the relationship between pages and reflect this in the PageObjects.

One consequence of this approach is that it may be necessary to model (for example) both a successful and unsuccessful login, or a click could have a different result depending on the state of the app. When this happens, it is common to have multiple methods on the PageObject:

```
public class LoginPage {
    public HomePage loginAs(String username, String password) {
        // ... clever magic happens here
    }

    public LoginPage loginAsExpectingError(String username, String password) {
        // ... failed login here, maybe because one or both of the username and password ar
    }

    public String getErrorMessage() {
        // So we can verify that the correct error is shown
    }
}
```

The code presented above shows an important point: the tests, not the PageObjects, should be responsible for making assertions about the state of a page. For example:

```
public void testMessagesAreReadOrUnread() {
    Inbox inbox = new Inbox(driver);
    inbox.assertMessageWithSubjectIsUnread("I like cheese");
    inbox.assertMessageWithSubjectIsNotUnread("I'm not fond of tofu");
}
```

could be re-written as:

```
public void testMessagesAreReadOrUnread() {
    Inbox inbox = new Inbox(driver);
    assertTrue(inbox.isMessageWithSubjectIsUnread("I like cheese"));
    assertFalse(inbox.isMessageWithSubjectIsUnread("I'm not fond of tofu"));
}
```

Of course, as with every guideline there are exceptions, and one that is commonly seen with PageObjects is to check that the WebDriver is on the correct page when we instantiate the PageObject. This is done in the example below.

Finally, a PageObject need not represent an entire page. It may represent a section that appears many times within a site or page, such as site navigation. The essential principle is that there is only one place in your test suite with knowledge of the structure of the HTML of a particular (part of a) page.

Summary

- The public methods represent the services that the page offers
- Try not to expose the internals of the page
- Generally don't make assertions
- Methods return other PageObjects
- Need not represent an entire page
- Different results for the same action are modelled as different methods

Example

```

public class LoginPage {
    private final WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;

        // Check that we're on the right page.
        if (!"Login".equals(driver.getTitle())) {
            // Alternatively, we could navigate to the login page, perhaps logging out first
            throw new IllegalStateException("This is not the login page");
        }
    }

    // The login page contains several HTML elements that will be represented as WebElements
    // The locators for these elements should only be defined once.
    By usernameLocator = By.id("username");
    By passwordLocator = By.id("passwd");
    By loginButtonLocator = By.id("login");

    // The login page allows the user to type their username into the username field
    public LoginPage typeUsername(String username) {
        // This is the only place that "knows" how to enter a username
        driver.findElement(usernameLocator).sendKeys(username);

        // Return the current page object as this action doesn't navigate to a page represen
        return this;
    }

    // The login page allows the user to type their password into the password field
    public LoginPage typePassword(String password) {
        // This is the only place that "knows" how to enter a password
        driver.findElement(passwordLocator).sendKeys(password);

        // Return the current page object as this action doesn't navigate to a page represen
        return this;
    }

    // The login page allows the user to submit the login form
    public HomePage submitLogin() {
        // This is the only place that submits the login form and expects the destination to
        // A seperate method should be created for the instance of clicking login whilst exp
        driver.findElement(loginButtonLocator).submit();

        // Return a new page object representing the destination. Should the login page ever
        // go somewhere else (for example, a legal disclaimer) then changing the method signa
        // for this method will mean that all tests that rely on this behaviour won't compil
        return new HomePage(driver);
    }

    // The login page allows the user to submit the login form knowing that an invalid userr
    public LoginPage submitLoginExpectingFailure() {
        // This is the only place that submits the login form and expects the destination to
        driver.findElement(loginButtonLocator).submit();

        // Return a new page object representing the destination. Should the user ever be na
        // expected to fail login, the script will fail when it attempts to instantiate the
        return new LoginPage(driver);
    }

    // Conceptually, the login page offers the user the service of being able to "log into"
    // the application using a user name and password.
    public HomePage loginAs(String username, String password) {
        // The PageObject methods that enter username, password & submit login have already
        typeUsername(username);
        typePassword(password);
        return submitLogin();
    }
}

```

Support in WebDriver

There is a PageFactory in the support package that provides support for this pattern, and helps to remove some boiler-plate code from your Page Objects at the same time.

6.4.2 - Domain specific language

A domain specific language (DSL) is a system which provides the user with an expressive means of solving a problem. It allows a user to interact with the system on their terms – not just programmer-speak.

Your users, in general, do not care how your site looks. They do not care about the decoration, animations, or graphics. They want to use your system to push their new employees through the process with minimal difficulty; they want to book travel to Alaska; they want to configure and buy unicorns at a discount. Your job as tester is to come as close as you can to “capturing” this mind-set. With that in mind, we set about “modeling” the application you are working on, such that the test scripts (the user’s only pre-release proxy) “speak” for, and represent the user.

The goal is to use *ubiquitous language*. Rather than referring to “load data into this table” or “click on the third column” it should be possible to use language such as “create a new account” or “order displayed results by name”

With Selenium, DSL is usually represented by methods, written to make the API simple and readable – they enable a report between the developers and the stakeholders (users, product owners, business intelligence specialists, etc.).

Benefits

- **Readable:** Business stakeholders can understand it.
- **Writable:** Easy to write, avoids unnecessary duplication.
- **Extensible:** Functionality can (reasonably) be added without breaking contracts and existing functionality.
- **Maintainable:** By leaving the implementation details out of test cases, you are well-insulated against changes to the AUT*.

Further Reading

(previously located: <https://github.com/SeleniumHQ/selenium/wiki/Domain-Driven-Design>)

There is a good book on Domain Driven Design by Eric Evans
<http://www.amazon.com/exec/obidos/ASIN/0321125215/domainlanguag-20>

And to whet your appetite there's a useful smaller book available online for download at
<http://www.infoq.com/minibooks/domain-driven-design-quickly>

Java

Here is an example of a reasonable DSL method in Java. For brevity's sake, it assumes the `driver` object is pre-defined and available to the method.

```
/**  
 * Takes a username and password, fills out the fields, and clicks "login".  
 * @return An instance of the AccountPage  
 */  
public AccountPage loginAsUser(String username, String password) {  
    WebElement loginField = driver.findElement(By.id("loginField"));  
    loginField.clear();  
    loginField.sendKeys(username);  
  
    // Fill out the password field. The locator we're using is "By.id", and we sh  
    // have it defined elsewhere in the class.  
    WebElement passwordField = driver.findElement(By.id("password"));  
    passwordField.clear();
```

```
passwordField.sendKeys(password);

// Click the login button, which happens to have the id "submit".
driver.findElement(By.id("submit")).click();

// Create and return a new instance of the AccountPage (via the built-in Selenium
// PageFactory).
return PageFactory.newInstance(AccountPage.class);
}
```

This method completely abstracts the concepts of input fields, buttons, clicking, and even pages from your test code. Using this approach, all a tester has to do is call this method. This gives you a maintenance advantage: if the login fields ever changed, you would only ever have to change this method - not your tests.

```
public void loginTest() {
    loginAsUser("cbrown", "cl0wn3");

    // Now that we're logged in, do some other stuff--since we used a DSL to support
    // our testers, it's as easy as choosing from available methods.
    do.something();
    do.somethingElse();
    Assert.assertTrue("Something should have been done!", something.wasDone());

    // Note that we still haven't referred to a button or web control anywhere in
    // the script...
}
```

It bears repeating: one of your primary goals should be writing an API that allows your tests to address **the problem at hand, and NOT the problem of the UI**. The UI is a secondary concern for your users – they do not care about the UI, they just want to get their job done. Your test scripts should read like a laundry list of things the user wants to DO, and the things they want to KNOW. The tests should not concern themselves with HOW the UI requires you to go about it.

***AUT**: Application under test

6.4.3 - Generating application state

Selenium should not be used to prepare a test case. All repetitive actions and preparations for a test case, should be done through other methods. For example, most web UIs have authentication (e.g. a login form). Eliminating logging in via web browser before every test will improve both the speed and stability of the test. A method should be created to gain access to the AUT* (e.g. using an API to login and set a cookie). Also, creating methods to pre-load data for testing should not be done using Selenium. As mentioned previously, existing APIs should be leveraged to create data for the AUT*.

***AUT**: Application under test

6.4.4 - Mock external services

Eliminating the dependencies on external services will greatly improve the speed and stability of your tests.

6.4.5 - Improved reporting

Selenium is not designed to report on the status of test cases run. Taking advantage of the built-in reporting capabilities of unit test frameworks is a good start. Most unit test frameworks have reports that can generate xUnit or HTML formatted reports. xUnit reports are popular for importing results to a Continuous Integration (CI) server like Jenkins, Travis, Bamboo, etc. Here are some links for more information regarding report outputs for several languages.

[NUnit 3 Console Runner](#)

[NUnit 3 Console Command Line](#)

[xUnit getting test results in TeamCity](#)

[xUnit getting test results in CruiseControl.NET](#)

[xUnit getting test results in Azure DevOps](#)

6.4.6 - Avoid sharing state

Although mentioned in several places it is worth mentioning again. Ensure tests are isolated from one another.

- Do not share test data. Imagine several tests that each query the database for valid orders before picking one to perform an action on. Should two tests pick up the same order you are likely to get unexpected behaviour.
- Clean up stale data in the application that might be picked up by another test e.g. invalid order records.
- Create a new WebDriver instance per test. This helps ensure test isolation and makes parallelization simpler.

6.4.7 - Tips on working with locators

When to use which locators and how best to manage them in your code.

Take a look at examples of the [supported locator strategies](#).

In general, if HTML IDs are available, unique, and consistently predictable, they are the preferred method for locating an element on a page. They tend to work very quickly, and forego much processing that comes with complicated DOM traversals.

If unique IDs are unavailable, a well-written CSS selector is the preferred method of locating an element. XPath works as well as CSS selectors, but the syntax is complicated and frequently difficult to debug. Though XPath selectors are very flexible, they are typically not performance tested by browser vendors and tend to be quite slow.

Selection strategies based on *linkText* and *partialLinkText* have drawbacks in that they only work on link elements. Additionally, they call down to [querySelectorAll](#) selectors internally in WebDriver.

Tag name can be a dangerous way to locate elements. There are frequently multiple elements of the same tag present on the page. This is mostly useful when calling the *findElements(By)* method which returns a collection of elements.

The recommendation is to keep your locators as compact and readable as possible. Asking WebDriver to traverse the DOM structure is an expensive operation, and the more you can narrow the scope of your search, the better.

6.4.8 - Test independency

Write each test as its own unit. Write the tests in a way that will not be reliant on other tests to complete:

Let us say there is a content management system with which you can create some custom content which then appears on your website as a module after publishing, and it may take some time to sync between the CMS and the application.

A wrong way of testing your module is that the content is created and published in one test, and then checking the module in another test. This is not feasible as the content may not be available immediately for the other test after publishing.

Instead, you can create a stub content which can be turned on and off within the affected test, and use that for validating the module. However, for content creation, you can still have a separate test.

6.4.9 - Consider using a fluent API

Martin Fowler coined the term “[Fluent API](#)”. Selenium already implements something like this in their `FluentWait` class, which is meant as an alternative to the standard `Wait` class. You could enable the Fluent API design pattern in your page object and then query the Google search page with a code snippet like this one:

```
driver.get( "http://www.google.com/webhp?hl=en&tab=ww" );
GoogleSearchPage gsp = new GoogleSearchPage();
gsp.withFluent().setSearchString().clickSearchButton();
```

The Google page object class with this fluent behavior might look like this:

```
public class GoogleSearchPage extends LoadableComponent<GoogleSearchPage> {
    private final WebDriver driver;
    private GSPFluentInterface gspfi;

    public class GSPFluentInterface {
        private GoogleSearchPage gsp;

        public GSPFluentInterface(GoogleSearchPage googleSearchPage) {
            gsp = googleSearchPage;
        }

        public GSPFluentInterface clickSearchButton() {
            gsp.searchButton.click();
            return this;
        }

        public GSPFluentInterface setSearchString( String sstr ) {
            clearAndType( gsp.searchField, sstr );
            return this;
        }
    }

    @FindBy(id = "gbqfq") private WebElement searchField;
    @FindBy(id = "gbqfb") private WebElement searchButton;
    public GoogleSearchPage(WebDriver driver) {
        gspfi = new GSPFluentInterface( this );
        this.get(); // If load() fails, calls isLoaded() until page is finished load
        PageFactory.initElements(driver, this); // Initialize WebElements on page
    }

    public GSPFluentInterface withFluent() {
        return gspfi;
    }

    public void clickSearchButton() {
        searchButton.click();
    }

    public void setSearchString( String sstr ) {
        clearAndType( searchField, sstr );
    }

    @Override
    protected void isLoaded() throws Error {
        Assert.assertTrue("Google search page is not yet loaded.", isSearchFieldVisi
    }
}
```

```
@Override  
protected void load() {  
    if ( isSFieldPresent ) {  
        Wait<WebDriver> wait = new WebDriverWait( driver, Duration.ofSeconds(3) );  
        wait.until( visibilityOfElementLocated( By.id("gbqfq") ) ).click();  
    }  
}  
}
```

6.4.10 - Fresh browser per test

Start each test from a clean known state. Ideally, spin up a new virtual machine for each test. If spinning up a new virtual machine is not practical, at least start a new WebDriver for each test. Most browser drivers like GeckoDriver and ChromeDriver will start with a clean known state with a new user profile, by default.

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

6.5 - Discouraged behaviors

Things to avoid when automating browsers with Selenium.

6.5.1 - Captchas

CAPTCHA, short for *Completely Automated Public Turing test to tell Computers and Humans Apart*, is explicitly designed to prevent automation, so do not try! There are two primary strategies to get around CAPTCHA checks:

- Disable CAPTCHAs in your test environment
- Add a hook to allow tests to bypass the CAPTCHA

6.5.2 - File downloads

Whilst it is possible to start a download by clicking a link with a browser under Selenium's control, the API does not expose download progress, making it less than ideal for testing downloaded files. This is because downloading files is not considered an important aspect of emulating user interaction with the web platform. Instead, find the link using Selenium (and any required cookies) and pass it to a HTTP request library like [libcurl](#).

The [HtmlUnit driver](#) can download attachments by accessing them as input streams by implementing the [AttachmentHandler](#) interface. The AttachmentHandler can be added to the [HtmlUnit](#) WebClient.

6.5.3 - HTTP response codes

For some browser configurations in Selenium RC, Selenium acted as a proxy between the browser and the site being automated. This meant that all browser traffic passed through Selenium could be captured or manipulated. The `captureNetworkTraffic()` method purported to capture all of the network traffic between the browser and the site being automated, including HTTP response codes.

Selenium WebDriver is a completely different approach to browser automation, preferring to act more like a user. This is represented in the way you write tests with WebDriver. In automated functional testing, checking the status code is not a particularly important detail of a test's failure; the steps that preceded it are more important.

The browser will always represent the HTTP status code, imagine for example a 404 or a 500 error page. A simple way to “fail fast” when you encounter one of these error pages is to check the page title or content of a reliable point (e.g. the `<h1>` tag) after every page load. If you are using the page object model, you can include this check in your class constructor or similar point where the page load is expected. Occasionally, the HTTP code may even be represented in the browser’s error page and you could use WebDriver to read this and improve your debugging output.

Checking the webpage itself is in line with WebDriver’s ideal practice of representing and asserting upon the user’s view of the website.

If you insist, an advanced solution to capturing HTTP status codes is to replicate the behaviour of Selenium RC by using a proxy. WebDriver API provides the ability to set a proxy for the browser, and there are a number of proxies that will programmatically allow you to manipulate the contents of requests sent to and received from the web server. Using a proxy lets you decide how you want to respond to redirection response codes. Additionally, not every browser makes the response codes available to WebDriver, so opting to use a proxy allows you to have a solution that works for every browser.

6.5.4 - Gmail, email and Facebook logins

For multiple reasons, logging into sites like Gmail and Facebook using WebDriver is not recommended. Aside from being against the usage terms for these sites (where you risk having the account shut down), it is slow and unreliable.

The ideal practice is to use the APIs that email providers offer, or in the case of Facebook the developer tools service which exposes an API for creating test accounts, friends and so forth. Although using an API might seem like a bit of extra hard work, you will be paid back in speed, reliability, and stability. The API is also unlikely to change, whereas webpages and HTML locators change often and require you to update your test framework.

Logging in to third party sites using WebDriver at any point of your test increases the risk of your test failing because it makes your test longer. A general rule of thumb is that longer tests are more fragile and unreliable.

WebDriver implementations that are [W3C conformant](#) also annotate the `navigator` object with a `WebDriver` property so that Denial of Service attacks can be mitigated.

6.5.5 - Test dependency

A common idea and misconception about automated testing is regarding a specific test order. Your tests should be able to run in **any** order, and not rely on other tests to complete in order to be successful.

6.5.6 - Performance testing

Performance testing using Selenium and WebDriver is generally not advised. Not because it is incapable, but because it is not optimised for the job and you are unlikely to get good results.

It may seem ideal to performance test in the context of the user but a suite of WebDriver tests are subjected to many points of external and internal fragility which are beyond your control; for example browser startup speed, speed of HTTP servers, response of third party servers that host JavaScript or CSS, and the instrumentation penalty of the WebDriver implementation itself. Variation at these points will cause variation in your results. It is difficult to separate the difference between the performance of your website and the performance of external resources, and it is also hard to tell what the performance penalty is for using WebDriver in the browser, especially if you are injecting scripts.

The other potential attraction is “saving time” — carrying out functional and performance tests at the same time. However, functional and performance tests have opposing objectives. To test functionality, a tester may need to be patient and wait for loading, but this will cloud the performance testing results and vice versa.

To improve the performance of your website, you will need to be able to analyse overall performance independent of environment differences, identify poor code practices, breakdown of performance of individual resources (i.e. CSS or JavaScript), in order to know what to improve. There are performance testing tools available that can do this job already, that provide reporting and analysis, and can even make improvement suggestions.

Example (open source) packages to use are: [JMeter](#)

6.5.7 - Link spidering

Using WebDriver to spider through links is not a recommended practice. Not because it cannot be done, but because WebDriver is definitely not the most ideal tool for this. WebDriver needs time to start up, and can take several seconds, up to a minute depending on how your test is written, just to get to the page and traverse through the DOM.

Instead of using WebDriver for this, you could save a ton of time by executing a [curl](#) command, or using a library such as BeautifulSoup since these methods do not rely on creating a browser and navigating to a page. You are saving tonnes of time by not using WebDriver for this task.

6.5.8 - Two Factor Authentication

Two Factor Authentication (2FA) is an authorization mechanism where a One Time Password (OTP) is generated using “Authenticator” mobile apps such as “Google Authenticator”, “Microsoft Authenticator” etc., or by SMS, e-mail to authenticate. Automating this seamlessly and consistently is a big challenge in Selenium. There are some ways to automate this process. But that will be another layer on top of our Selenium tests and not as secure. So, you should avoid automating 2FA.

There are few options to get around 2FA checks:

- Disable 2FA for certain Users in the test environment, so that you can use those user credentials in the automation.
- Disable 2FA in your test environment.
- Disable 2FA if you login from certain IPs. That way we can configure our test machine IPs to avoid this.

7 - Legacy

Documentation related to the legacy components of Selenium. Meant to be kept purely for historical reasons and not as an incentive to use deprecated components.

7.1 - Selenium RC (Selenium 1)

The original version of Selenium

Introduction

Selenium RC was the main Selenium project for a long time, before the WebDriver/Selenium merge brought up Selenium 2, a more powerful tool. It is worth to highlight that Selenium 1 is not supported anymore.

How Selenium RC Works

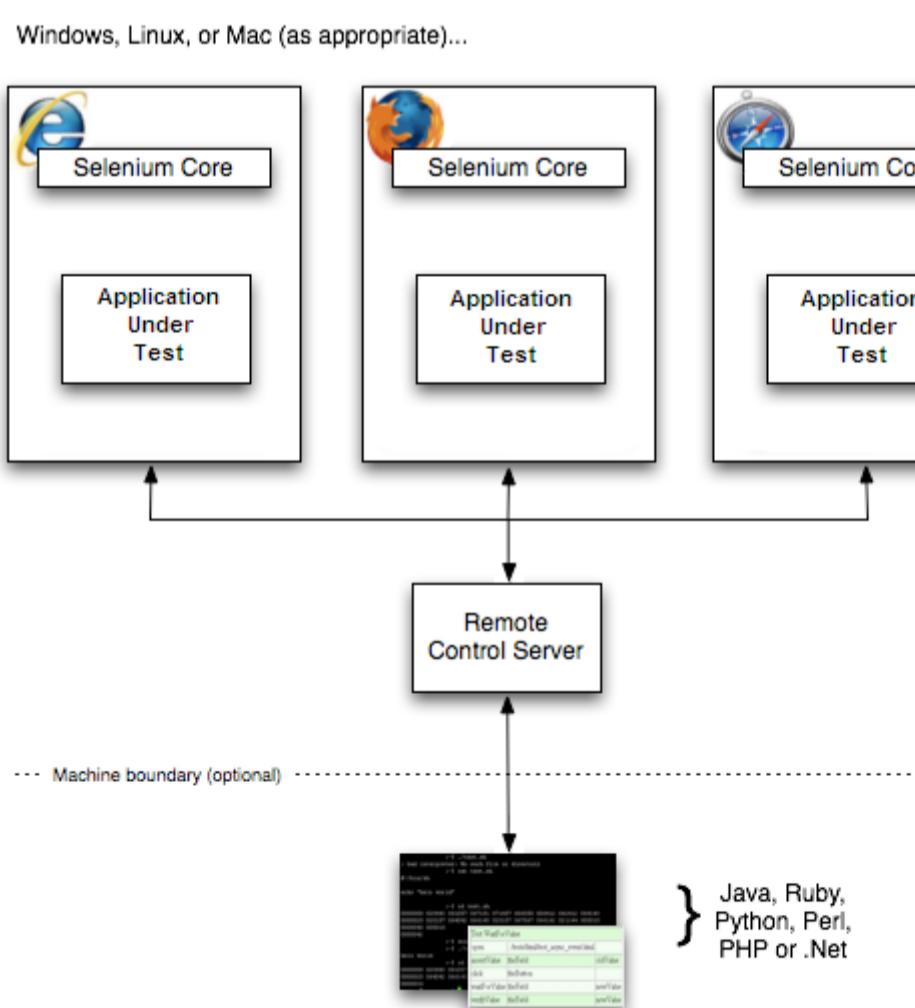
First, we will describe how the components of Selenium RC operate and the role each plays in running your test scripts.

RC Components

Selenium RC components are:

- The Selenium Server which launches and kills browsers, interprets and runs the Selenese commands passed from the test program, and acts as an *HTTP proxy*, intercepting and verifying HTTP messages passed between the browser and the AUT.
- Client libraries which provide the interface between each programming language and the Selenium RC Server.

Here is a simplified architecture diagram:



The diagram shows the client libraries communicate with the Server passing each Selenium command for execution. Then the server passes the Selenium command to the browser using Selenium-Core JavaScript commands. The browser, using its JavaScript interpreter, executes the Selenium command. This runs the Selenese action or verification you specified in your test script.

Selenium Server

Selenium Server receives Selenium commands from your test program, interprets them, and reports back to your program the results of running those tests.

The RC server bundles Selenium Core and automatically injects it into the browser. This occurs when your test program opens the browser (using a client library API function). Selenium-Core is a JavaScript program, actually a set of JavaScript functions which interprets and executes Selenese commands using the browser's built-in JavaScript interpreter.

The Server receives the Selenese commands from your test program using simple HTTP GET/POST requests. This means you can use any programming language that can send HTTP requests to automate Selenium tests on the browser.

Client Libraries

The client libraries provide the programming support that allows you to run Selenium commands from a program of your own design. There is a different client library for each supported language. A Selenium client library provides a programming interface (API), i.e., a set of functions, which run Selenium commands from your own program. Within each interface, there is a programming function that supports each Selenese command.

The client library takes a Selenese command and passes it to the Selenium Server for processing a specific action or test against the application under test (AUT). The client library also receives the result of that command and passes it back to your program. Your program can receive the result and store it into a program variable and report it as a success or failure, or possibly take corrective action if it was an unexpected error.

So to create a test program, you simply write a program that runs a set of Selenium commands using a client library API. And, optionally, if you already have a Selenese test script created in the Selenium-IDE, you can *generate the Selenium RC code*. The Selenium-IDE can translate (using its Export menu item) its Selenium commands into a client-driver's API function calls. See the Selenium-IDE chapter for specifics on exporting RC code from Selenium-IDE.

Installation

Installation is rather a misnomer for Selenium. Selenium has a set of libraries available in the programming language of your choice. You could download them from the [downloads page](#).

Once you've chosen a language to work with, you simply need to:

- Install the Selenium RC Server.
- Set up a programming project using a language specific client driver.

Installing Selenium Server

The Selenium RC server is simply a Java *jar* file (*selenium-server-standalone-.jar*), which doesn't require any special installation. Just downloading the zip file and extracting the server in the desired directory is sufficient.

Running Selenium Server

Before starting any tests you must start the server. Go to the directory where Selenium RC's server is located and run the following from a command-line console.

```
java -jar selenium-server-standalone-<version-number>.jar
```

This can be simplified by creating a batch or shell executable file (.bat on Windows and .sh on Linux) containing the command above. Then make a shortcut to that executable on your desktop and simply double-click the icon to start the server.

For the server to run you'll need Java installed and the PATH environment variable correctly configured to run it from the console. You can check that you have Java correctly installed by running the following on a console.

```
java -version
```

If you get a version number (which needs to be 1.5 or later), you're ready to start using Selenium RC.

Using the Java Client Driver

- Download Selenium java client driver zip from the SeleniumHQ [downloads page](#).
- Extract selenium-java-.jar file
- Open your desired Java IDE (Eclipse, NetBeans, IntelliJ, Netweaver, etc.)
- Create a java project.
- Add the selenium-java-.jar files to your project as references.
- Add to your project classpath the file selenium-java-.jar.
- From Selenium-IDE, export a script to a Java file and include it in your Java project, or write your Selenium test in Java using the selenium-java-client API. The API is presented later in this chapter. You can either use JUnit, or TestNG to run your test, or you can write your own simple main() program. These concepts are explained later in this section.
- Run Selenium server from the console.
- Execute your test from the Java IDE or from the command-line.

For details on Java test project configuration, see the Appendix sections Configuring Selenium RC With Eclipse and Configuring Selenium RC With IntelliJ.

Using the Python Client Driver

- Install Selenium via PIP, instructions linked at SeleniumHQ [downloads page](#)
- Either write your Selenium test in Python or export a script from Selenium-IDE to a python file.
- Run Selenium server from the console
- Execute your test from a console or your Python IDE

For details on Python client driver configuration, see the appendix Python Client Driver Configuration.

Using the .NET Client Driver

- Download Selenium RC from the SeleniumHQ [downloads page](#)
- Extract the folder
- Download and install [NUnit](#) (Note: You can use NUnit as your test engine. If you're not familiar yet with NUnit, you can also write a simple main() function to run your tests; however NUnit is very useful as a test engine.)
- Open your desired .Net IDE (Visual Studio, SharpDevelop, MonoDevelop)
- Create a class library (.dll)
- Add references to the following DLLs: nmock.dll, nunit.core.dll, nunit.framework.dll, ThoughtWorks.Selenium.Core.dll, ThoughtWorks.Selenium.IntegrationTests.dll and ThoughtWorks.Selenium.UnitTests.dll
- Write your Selenium test in a .Net language (C#, VB.Net), or export a script from Selenium-IDE to a C# file and copy this code into the class file you just created.
- Write your own simple main() program or you can include NUnit in your project for running your test. These concepts are explained later in this chapter.
- Run Selenium server from console
- Run your test either from the IDE, from the NUnit GUI or from the command line

For specific details on .NET client driver configuration with Visual Studio, see the appendix .NET client driver configuration.

Using the Ruby Client Driver

- If you do not already have RubyGems, install it from RubyForge.
- Run `gem install selenium-client`
- At the top of your test script, add `require "selenium/client"`
- Write your test script using any Ruby test harness (eg Test::Unit, Mini::Test or RSpec).
- Run Selenium RC server from the console.
- Execute your test in the same way you would run any other Ruby script.

For details on Ruby client driver configuration, see the [Selenium-Client documentation](#) .

From Selenese to a Program

The primary task for using Selenium RC is to convert your Selenese into a programming language. In this section, we provide several different language-specific examples.

Sample Test Script

Let's start with an example Selenese test script. Imagine recording the following test with Selenium-IDE.

```
open      /
type      q          selenium rc
clickAndWait      btnG
assertTextPresent  Results * for selenium rc
```

Note: This example would work with the Google search page <http://www.google.com>

Selenese as Programming Code

Here is the test script exported (via Selenium-IDE) to each of the supported programming languages. If you have at least basic knowledge of an object- oriented programming language, you will understand how Selenium runs Selenese commands by reading one of these examples. To see an example in a specific language, select one of these buttons.

CSharp

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
using Selenium;

namespace SeleniumTests
{
    [TestFixture]
    public class NewTest
    {
        private ISelenium selenium;
        private StringBuilder verificationErrors;

        [SetUp]

```

```

public void SetupTest()
{
    selenium = new DefaultSelenium("localhost", 4444, "*firefox"
    selenium.Start();
    verificationErrors = new StringBuilder();
}

[TearDown]
public void TeardownTest()
{
    try
    {
        selenium.Stop();
    }
    catch (Exception)
    {
        // Ignore errors if unable to close the browser
    }
    Assert.AreEqual("", verificationErrors.ToString());
}

[Test]
public void TheNewTest()
{
    selenium.Open("/");
    selenium.Type("q", "selenium rc");
    selenium.Click("btnG");
    selenium.WaitForPageToLoad("30000");
    Assert.AreEqual("selenium rc - Google Search", selenium.GetT
}
}

```

Java

```

/** Add JUnit framework to your classpath if not already there
 * for this example to work
 */
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class NewTest extends SeleneseTestCase {
    public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");
    }
    public void testNew() throws Exception {
        selenium.open("/");
        selenium.type("q", "selenium rc");
        selenium.click("btnG");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("Results * for selenium rc"));
    }
}

```

Php

```

<?php

require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class Example extends PHPUnit_Extensions_SeleniumTestCase
{
    function setUp()
    {
        $this->setBrowser("*firefox");
        $this->setBrowserUrl("http://www.google.com/");
    }

    function testMyTestCase()
    {
        $this->open("/");
        $this->type("q", "selenium rc");
        $this->click("btnG");
        $this->waitForPageToLoad("30000");
        $this->assertTrue($this->isTextPresent("Results * for selenium rc"));
    }
}

?>

```

Python

```

from selenium import selenium
import unittest, time, re

class NewTest(unittest.TestCase):
    def setUp(self):
        self.verificationErrors = []
        self.selenium = selenium("localhost", 4444, "*firefox",
                               "http://www.google.com/")
        self.selenium.start()

    def test_new(self):
        sel = self.selenium
        sel.open("/")
        sel.type("q", "selenium rc")
        sel.click("btnG")
        sel.wait_for_page_to_load("30000")
        self.failUnless(sel.is_text_present("Results * for selenium rc"))

    def tearDown(self):
        self.selenium.stop()
        self.assertEqual([], self.verificationErrors)

```

Ruby

```

require "selenium/client"
require "test/unit"

class NewTest < Test::Unit::TestCase
    def setup

```

```

@verification_errors = []
if $selenium
  @selenium = $selenium
else
  @selenium = Selenium::Client::Driver.new("localhost", 4444, "*firefox")
  @selenium.start
end
@selenium.set_context("test_new")
end

def teardown
  @selenium.stop unless $selenium
  assert_equal [], @verification_errors
end

def test_new
  @selenium.open "/"
  @selenium.type "q", "selenium rc"
  @selenium.click "btnG"
  @selenium.wait_for_page_to_load "30000"
  assert @selenium.is_text_present("Results * for selenium rc")
end
end

```

In the next section we'll explain how to build a test program using the generated code.

Programming Your Test

Now we'll illustrate how to program your own tests using examples in each of the supported programming languages. There are essentially two tasks:

- Generate your script into a programming language from Selenium-IDE, optionally modifying the result.
- Write a very simple main program that executes the generated code.

Optionally, you can adopt a test engine platform like JUnit or TestNG for Java, or NUnit for .NET if you are using one of those languages.

Here, we show language-specific examples. The language-specific APIs tend to differ from one to another, so you'll find a separate explanation for each.

- Java
- C#
- Python
- Ruby
- Perl, PHP

Java

For Java, people use either JUnit or TestNG as the test engine.

Some development environments like Eclipse have direct support for these via plug-ins. This makes it even easier. Teaching JUnit or TestNG is beyond the scope of this document however materials may be found online and there are publications available. If you are already a “java-shop” chances are your developers will already have some experience with one of these test frameworks.

You will probably want to rename the test class from “NewTest” to something of your own choosing. Also, you will need to change the browser-open parameters in the statement:

```
selenium = new DefaultSelenium("localhost", 4444, "*iehta", "http://www.goog
```

The Selenium-IDE generated code will look like this. This example has comments added manually for additional clarity.

```
package com.example.tests;
// We specify the package of our tests

import com.thoughtworks.selenium.*;
// This is the driver's import. You'll use this for instantiating a
// browser and making it do what you need.

import java.util.regex.Pattern;
// Selenium-IDE adds the Pattern module because it's sometimes used for
// regex validations. You can remove the module if it's not used in your
// script.

public class NewTest extends SeleneseTestCase {
// We create our Selenium test case

    public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");
        // We instantiate and start the browser
    }

    public void testNew() throws Exception {
        selenium.open("/");
        selenium.type("q", "selenium rc");
        selenium.click("btnG");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("Results * for selenium rc"));
        // These are the real test steps
    }
}
```

C#

The .NET Client Driver works with Microsoft.NET. It can be used with any .NET testing framework like NUnit or the Visual Studio 2005 Team System.

Selenium-IDE assumes you will use NUnit as your testing framework. You can see this in the generated code below. It includes the *using* statement for NUnit along with corresponding NUnit attributes identifying the role for each member function of the test class.

You will probably have to rename the test class from “NewTest” to something of your own choosing. Also, you will need to change the browser-open parameters in the statement:

```
selenium = new DefaultSelenium("localhost", 4444, "*iehta", "http://www.goog
```

The generated code will look similar to this.

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
```

```
using Selenium;

namespace SeleniumTests

{
    [TestFixture]

    public class NewTest

    {
        private ISelenium selenium;

        private StringBuilder verificationErrors;

        [SetUp]

        public void SetupTest()

        {
            selenium = new DefaultSelenium("localhost", 4444, "*iehta",
                "http://www.google.com/");

            selenium.Start();

            verificationErrors = new StringBuilder();
        }

        [TearDown]

        public void TeardownTest()
        {
            try
            {
                selenium.Stop();
            }

            catch (Exception)
            {
                // Ignore errors if unable to close the browser
            }

            Assert.AreEqual("", verificationErrors.ToString());
        }

        [Test]

        public void TheNewTest()
        {
            // Open Google search engine.
            selenium.Open("http://www.google.com/");

            // Assert Title of page.
            Assert.AreEqual("Google", selenium.GetTitle());

            // Provide search term as "Selenium OpenQA"
            selenium.Type("q", "Selenium OpenQA");

            // Read the keyed search term and assert it.
            Assert.AreEqual("Selenium OpenQA", selenium.GetValue("q"));

            // Click on Search button.
            selenium.Click("btnG");
        }
    }
}
```

```

// Wait for page to load.
selenium.WaitForPageToLoad("5000");

// Assert that "www.openqa.org" is available in search results.
Assert.IsTrue(selenium.IsTextPresent("www.openqa.org"));

// Assert that page title is - "Selenium OpenQA - Google Search"
Assert.AreEqual("Selenium OpenQA - Google Search",
    selenium.GetTitle());
}

}

}

```

You can allow NUnit to manage the execution of your tests. Or alternatively, you can write a simple `main()` program that instantiates the test object and runs each of the three methods, `SetupTest()`, `TheNewTest()`, and `TeardownTest()` in turn.

Python

Pyunit is the test framework to use for Python.

The basic test structure is:

```

from selenium import selenium
# This is the driver's import. You'll use this class for instantiating a
# browser and making it do what you need.

import unittest, time, re
# This are the basic imports added by Selenium-IDE by default.
# You can remove the modules if they are not used in your script.

class NewTest(unittest.TestCase):
# We create our unittest test case

    def setUp(self):
        self.verificationErrors = []
        # This is an empty array where we will store any verification errors
        # we find in our tests

        self.selenium = selenium("localhost", 4444, "*firefox",
                               "http://www.google.com/")
        self.selenium.start()
        # We instantiate and start the browser

    def test_new(self):
        # This is the test code. Here you should put the actions you need
        # the browser to do during your test.

        sel = self.selenium
        # We assign the browser to the variable "sel" (just to save us from
        # typing "self.selenium" each time we want to call the browser).

        sel.open("/")
        sel.type("q", "selenium rc")
        sel.click("btnG")
        sel.wait_for_page_to_load("30000")
        self.failUnless(sel.is_text_present("Results * for selenium rc"))
        # These are the real test steps

    def tearDown(self):

```

```
self.selenium.stop()
# we close the browser (I'd recommend you to comment this line while
# you are creating and debugging your tests)

self.assertEqual([], self.verifyErrors)
# And make the test fail if we found that any verification errors
# were found
```

Ruby

Old (pre 2.0) versions of Selenium-IDE generate Ruby code that requires the old Selenium gem. Therefore, it is advisable to update any Ruby scripts generated by the IDE as follows:

1. On line 1, change `require "selenium"` to `require "selenium/client"`
2. On line 11, change `Selenium::SeleniumDriver.new` to `Selenium::Client::Driver.new`

You probably also want to change the class name to something more informative than “Untitled,” and change the test method’s name to something other than “`test_untitled`.”

Here is a simple example created by modifying the Ruby code generated by Selenium IDE, as described above.

```
# load the Selenium-Client gem
require "selenium/client"

# Load Test::Unit, Ruby's default test framework.
# If you prefer RSpec, see the examples in the Selenium-Client
# documentation.
require "test/unit"

class Untitled < Test::Unit::TestCase

  # The setup method is called before each test.
  def setup

    # This array is used to capture errors and display them at the
    # end of the test run.
    @verification_errors = []

    # Create a new instance of the Selenium-Client driver.
    @selenium = Selenium::Client::Driver.new \
      :host => "localhost",
      :port => 4444,
      :browser => "*chrome",
      :url => "http://www.google.com/",
      :timeout_in_second => 60

    # Start the browser session
    @selenium.start

    # Print a message in the browser-side log and status bar
    # (optional).
    @selenium.set_context("test_untitled")

  end

  # The teardown method is called after each test.
  def teardown
```

```

# Stop the browser session.
@selenium.stop

# Print the array of error messages, if any.
assert_equal [], @verification_errors
end

# This is the main body of your test.
def testUntitled

  # Open the root of the site we specified when we created the
  # new driver instance, above.
  @selenium.open "/"

  # Type 'selenium rc' into the field named 'q'
  @selenium.type "q", "selenium rc"

  # Click the button named "btnG"
  @selenium.click "btnG"

  # Wait for the search results page to load.
  # Note that we don't need to set a timeout here, because that
  # was specified when we created the new driver instance, above.
  @selenium.wait_for_page_to_load

begin

  # Test whether the search results contain the expected text.
  # Notice that the star (*) is a wildcard that matches any
  # number of characters.
  assert @selenium.is_text_present("Results * for selenium rc")

  rescue Test::Unit::AssertionFailedError

    # If the assertion fails, push it onto the array of errors.
    @verification_errors << $!

  end
end
end

```

Perl, PHP

The members of the documentation team have not used Selenium RC with Perl or PHP. If you are using Selenium RC with either of these two languages please contact the Documentation Team (see the chapter on contributing). We would love to include some examples from you and your experiences, to support Perl and PHP users.

Learning the API

The Selenium RC API uses naming conventions that, assuming you understand Selenese, much of the interface will be self-explanatory. Here, however, we explain the most critical and possibly less obvious aspects.

Starting the Browser

CSharp

```
selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://www.  
selenium.Start();
```

Java

```
setUp("http://www.google.com/", "*firefox");
```

Perl

```
my $sel = Test::WWW::Selenium->new( host => "localhost",  
                                      port => 4444,  
                                      browser => "*firefox",  
                                      browser_url => "http://www.google.com/
```

PHP

```
$this->setBrowser("*firefox");  
$this->setBrowserUrl("http://www.google.com/");
```

Python

```
self.selenium = selenium("localhost", 4444, "*firefox",  
                        "http://www.google.com/")  
self.selenium.start()
```

Ruby

```
@selenium = Selenium::ClientDriver.new("localhost", 4444, "*firefox", "htt  
@selenium.start
```

Each of these examples opens the browser and represents that browser by assigning a “browser instance” to a program variable. This program variable is then used to call methods from the browser. These methods execute the Selenium commands, i.e. like *open* or *type* or the *verify* commands.

The parameters required when creating the browser instance are:

- **host** Specifies the IP address of the computer where the server is located. Usually, this is the same machine as where the client is running, so in this case *localhost* is passed. In some clients this is an optional parameter.
- **port** Specifies the TCP/IP socket where the server is listening waiting for the client to establish a connection. This also is optional in some client drivers.
- **browser** The browser in which you want to run the tests. This is a required parameter.
- **url** The base url of the application under test. This is required by all the client libs and is integral information for starting up the browser-proxy-AUT communication.

Note that some of the client libraries require the browser to be started explicitly by calling its `start()` method.

Running Commands

Once you have the browser initialized and assigned to a variable (generally named “selenium”) you can make it run Selenese commands by calling the respective methods from the browser variable. For example, to call the `type` method of the selenium object:

```
selenium.type("field-id","string to type")
```

In the background the browser will actually perform a `type` operation, essentially identical to a user typing input into the browser, by using the locator and the string you specified during the method call.

Reporting Results

Selenium RC does not have its own mechanism for reporting results. Rather, it allows you to build your reporting customized to your needs using features of your chosen programming language. That’s great, but what if you simply want something quick that’s already done for you? Often an existing library or test framework can meet your needs faster than developing your own test reporting code.

Test Framework Reporting Tools

Test frameworks are available for many programming languages. These, along with their primary function of providing a flexible test engine for executing your tests, include library code for reporting results. For example, Java has two commonly used test frameworks, JUnit and TestNG. .NET also has its own, NUnit.

We won’t teach the frameworks themselves here; that’s beyond the scope of this user guide. We will simply introduce the framework features that relate to Selenium along with some techniques you can apply. There are good books available on these test frameworks however along with information on the internet.

Test Report Libraries

Also available are third-party libraries specifically created for reporting test results in your chosen programming language. These often support a variety of formats such as HTML or PDF.

What’s The Best Approach?

Most people new to the testing frameworks will begin with the framework’s built-in reporting features. From there most will examine any available libraries as that’s less time consuming than developing your own. As you begin to use Selenium no doubt you will start putting in your own “print statements” for reporting progress. That may gradually lead to you developing your own reporting, possibly in parallel to using a library or test framework. Regardless, after the initial, but short, learning curve you will naturally develop what works best for your own situation.

Test Reporting Examples

To illustrate, we’ll direct you to some specific tools in some of the other languages supported by Selenium. The ones listed here are commonly used and have been used extensively (and therefore recommended) by the authors of this guide.

Test Reports in Java

- If Selenium Test cases are developed using JUnit then JUnit Report can be used to generate test reports.

- If Selenium Test cases are developed using TestNG then no external task is required to generate test reports. The TestNG framework generates an HTML report which list details of tests.
- ReportNG is a HTML reporting plug-in for the TestNG framework. It is intended as a replacement for the default TestNG HTML report. ReportNG provides a simple, colour-coded view of the test results.

Logging the Selenese Commands

- Logging Selenium can be used to generate a report of all the Selenese commands in your test along with the success or failure of each. Logging Selenium extends the Java client driver to add this Selenese logging ability.

Test Reports for Python

- When using Python Client Driver then HTMLTestRunner can be used to generate a Test Report.

Test Reports for Ruby

- If RSpec framework is used for writing Selenium Test Cases in Ruby then its HTML report can be used to generate a test report.

Adding Some Spice to Your Tests

Now we'll get to the whole reason for using Selenium RC, adding programming logic to your tests. It's the same as for any program. Program flow is controlled using condition statements and iteration. In addition you can report progress information using I/O. In this section we'll show some examples of how programming language constructs can be combined with Selenium to solve common testing problems.

You will find as you transition from the simple tests of the existence of page elements to tests of dynamic functionality involving multiple web-pages and varying data that you will require programming logic for verifying expected results. Basically, the Selenium-IDE does not support iteration and standard condition statements. You can do some conditions by embedding JavaScript in Selenese parameters, however iteration is impossible, and most conditions will be much easier in a programming language. In addition, you may need exception handling for error recovery. For these reasons and others, we have written this section to illustrate the use of common programming techniques to give you greater 'verification power' in your automated testing.

The examples in this section are written in C# and Java, although the code is simple and can be easily adapted to the other supported languages. If you have some basic knowledge of an object-oriented programming language you shouldn't have difficulty understanding this section.

Iteration

Iteration is one of the most common things people need to do in their tests. For example, you may want to execute a search multiple times. Or, perhaps for verifying your test results you need to process a "result set" returned from a database.

Using the same Google search example we used earlier, let's check the Selenium search results. This test could use the Selenese:

```
open      /
type      q          selenium rc
clickAndWait  btnG
assertTextPresent Results * for selenium rc
```

type	q	selenium ide
clickAndWait	btnG	
assertTextPresent	Results * for selenium ide	
type	q	selenium grid
clickAndWait	btnG	
assertTextPresent	Results * for selenium grid	

The code has been repeated to run the same steps 3 times. But multiple copies of the same code is not good program practice because it's more work to maintain. By using a programming language, we can iterate over the search results for a more flexible and maintainable solution.

In C#

```
// Collection of String values.
String[] arr = {"ide", "rc", "grid"};

// Execute loop for each String in array 'arr'.
foreach (String s in arr) {
    sel.open("/");
    sel.type("q", "selenium " +s);
    sel.click("btnG");
    sel.waitForPageToLoad("30000");
    assertTrue("Expected text: " +s+ " is missing on page."
        , sel.isTextPresent("Results * for selenium " + s));
}
```

Condition Statements

To illustrate using conditions in tests we'll start with an example. A common problem encountered while running Selenium tests occurs when an expected element is not available on page. For example, when running the following line:

```
selenium.type("q", "selenium " +s);
```

If element 'q' is not on the page then an exception is thrown:

```
com.thoughtworks.selenium.SeleniumException: ERROR: Element q not found
```

This can cause your test to abort. For some tests that's what you want. But often that is not desirable as your test script has many other subsequent tests to perform.

A better approach is to first validate whether the element is really present and then take alternatives when it is not. Let's look at this using Java.

```
// If element is available on page then perform type operation.
if(selenium.isElementPresent("q")) {
    selenium.type("q", "Selenium rc");
} else {
```

```
System.out.printf("Element: " +q+ " is not available on page.")  
}
```

The advantage of this approach is to continue with test execution even if some of elements are not available on page.

Executing JavaScript from Your Test

JavaScript comes very handy in exercising an application which is not directly supported by Selenium. The **getEval** method of Selenium API can be used to execute JavaScript from Selenium RC.

Consider an application having check boxes with no static identifiers. In this case one could evaluate JavaScript from Selenium RC to get ids of all check boxes and then exercise them.

```
public static String[] getAllCheckboxIds () {  
    String script = "var inputId = new Array();"; // Create array in java sc  
    script += "var cnt = 0;"; // Counter for check box ids.  
    script += "var inputFields = new Array();"; // Create array in java scr  
    script += "inputFields = window.document.getElementsByTagName('input');"  
    script += "for(var i=0; i<inputFields.length; i++) {"; // Loop through t  
    script += "if(inputFields[i].id !=null " +  
    "&& inputFields[i].id !='undefined' " +  
    "&& inputFields[i].getAttribute('type') == 'checkbox') {"; // If input f  
    script += "inputId[cnt]=inputFields[i].id ;" + // Save check box id to i  
    "cnt++;" + // increment the counter.  
    "}" + // end of if.  
    "}" + // end of for.  
    script += "inputId.toString();"; // Convert array in to string.  
    String[] checkboxIds = selenium.getEval(script).split(","); // Split the  
    return checkboxIds;  
}
```

To count number of images on a page:

```
selenium.getEval("window.document.images.length");
```

Remember to use window object in case of DOM expressions as by default selenium window is referred to, not the test window.

Server Options

When the server is launched, command line options can be used to change the default server behaviour.

Recall, the server is started by running the following.

```
$ java -jar selenium-server-standalone-<version-number>.jar
```

To see the list of options, run the server with the `-h` option.

```
$ java -jar selenium-server-standalone-<version-number>.jar -h
```

You'll see a list of all the options you can use with the server and a brief description of each. The provided descriptions will not always be enough, so we've provided explanations for some of the more important options.

Proxy Configuration

If your AUT is behind an HTTP proxy which requires authentication then you should configure http.proxyHost, http.proxyPort, http.proxyUser and http.proxyPassword using the following command.

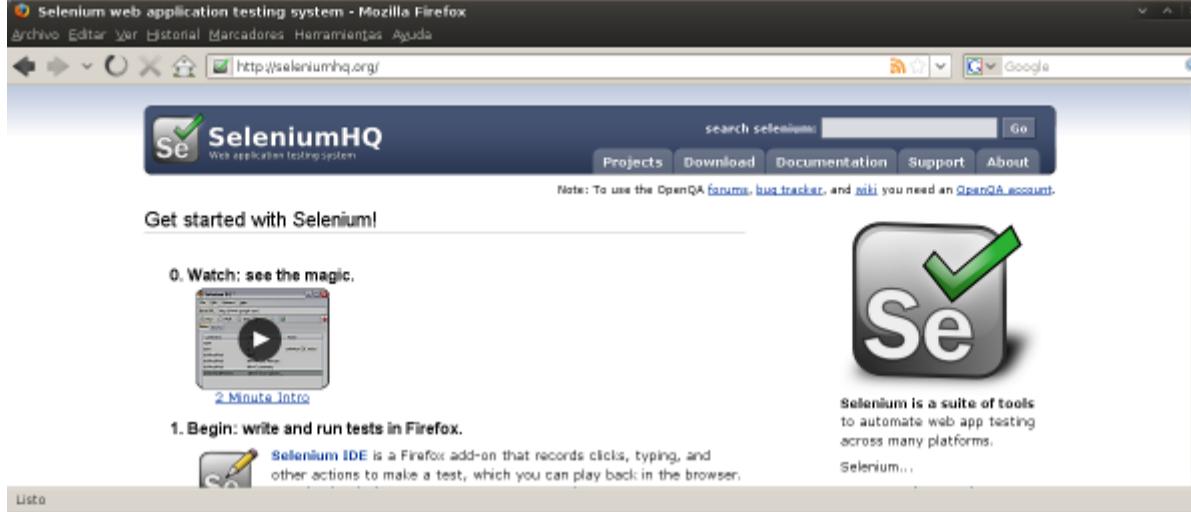
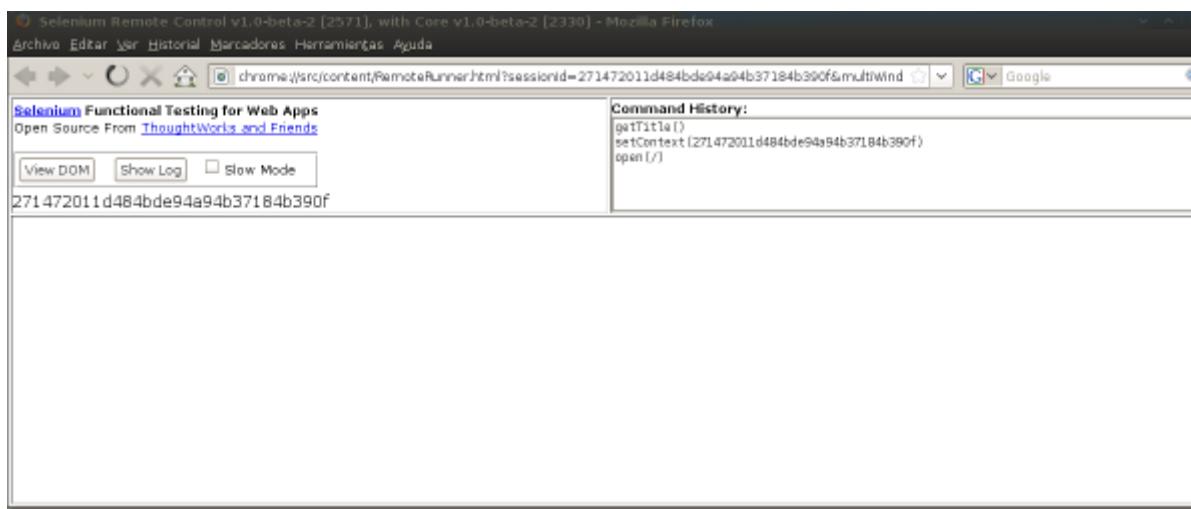
```
$ java -jar selenium-server-standalone-<version-number>.jar -Dhttp.proxyHost=
```

Multi-Window Mode

If you are using Selenium 1.0 you can probably skip this section, since multiwindow mode is the default behavior. However, prior to version 1.0, Selenium by default ran the application under test in a sub frame as shown here.



Some applications didn't run correctly in a sub frame, and needed to be loaded into the top frame of the window. The multi-window mode option allowed the AUT to run in a separate window rather than in the default frame where it could then have the top frame it required.



For older versions of Selenium you must specify multiwindow mode explicitly with the following option:

```
-multiwindow
```

As of Selenium RC 1.0, if you want to run your test within a single frame (i.e. using the standard for earlier Selenium versions) you can state this to the Selenium Server using the option

```
-singlewindow
```

Specifying the Firefox Profile

Firefox will not run two instances simultaneously unless you specify a separate profile for each instance. Selenium RC 1.0 and later runs in a separate profile automatically, so if you are using Selenium 1.0, you can probably skip this section. However, if you're using an older version of Selenium or if you need to use a specific profile for your tests (such as adding an https certificate or having some addons installed), you will need to explicitly specify the profile.

First, to create a separate Firefox profile, follow this procedure. Open the Windows Start menu, select "Run", then type and enter one of the following:

```
firefox.exe -profilemanager
```

```
firefox.exe -P
```

Create the new profile using the dialog. Then when you run Selenium Server, tell it to use this new Firefox profile with the server command-line option `-firefoxProfileTemplate` and specify the path to the profile using its filename and directory path.

```
-firefoxProfileTemplate "path to the profile"
```

Warning: Be sure to put your profile in a new folder separate from the default!!! The Firefox profile manager tool will delete all files in a folder if you delete a profile, regardless of whether they are profile files or not.

More information about Firefox profiles can be found in [Mozilla's Knowledge Base](#)

Run Selenese Directly Within the Server Using -htmlSuite

You can run Selenese html files directly within the Selenium Server by passing the html file to the server's command line. For instance:

```
java -jar selenium-server-standalone-<version-number>.jar -htmlSuite "*firefox"
"http://www.google.com" "c:\absolute\path\to\my\HTMLSuite.html"
"c:\absolute\path\to\my\results.html"
```

This will automatically launch your HTML suite, run all the tests and save a nice HTML report with the results.

Note: When using this option, the server will start the tests and wait for a specified number of seconds for the test to complete; if the test doesn't complete within that amount of time, the command will exit with a non-zero exit code and no results file will be generated.

This command line is very long so be careful when you type it. Note this requires you to pass in an HTML Selenese suite, not a single test. Also be aware the -htmlSuite option is incompatible with --interactive You cannot run both at the same time.

Selenium Server Logging

Server-Side Logs

When launching Selenium server the **-log** option can be used to record valuable debugging information reported by the Selenium Server to a text file.

```
java -jar selenium-server-standalone-<version-number>.jar -log selenium.log
```

This log file is more verbose than the standard console logs (it includes DEBUG level logging messages). The log file also includes the logger name, and the ID number of the thread that logged the message. For example:

```
20:44:25 DEBUG [12] org.openqa.selenium.server.SeleniumDriverResourceHandler
Browser 465828/:top frame1 posted START NEW
```

The message format is

```
TIMESTAMP(HH:mm:ss) LEVEL [THREAD] LOGGER - MESSAGE
```

This message may be multiline.

Browser-Side Logs

JavaScript on the browser side (Selenium Core) also logs important messages; in many cases, these can be more useful to the end-user than the regular Selenium Server logs. To access browser-side logs, pass the **-browserSideLog** argument to the Selenium Server.

```
java -jar selenium-server-standalone-<version-number>.jar -browserSideLog
```

-browserSideLog must be combined with the **-log** argument, to log browserSideLogs (as well as all other DEBUG level logging messages) to a file.

Specifying the Path to a Specific Browser

You can specify to Selenium RC a path to a specific browser. This is useful if you have different versions of the same browser and you wish to use a specific one. Also, this is used to allow your tests to run against a browser not directly supported by Selenium RC. When specifying the run mode, use the *custom specifier followed by the full path to the browser's executable:

```
*custom <path to browser>
```

Selenium RC Architecture

Note: This topic tries to explain the technical implementation behind Selenium RC. It's not fundamental for a Selenium user to know this, but could be useful for understanding some of the problems you might find in the future.

To understand in detail how Selenium RC Server works and why it uses proxy injection and heightened privilege modes you must first understand the [same origin policy](#).

The Same Origin Policy

The main restriction that Selenium faces is the Same Origin Policy. This security restriction is applied by every browser in the market and its objective is to ensure that a site's content will never be accessible by a script from another site. The Same Origin Policy dictates that any code loaded within the browser can only operate within that website's domain. It cannot perform functions on another website. So for example, if the browser loads JavaScript code when it loads www.mysite.com, it cannot run that loaded code against www.mysite2.com—even if that's another of your sites. If this were possible, a script placed on any website you open would be able to read information on your bank account if you had the account page opened on other tab. This is called XSS (Cross-site Scripting).

To work within this policy, Selenium-Core (and its JavaScript commands that make all the magic happen) must be placed in the same origin as the Application Under Test (same URL).

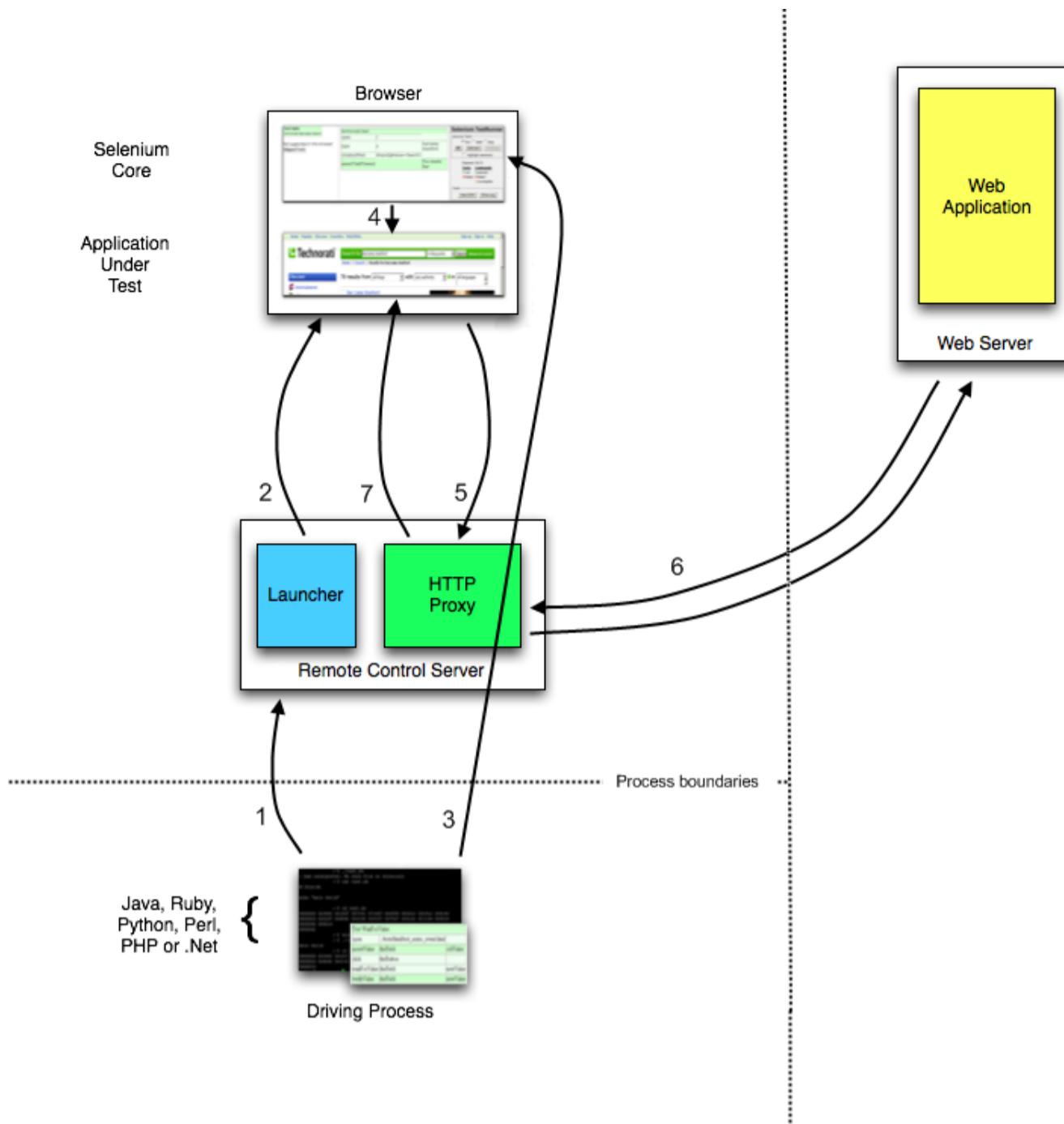
Historically, Selenium-Core was limited by this problem since it was implemented in JavaScript. Selenium RC is not, however, restricted by the Same Origin Policy. Its use of the Selenium Server as a proxy avoids this problem. It, essentially, tells the browser that the browser is working on a single “spoofed” website that the Server provides.

Note: You can find additional information about this topic on Wikipedia pages about Same Origin Policy and XSS.

Proxy Injection

The first method Selenium used to avoid the The Same Origin Policy was Proxy Injection. In Proxy Injection Mode, the Selenium Server acts as a client-configured [HTTP proxy](#)¹, that sits between the browser and the Application Under Test². It then masks the AUT under a fictional URL (embedding Selenium-Core and the set of tests and delivering them as if they were coming from the same origin).

Here is an architectural diagram.



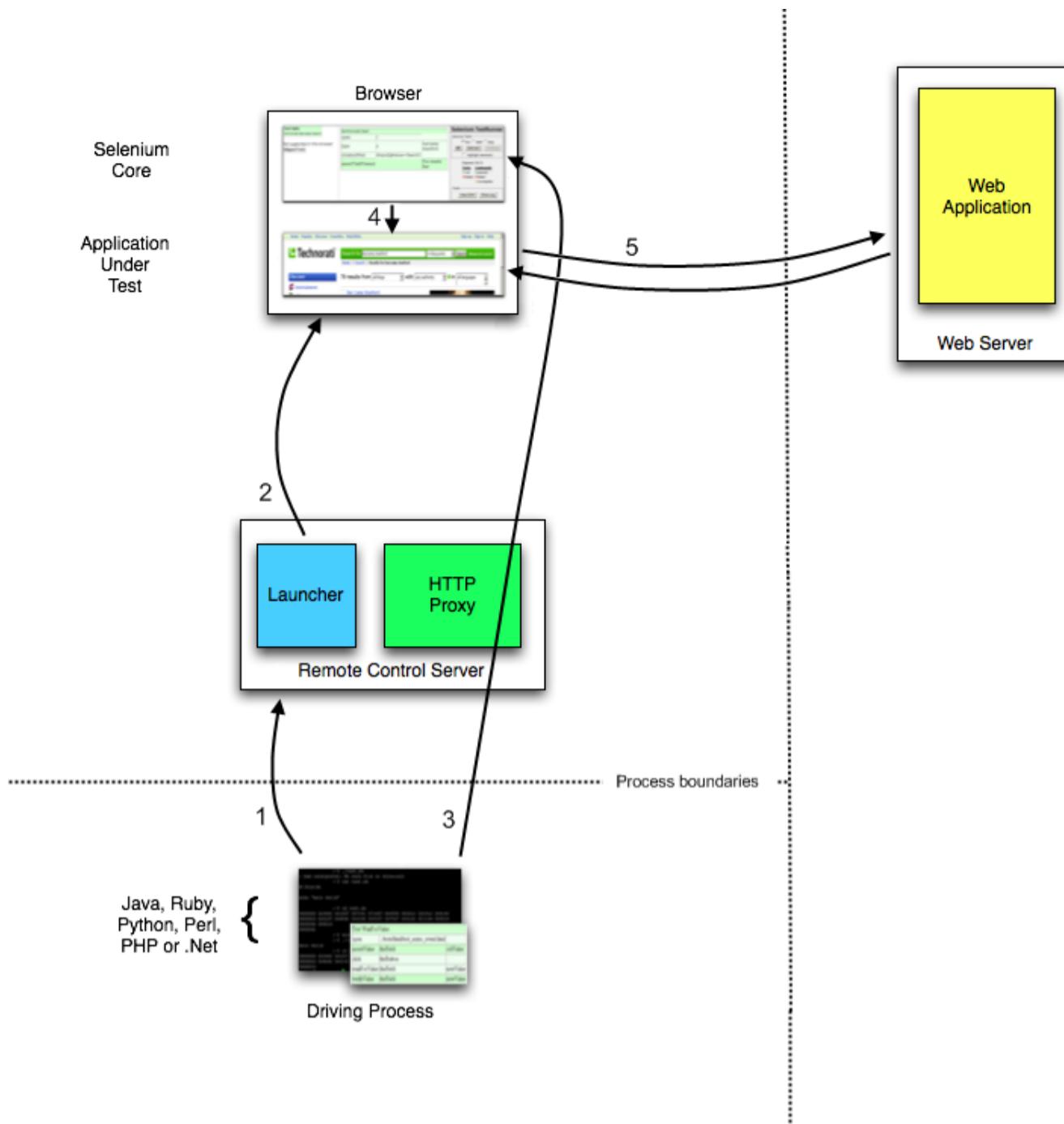
As a test suite starts in your favorite language, the following happens:

1. The client/driver establishes a connection with the selenium-RC server.
2. Selenium RC server launches a browser (or reuses an old one) with a URL that injects Selenium-Core's JavaScript into the browser-loaded web page.
3. The client-driver passes a Selenese command to the server.
4. The Server interprets the command and then triggers the corresponding JavaScript execution to execute that command within the browser. Selenium-Core instructs the browser to act on that first instruction, typically opening a page of the AUT.
5. The browser receives the open request and asks for the website's content from the Selenium RC server (set as the HTTP proxy for the browser to use).
6. Selenium RC server communicates with the Web server asking for the page and once it receives it, it sends the page to the browser masking the origin to look like the page comes from the same server as Selenium-Core (this allows Selenium-Core to comply with the Same Origin Policy).
7. The browser receives the web page and renders it in the frame/window reserved for it.

Heightened Privileges Browsers

This workflow in this method is very similar to Proxy Injection but the main difference is that the browsers are launched in a special mode called *Heightened Privileges*, which allows websites to do things that are not commonly permitted (as doing XSS_, or filling file upload inputs and pretty useful stuff for Selenium). By using these browser modes, Selenium Core is able to directly open the AUT and read/interact with its content without having to pass the whole AUT through the Selenium RC server.

Here is the architectural diagram.



As a test suite starts in your favorite language, the following happens:

1. The client/driver establishes a connection with the selenium-RC server.
2. Selenium RC server launches a browser (or reuses an old one) with a URL that will load Selenium-Core in the web page.
3. Selenium-Core gets the first instruction from the client/driver (via another HTTP request made to the Selenium RC Server).
4. Selenium-Core acts on that first instruction, typically opening a page of the AUT.
5. The browser receives the open request and asks the Web Server for the page. Once the browser receives the web page, renders it in the frame/window reserved for it.

Handling HTTPS and Security Popups

Many applications switch from using HTTP to HTTPS when they need to send encrypted information such as passwords or credit card information. This is common with many of today's web applications. Selenium RC supports this.

To ensure the HTTPS site is genuine, the browser will need a security certificate. Otherwise, when the browser accesses the AUT using HTTPS, it will assume that application is not 'trusted'. When this occurs the browser displays security popups, and these popups cannot be closed using Selenium RC.

When dealing with HTTPS in a Selenium RC test, you must use a run mode that supports this and handles the security certificate for you. You specify the run mode when your test program initializes Selenium.

In Selenium RC 1.0 beta 2 and later use *firefox or *iexplore for the run mode. In earlier versions, including Selenium RC 1.0 beta 1, use *chrome or *iehta, for the run mode. Using these run modes, you will not need to install any special security certificates; Selenium RC will handle it for you.

In version 1.0 the run modes *firefox or *iexplore are recommended. However, there are additional run modes of *iexploreproxy and *firefoxproxy. These are provided for backwards compatibility only, and should not be used unless required by legacy test programs. Their use will present limitations with

security certificate handling and with the running of multiple windows if your application opens additional browser windows.

In earlier versions of Selenium RC, *chrome or *iehta were the run modes that supported HTTPS and the handling of security popups. These were considered ‘experimental modes although they became quite stable and many people used them. If you are using Selenium 1.0 you do not need, and should not use, these older run modes.

Security Certificates Explained

Normally, your browser will trust the application you are testing by installing a security certificate which you already own. You can check this in your browser’s options or Internet properties (if you don’t know your AUT’s security certificate ask your system administrator). When Selenium loads your browser it injects code to intercept messages between the browser and the server. The browser now thinks untrusted software is trying to look like your application. It responds by alerting you with popup messages.

To get around this, Selenium RC, (again when using a run mode that support this) will install its own security certificate, temporarily, to your client machine in a place where the browser can access it. This tricks the browser into thinking it’s accessing a site different from your AUT and effectively suppresses the popups.

Another method used with earlier versions of Selenium was to install the Cybervillians security certificate provided with your Selenium installation. Most users should no longer need to do this however; if you are running Selenium RC in proxy injection mode, you may need to explicitly install this security certificate.

Supporting Additional Browsers and Browser Configurations

The Selenium API supports running against multiple browsers in addition to Internet Explorer and Mozilla Firefox. See the <https://selenium.dev> website for supported browsers. In addition, when a browser is not directly supported, you may still run your Selenium tests against a browser of your choosing by using the “*custom” run-mode (i.e. in place of *firefox or *iexplore) when your test application starts the browser. With this, you pass in the path to the browsers executable within the API call. This can also be done from the Server in interactive mode.

```
cmd=getNewBrowserSession&1=*custom c:\Program Files\Mozilla Firefox\MyBrowser
```

Running Tests with Different Browser Configurations

Normally Selenium RC automatically configures the browser, but if you launch the browser using the “*custom” run mode, you can force Selenium RC to launch the browser as-is, without using an automatic configuration.

For example, you can launch Firefox with a custom configuration like this:

```
cmd=getNewBrowserSession&1=*custom c:\Program Files\Mozilla Firefox\firefox.e
```

Note that when launching the browser this way, you must manually configure the browser to use the Selenium Server as a proxy. Normally this just means opening your browser preferences and specifying “localhost:4444” as an HTTP proxy, but instructions for this can differ radically from browser to browser. Consult your browser’s documentation for details.

Be aware that Mozilla browsers can vary in how they start and stop. One may need to set the MOZ_NO_REMOTE environment variable to make Mozilla browsers behave a little more predictably. Unix users should avoid launching the browser using a shell script; it’s generally better to use the

binary executable (e.g. firefox-bin) directly.

Troubleshooting Common Problems

When getting started with Selenium RC there's a few potential problems that are commonly encountered. We present them along with their solutions here.

Unable to Connect to Server

When your test program cannot connect to the Selenium Server, Selenium throws an exception in your test program. It should display this message or a similar one:

```
"Unable to connect to remote server (Inner Exception Message:  
No connection could be made because the target machine actively  
refused it )"  
  
(using .NET and XP Service Pack 2)
```

If you see a message like this, be sure you started the Selenium Server. If so, then there is a problem with the connectivity between the Selenium Client Library and the Selenium Server.

When starting with Selenium RC, most people begin by running their test program (with a Selenium Client Library) and the Selenium Server on the same machine. To do this use “localhost” as your connection parameter. We recommend beginning this way since it reduces the influence of potential networking problems which you’re getting started. Assuming your operating system has typical networking and TCP/IP settings you should have little difficulty. In truth, many people choose to run the tests this way.

If, however, you do want to run Selenium Server on a remote machine, the connectivity should be fine assuming you have valid TCP/IP connectivity between the two machines.

If you have difficulty connecting, you can use common networking tools like *ping*, *telnet*, *ifconfig*(Unix)/*ipconfig*(Windows), etc to ensure you have a valid network connection. If unfamiliar with these, your system administrator can assist you.

Unable to Load the Browser

Ok, not a friendly error message, sorry, but if the Selenium Server cannot load the browser you will likely see this error.

```
(500) Internal Server Error
```

This could be caused by

- Firefox (prior to Selenium 1.0) cannot start because the browser is already open and you did not specify a separate profile. See the section on Firefox profiles under Server Options.
- The run mode you’re using doesn’t match any browser on your machine. Check the parameters you passed to Selenium when your program opens the browser.
- You specified the path to the browser explicitly (using “*custom”—see above) but the path is incorrect. Check to be sure the path is correct. Also check the user group to be sure there are no known issues with your browser and the “*custom” parameters.

Selenium Cannot Find the AUT

If your test program starts the browser successfully, but the browser doesn’t display the website you’re testing, the most likely cause is your test program is not using the correct URL.

This can easily happen. When you use Selenium-IDE to export your script, it inserts a dummy URL. You must manually change the URL to the correct one for your application to be tested.

Firefox Refused Shutdown While Preparing a Profile

This most often occurs when you run your Selenium RC test program against Firefox, but you already have a Firefox browser session running and, you didn't specify a separate profile when you started the Selenium Server. The error from the test program looks like this:

```
Error: java.lang.RuntimeException: Firefox refused shutdown while
preparing a profile
```

Here's the complete error message from the server:

```
16:20:03.919 INFO - Preparing Firefox profile...
16:20:27.822 WARN - GET /selenium-server/driver/?cmd=getNewBrowserSession&1=
efox&2=http%3a%2f%2fsage-webapp1.qa.idc.com HTTP/1.1
java.lang.RuntimeException: Firefox refused shutdown while preparing a profile
    at org.openqa.selenium.server.browserlaunchers.FirefoxCustomProfileL
her.waitForFullProfileToBeCreated(FirefoxCustomProfileLauncher.java:277)
...
Caused by: org.openqa.selenium.server.browserlaunchers.FirefoxCustomProfileL
her$FileLockRemainedException: Lock file still present! C:\DOCUME~1\jsvec\LO
~1\Temp\customProfileDir203138\parent.lock
```

To resolve this, see the section on Specifying a Separate Firefox Profile

Versioning Problems

Make sure your version of Selenium supports the version of your browser. For example, Selenium RC 0.92 does not support Firefox 3. At times you may be lucky (I was). But don't forget to check which browser versions are supported by the version of Selenium you are using. When in doubt, use the latest release version of Selenium with the most widely used version of your browser.

Error message: “(Unsupported major.minor version 49.0)” while starting server

This error says you're not using a correct version of Java. The Selenium Server requires Java 1.5 or higher.

To check double-check your java version, run this from the command line.

```
java -version
```

You should see a message showing the Java version.

```
java version "1.5.0_07"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-b03)
Java HotSpot(TM) Client VM (build 1.5.0_07-b03, mixed mode)
```

If you see a lower version number, you may need to update the JRE, or you may simply need to add it to your PATH environment variable.

404 error when running the getNewBrowserSession command

If you're getting a 404 error while attempting to open a page on "<http://www.google.com/selenium-server/>", then it must be because the Selenium Server was not correctly configured as a proxy. The "selenium-server" directory doesn't exist on google.com; it only appears to exist when the proxy is properly configured. Proxy Configuration highly depends on how the browser is launched with firefox, iexplore, opera, or custom.

- iexplore: If the browser is launched using *iexplore, you could be having a problem with Internet Explorer's proxy settings. Selenium Server attempts to configure the global proxy settings in the Internet Options Control Panel. You must make sure that those are correctly configured when Selenium Server launches the browser. Try looking at your Internet Options control panel. Click on the "Connections" tab and click on "LAN Settings".
 - If you need to use a proxy to access the application you want to test, you'll need to start Selenium Server with "-Dhttp.proxyHost"; see the [Proxy Configuration](#) for more details.
 - You may also try configuring your proxy manually and then launching the browser with *custom, or with *iehta browser launcher.
- custom: When using *custom you must configure the proxy correctly(manually), otherwise you'll get a 404 error. Double-check that you've configured your proxy settings correctly. To check whether you've configured the proxy correctly is to attempt to intentionally configure the browser incorrectly. Try configuring the browser to use the wrong proxy server hostname, or the wrong port. If you had successfully configured the browser's proxy settings incorrectly, then the browser will be unable to connect to the Internet, which is one way to make sure that one is adjusting the relevant settings.
- For other browsers (*firefox, *opera) we automatically hard-code the proxy for you, and so there are no known issues with this functionality. If you're encountering 404 errors and have followed this user guide carefully post your results to user group for some help from the user community.

Permission Denied Error

The most common reason for this error is that your session is attempting to violate the same-origin policy by crossing domain boundaries (e.g., accesses a page from `http://domain1` and then accesses a page from `http://domain2`) or switching protocols (moving from `http://domainX` to `https://domainX`).

This error can also occur when JavaScript attempts to find UI objects which are not yet available (before the page has completely loaded), or are no longer available (after the page has started to be unloaded). This is most typically encountered with AJAX pages which are working with sections of a page or subframes that load and/or reload independently of the larger page.

This error can be intermittent. Often it is impossible to reproduce the problem with a debugger because the trouble stems from race conditions which are not reproducible when the debugger's overhead is added to the system. Permission issues are covered in some detail in the tutorial. Read the section about the [The Same Origin Policy](#), [Proxy Injection](#) carefully.

Handling Browser Popup Windows

There are several kinds of "Popups" that you can get during a Selenium test. You may not be able to close these popups by running Selenium commands if they are initiated by the browser and not your AUT. You may need to know how to manage these. Each type of popup needs to be addressed differently.

- HTTP basic authentication dialogs: These dialogs prompt for a username/password to login to the site. To login to a site that requires HTTP basic authentication, use a username and password in the URL, as described in [RFC 1738](#), like this:
`open("http://myusername:myuserpassword@myexample.com/blah/blah").`
- SSL certificate warnings: Selenium RC automatically attempts to spoof SSL certificates when it is enabled as a proxy; see more on this in the section on HTTPS. If your browser is configured correctly, you should never see SSL certificate warnings, but you may need to configure your browser to trust our dangerous "CyberVillains" SSL certificate authority. Again, refer to the HTTPS section for how to do this.

- modal JavaScript alert/confirmation/prompt dialogs: Selenium tries to conceal those dialogs from you (by replacing window.alert, window.confirm and window.prompt) so they won't stop the execution of your page. If you're seeing an alert pop-up, it's probably because it fired during the page load process, which is usually too early for us to protect the page. Selenese contains commands for asserting or verifying alert and confirmation popups. See the sections on these topics in Chapter 4.

On Linux, why isn't my Firefox browser session closing?

On Unix/Linux you must invoke “firefox-bin” directly, so make sure that executable is on the path. If executing Firefox through a shell script, when it comes time to kill the browser Selenium RC will kill the shell script, leaving the browser running. You can specify the path to firefox-bin directly, like this.

```
cmd=getNewBrowserSession&1=*firefox /usr/local/firefox/firefox-bin&2=http://w
```

Firefox *chrome doesn't work with custom profile

Check Firefox profile folder -> prefs.js -> user_pref("browser.startup.page", 0); Comment this line like this: //user_pref("browser.startup.page", 0); and try again.

Is it ok to load a custom pop-up as the parent page is loading (i.e., before the parent page's javascript window.onload() function runs)?

No. Selenium relies on interceptors to determine window names as they are being loaded. These interceptors work best in catching new windows if the windows are loaded AFTER the onload() function. Selenium may not recognize windows loaded before the onload function.

Firefox on Linux

On Unix/Linux, versions of Selenium before 1.0 needed to invoke “firefox-bin” directly, so if you are using a previous version, make sure that the real executable is on the path.

On most Linux distributions, the real *firefox-bin* is located on:

```
/usr/lib/firefox-x.x.x/
```

Where the x.x.x is the version number you currently have. So, to add that path to the user's path. you will have to add the following to your .bashrc file:

```
export PATH="$PATH:/usr/lib/firefox-x.x.x/"
```

If necessary, you can specify the path to firefox-bin directly in your test, like this:

```
"*firefox /usr/lib/firefox-x.x.x/firefox-bin"
```

IE and Style Attributes

If you are running your tests on Internet Explorer and you cannot locate elements using their `style` attribute. For example:

```
//td[@style="background-color:yellow"]
```

This would work perfectly in Firefox, Opera or Safari but not with IE. IE interprets the keys in `@style` as uppercase. So, even if the source code is in lowercase, you should use:

```
//td[@style="BACKGROUND-COLOR:yellow"]
```

This is a problem if your test is intended to work on multiple browsers, but you can easily code your test to detect the situation and try the alternative locator that only works in IE.

Error encountered - “Cannot convert object to primitive value” with shut down of *googlechrome browser

To avoid this error you have to start browser with an option that disables same origin policy checks:

```
selenium.start("commandLineFlags=--disable-web-security");
```

Error encountered in IE - “Couldn’t open app window; is the pop-up blocker enabled?”

To avoid this error you have to configure the browser: disable the popup blocker AND uncheck ‘Enable Protected Mode’ option in Tools » Options » Security.

1. The proxy is a third person in the middle that passes the ball between the two parts. It acts as a “web server” that delivers the AUT to the browser. Being a proxy gives Selenium Server the capability of “lying” about the AUT’s real URL. 
2. The browser is launched with a configuration profile that has set localhost:4444 as the HTTP proxy, this is why any HTTP request that the browser does will pass through Selenium server and the response will pass through it and not from the real server. 

7.2 - Selenium 2

Selenium 2 was a rewrite of Selenium 1 that was implemented with WebDriver code.

7.2.1 - Migrating from RC to WebDriver

Information on Updating from Selenium 1 to Selenium 2.

How to Migrate to Selenium WebDriver

A common question when adopting Selenium 2 is what's the correct thing to do when adding new tests to an existing set of tests? Users who are new to the framework can begin by using the new WebDriver APIs for writing their tests. But what of users who already have suites of existing tests? This guide is designed to demonstrate how to migrate your existing tests to the new APIs, allowing all new tests to be written using the new features offered by WebDriver.

The method presented here describes a piecemeal migration to the WebDriver APIs without needing to rework everything in one massive push. This means that you can allow more time for migrating your existing tests, which may make it easier for you to decide where to spend your effort.

This guide is written using Java, because this has the best support for making the migration. As we provide better tools for other languages, this guide shall be expanded to include those languages.

Why Migrate to WebDriver

Moving a suite of tests from one API to another API requires an enormous amount of effort. Why would you and your team consider making this move? Here are some reasons why you should consider migrating your Selenium Tests to use WebDriver.

- Smaller, compact API. WebDriver's API is more Object Oriented than the original Selenium RC API. This can make it easier to work with.
- Better emulation of user interactions. Where possible, WebDriver makes use of native events in order to interact with a web page. This more closely mimics the way that your users work with your site and apps. In addition, WebDriver offers the advanced user interactions APIs which allow you to model complex interactions with your site.
- Support by browser vendors. Opera, Mozilla and Google are all active participants in WebDriver's development, and each have engineers working to improve the framework. Often, this means that support for WebDriver is baked into the browser itself: your tests run as fast and as stably as possible.

Before Starting

In order to make the process of migrating as painless as possible, make sure that all your tests run properly with the latest Selenium release. This may sound obvious, but it's best to have it said!

Getting Started

The first step when starting the migration is to change how you obtain your instance of Selenium. When using Selenium RC, this is done like so:

```
Selenium selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://w
selenium.start();
```

This should be replaced like so:

```
WebDriver driver = new FirefoxDriver();
Selenium selenium = new WebDriverBackedSelenium(driver, "http://www.yoursite.com")
```

Next Steps

Once your tests execute without errors, the next stage is to migrate the actual test code to use the WebDriver APIs. Depending on how well abstracted your code is, this might be a short process or a long one. In either case, the approach is the same and can be summed up simply: modify code to use the new API when you come to edit it.

If you need to extract the underlying WebDriver implementation from the Selenium instance, you can simply cast it to WrapsDriver:

```
WebDriver driver = ((WrapsDriver) selenium).getWrappedDriver();
```

This allows you to continue passing the Selenium instance around as normal, but to unwrap the WebDriver instance as required.

At some point, you're codebase will mostly be using the newer APIs. At this point, you can flip the relationship, using WebDriver throughout and instantiating a Selenium instance on demand:

```
Selenium selenium = new WebDriverBackedSelenium(driver, baseUrl);
```

Common Problems

Fortunately, you're not the first person to go through this migration, so here are some common problems that others have seen, and how to solve them.

Clicking and Typing is More Complete

A common pattern in a Selenium RC test is to see something like:

```
selenium.type("name", "exciting tex");
selenium.keyDown("name", "t");
selenium.keyPress("name", "t");
selenium.keyUp("name", "t");
```

This relies on the fact that “type” simply replaces the content of the identified element without also firing all the events that would normally be fired if a user interacts with the page. The final direct invocations of “key*” cause the JS handlers to fire as expected.

When using the WebDriverBackedSelenium, the result of filling in the form field would be “exciting texttt”: not what you'd expect! The reason for this is that WebDriver more accurately emulates user behavior, and so will have been firing events all along.

This same fact may sometimes cause a page load to fire earlier than it would do in a Selenium 1 test. You can tell that this has happened if a “StaleElementException” is thrown by WebDriver.

WaitForPageToLoad Returns Too Soon

Discovering when a page load is complete is a tricky business. Do we mean “when the load event fires”, “when all AJAX requests are complete”, “when there’s no network traffic”, “when document.readyState has changed” or something else entirely?

WebDriver attempts to simulate the original Selenium behavior, but this doesn’t always work perfectly for various reasons. The most common reason is that it’s hard to tell the difference between a page load not having started yet, and a page load having completed between method calls. This sometimes means that control is returned to your test before the page has finished (or even started!) loading.

The solution to this is to wait on something specific. Commonly, this might be for the element you want to interact with next, or for some Javascript variable to be set to a specific value. An example would be:

```
Wait<WebDriver> wait = new WebDriverWait(driver, Duration.ofSeconds(30));
WebElement element= wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("some_id")));
```

Where “visibilityOfElementLocated” is implemented as:

```
public ExpectedCondition<WebElement> visibilityOfElementLocated(final By locator) {
    return new ExpectedCondition<WebElement>() {
        public WebElement apply(WebDriver driver) {
            WebElement toReturn = driver.findElement(locator);
            if (toReturn.isDisplayed()) {
                return toReturn;
            }
            return null;
        }
    };
}
```

This may look complex, but it’s almost all boiler-plate code. The only interesting bit is that the “ExpectedCondition” will be evaluated repeatedly until the “apply” method returns something that is neither “null” nor Boolean.FALSE.

Of course, adding all these “wait” calls may clutter up your code. If that’s the case, and your needs are simple, consider using the implicit waits:

```
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
```

By doing this, every time an element is located, if the element is not present, the location is retried until either it is present, or until 30 seconds have passed.

Finding By XPath or CSS Selectors Doesn’t Always Work, But It Does In Selenium 1

In Selenium 1, it was common for xpath to use a bundled library rather than the capabilities of the browser itself. WebDriver will always use the native browser methods unless there’s no alternative. That means that complex xpath expressions may break on some browsers.

CSS Selectors in Selenium 1 were implemented using the Sizzle library. This implements a superset of the CSS Selector spec, and it’s not always clear where you’ve crossed the line. If you’re using the WebDriverBackedSelenium and use a Sizzle locator instead of a CSS Selector for finding elements, a warning will be logged to the console. It’s worth taking the time to look for these, particularly if tests are failing because of not being able to find elements.

There is No Browserbot

Selenium RC was based on Selenium Core, and therefore when you executed Javascript, you could access bits of Selenium Core to make things easier. As WebDriver is not based on Selenium Core, this is no longer possible. How can you tell if you're using Selenium Core? Simple! Just look to see if your "getEval" or similar calls are using "selenium" or "browserbot" in the evaluated Javascript.

You might be using the browserbot to obtain a handle to the current window or document of the test. Fortunately, WebDriver always evaluates JS in the context of the current window, so you can use "window" or "document" directly.

Alternatively, you might be using the browserbot to locate elements. In WebDriver, the idiom for doing this is to first locate the element, and then pass that as an argument to the Javascript. Thus:

```
String name = selenium.getEval(  
    "selenium.browserbot.findElement('id=foo', browserbot.getCurrentWindow()).ta
```

becomes:

```
WebElement element = driver.findElement(By.id("foo"));  
String name = (String) ((JavascriptExecutor) driver).executeScript(  
    "return arguments[0].tagName", element);
```

Notice how the passed in "element" variable appears as the first item in the JS standard "arguments" array.

Executing Javascript Doesn't Return Anything

WebDriver's JavascriptExecutor will wrap all JS and evaluate it as an anonymous expression. This means that you need to use the "return" keyword:

```
String title = selenium.getEval("browserbot.getCurrentWindow().document.title");
```

becomes:

```
((JavascriptExecutor) driver).executeScript("return document.title;");
```

7.2.2 - Backing Selenium with WebDriver

The Java and .NET versions of Selenium 2 provided implementations of the original Selenium API

(Previously located: <https://github.com/SeleniumHQ/selenium/wiki/Selenium-Emulation>)

Backing Selenium with WebDriver

The Java and .NET versions of WebDriver provide implementations of the existing Selenium API. In Java, it is used like so:

```
// You may use any WebDriver implementation. Firefox is used here as an example
WebDriver driver = new FirefoxDriver();

// A "base url", used by selenium to resolve relative URLs
String baseUrl = "http://www.google.com";

// Create the Selenium implementation
Selenium selenium = new WebDriverBackedSelenium(driver, baseUrl);

// Perform actions with selenium
selenium.open("http://www.google.com");
selenium.type("name=q", "cheese");
selenium.click("name=btnG");

// And get the underlying WebDriver implementation back. This will refer to the
// same WebDriver instance as the "driver" variable above.
WebDriver driverInstance = ((WebDriverBackedSelenium) selenium).getUnderlyingWebDriver();
```

Pros

- Allows for WebDriver and Selenium to live side-by-side.
- Provides a simple mechanism for a managed migration from the existing Selenium API to WebDriver's.
- Does not require the standalone Selenium RC server to be run

Cons

- Does not implement every method
 - But we'd love feedback!
- Does also emulate Selenium Core
 - So more advanced Selenium usage (that is, using "browserbot" or other built-in Javascript methods from Selenium Core) may need work
- Some methods may be slower due to underlying implementation differences
- Does not support Selenium's "user extensions" (*i.e.*, user-extensions.js)

Notes

After creating a `WebDriverBackedSelenium` instance with a given Driver, one does not have to call `start()` - as the creation of the Driver already started the session. At the end of the test, `stop()` should be called **instead** of the Driver's `quit()` method.

This is more similar to WebDriver's behaviour - as creating a Driver instance starts a session, yet it has to be terminated explicitly with a call to `quit()`.

Backing Selenium with RemoteWebDriver

Starting with release 2.19, `WebDriverBackedSelenium` can be used from any language supported by WebDriver and Selenium.

For example, in Python:

```
driver = RemoteWebDriver(desired_capabilities = DesiredCapabilities.FIREFOX)
selenium = DefaultSelenium('localhost', '4444', '*webdriver', 'http://www.google.com')
selenium.start(driver = driver)
```

Provided you keep a reference to the original WebDriver and Selenium objects you created, you can use even the two APIs interchangeably. The magic is the “`*webdriver`” browser name passed to the Selenium instance, and that you pass the WebDriver instance when calling `start()`.

In languages where `DefaultSelenium` doesn’t have `start(driver)`, you can connect the WebDriver and Selenium objects together yourself, by supplying the WebDriver session ID to the Selenium object.

For example, in C#:

```
RemoteWebDriver driver = new RemoteWebDriver(DesiredCapabilities.Firefox());
string sessionId = (string) driver.Capabilities.GetCapability("webdriver.remote.sessionid");
DefaultSelenium selenium = new DefaultSelenium("localhost", 4444, "*webdriver", "http://www.
selenium.Start("webdriver.remote.sessionid=" + sessionId);
```

Backing WebDriver with Selenium

WebDriver doesn’t support as many browsers as Selenium does, so in order to provide that support while still using the `webdriver` API, you can make use of the `SeleneseCommandExecutor`. It is done like this:

```
Capabilities capabilities = new DesiredCapabilities()
capabilities.setBrowserName("safari");
CommandExecutor executor = new SeleneseCommandExecutor("http:localhost:4444/", "http://www.g
WebDriver driver = new RemoteWebDriver(executor, capabilities);
```

There are currently some major limitations with this approach, notably that `findElements` doesn’t work as expected. Also, because we’re using Selenium Core for the heavy lifting of driving the browser, you are limited by the Javascript sandbox.

7.2.3 - Legacy Firefox Driver

The legacy Firefox Driver was developed as a browser extension by the Selenium Developers. Firefox updated their security model, so it no longer works. You now need to use geckodriver.

This documentation previously located [on the wiki](#)
You can read more about [geckodriver](#).

About

Firefox driver is included in the selenium-server-stanalone.jar available in the downloads. The driver comes in the form of an xpi (firefox extension) which is added to the firefox profile when you start a new instance of FirefoxDriver.

Pros

- Runs in a real browser and supports Javascript
- Faster than the InternetExplorerDriver

Cons

- Slower than the HtmlUnitDriver

Important System Properties

The following system properties (read using `System.getProperty()` and set using `System.setProperty()` in Java code or the “`-DpropertyName=value`” command line flag) are used by the FirefoxDriver:

Property	What it means
<code>webdriver.firefox.bin</code>	The location of the binary used to control firefox.
<code>webdriver.firefox.marionette</code>	Boolean value, if set on standalone-server will ignore any “marionette” desired capability requested and force firefox to use GeckoDriver (true) or Legacy Firefox Driver (false)
<code>webdriver.firefox.profile</code>	The name of the profile to use when starting firefox. This defaults to webdriver creating an anonymous profile
<code>webdriver.firefox.useExisting</code>	Never use in production Use a running instance of firefox if one is present
<code>webdriver.firefox.logfile</code>	Log file to dump firefox stdout/stderr to

Normally the Firefox binary is assumed to be in the default location for your particular operating system:

OS	Expected Location of Firefox
Linux	firefox (found using “which”)
Mac	/Applications/Firefox.app/Contents/MacOS/firefox-bin
Windows	%PROGRAMFILES%\Mozilla Firefox\firefox.exe

By default, the Firefox driver creates an anonymous profile

Running with firebug

Download the firebug xpi file from mozilla and start the profile as follows:

```
File file = new File("firebug-1.8.1.xpi");
FirefoxProfile firefoxProfile = new FirefoxProfile();
firefoxProfile.addExtension(file);
firefoxProfile.setPreference("extensions.firebug.currentVersion", "1.8.1"); // Avoid starting the browser with the extension loaded

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

FirefoxDriver Internals

(Previously located: <https://github.com/SeleniumHQ/selenium/wiki/FirefoxDriver-Internals>)

The FirefoxDriver is largely written in the form of a Firefox extension. Language bindings control the driver by connecting over a socket and sending commands (described in the [JsonWireProtocol](#) page) in UTF-8. The extension makes use of the XPCOM primitives offered by Firefox in order to do its work. The important thing to notice is that the command names map directly onto methods exposed on the “FirefoxDriver.prototype” in the javascript code.

Working on the FirefoxDriver Code

Firstly, make sure that there's no old version of the FirefoxDriver installed:

- Start firefox's profile manager: `firefox -ProfileManager`
- Delete the existing WebDriver profile if there is one. Delete the files too (it's an option that's offered when you delete the profile in the profile manager)

Secondly, take a look at the [Mozilla Developer Center](#), particularly the section to do with [setting up an extension development environment](#). You should now be ready to edit code. It's best to create a test around the area of code that you're working on, and to run this using the `SingleTestSuite`. The FirefoxDriver logs errors to Firefox's error console (“Tools->Error Console”), so if a test fails, that's a great place to start looking.

To actually log information to the console, use the “`utils.dumpn()`” method in your javascript code. If you find that you'd like to examine an object in detail, use the “`utils.dump()`” method, which will report which interfaces an object implements, as well as outputting as much information as it can to the console..

Flow of Control: Starting Firefox

The following steps are performed when instantiating an instance of the FirefoxDriver:

1. Grab the “locking port”

7.2.4 - Selenium grid 2

Selenium Grid 2 supported WebDriver and Selenium RC. It was replaced by Grid 3 which removed RC code. Grid 3 was completely rewritten for the new Grid 4.

This documentation previously located [on the wiki](#)

You can read our documentation for more information about [Grid 4](#)

Introduction

Grid allows you to :

- scale by distributing tests on several machines (parallel execution)
- manage multiple environments from a central point, making it easy to run the tests against a vast combination of browsers / OS.
- minimize the maintenance time for the grid by allowing you to implement custom hooks to leverage virtual infrastructure for instance.

Quick Start

This example will show you how to start the Selenium 2 Hub, and register both a WebDriver node and a Selenium 1 RC legacy node. We'll also show you how to call the grid from Java. The hub and nodes are shown here running on the same machine, but of course you can copy the selenium-server-standalone to multiple machines. Note: The selenium-server-standalone package includes the Hub, WebDriver, and legacy RC needed to run the grid. Ant is not required anymore. You can download the selenium-server-standalone- * .jar from <http://selenium-release.storage.googleapis.com/index.html> This walk-through assumes you already have Java installed.

Step 1: Start the hub

The Hub is the central point that will receive all the test request and distribute them to the right nodes.

Open a command prompt and navigate to the directory where you copied the selenium-server-standalone file. Type the following command:

```
java -jar selenium-server-standalone-<version>.jar -role hub
```

The hub will automatically start-up using port 4444 by default. To change the default port, you can add the optional parameter -port when you run the command. You can view the status of the hub by opening a browser window and navigating to: <http://localhost:4444/grid/console>

Step 2: Start the nodes

Regardless on whether you want to run a grid with new WebDriver functionality, or a grid with Selenium 1 RC functionality, or both at the same time, you use the same selenium-server-standalone jar file to start the nodes.

```
java -jar selenium-server-standalone-<version>.jar -role node -hub http://localhost:4444/gr
```

Note: The port defaults to 5555 if not specified whenever the “-role” option is provided and is not hub.

For backwards compatibility “wd” and “rc” roles are still a valid subset of the “node” role. But those roles limit the types of remote connections to their corresponding API, while “node” allows both RC and WebDriver remote connections.

Using grid to run tests

(using java as an example) Now that the grid is in-place, we need to access the grid from our test cases. For the Selenium 1 RC nodes, you can continue to use the DefaultSelenium object and pass in the hub information:

```
Selenium selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://www.google.co
```

For WebDriver nodes, you will need to use the **RemoteWebDriver** and the **DesiredCapabilities** object to define which browser, version and platform you wish to use. Create the target browser capabilities you want to run the tests against:

```
DesiredCapabilities capability = DesiredCapabilities.firefox();
```

Pass that into the RemoteWebDriver object:

```
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444/wd/hub"), capability);
```

The hub will then assign the test to a matching node.

A node matches if all the requested capabilities are met. To request specific capabilities on the grid, specify them before passing it into the WebDriver object.

```
capability.setBrowserName();
capability.setPlatform();
capability.setVersion()
capability.setCapability(,);
```

Example: A node registered with the setting:

```
-browser browserName=firefox,version=3.6,platform=LINUX
```

will be a match for:

```
capability.setBrowserName("firefox" );
capability.setPlatform("LINUX");
capability.setVersion("3.6");
```

and would also be a match for

```
capability.setBrowserName("firefox" );
capability.setVersion("3.6");
```

The capabilities that are not specified will be ignored. If you specify capabilities that do not exist on your grid (for example, your test specifies Firefox version 4.0, but have no Firefox 4 instance) then there will be no match and the test will fail to run.

Configuring the nodes

The node can be configured in 2 different ways; one is by specifying command-line parameters, the other is by specifying a JSON file.

Configuring the nodes by command line

By default, starting the node allows for concurrent use of 11 browsers...: 5 Firefox, 5 Chrome, 1 Internet Explorer. The maximum number of concurrent tests is set to 5 by default. To change this and other browser settings, you can pass in parameters to each -browser switch (each switch represents a node

based on your parameters). If you use the -browser parameter, the default browsers will be ignored and only what you specify command line will be used.

```
-browser browserName=firefox,version=3.6,maxInstances=5,platform=LINUX
```

This setting starts 5 Firefox 3.6 nodes on a Linux machine.

If your remote machine has multiple versions of Firefox you'd like to use, you can map the location of each binary to a particular version on the same machine:

```
-browser browserName=firefox,version=3.6,firefox_binary=/home/myhomedir/firefox36/firefox,ma
```

Tip: If you need to provide a space somewhere in your browser parameters, then surround the parameters with quotation marks:

```
-browser "browserName=firefox,version=3.6,firefox_binary=c:\Program Files\firefox ,maxInstan
```

Optional parameters

- `-port 4444` (4444 is default)
- `-host <IP | hostname>` specify the hostname or IP. usually not needed and determined automatically. For exotic network configuration, network with VPN, specifying the host might be necessary.
- `-timeout 30` (300 is default) The timeout in seconds before the hub automatically releases a node that hasn't received any requests for more than the specified number of seconds. After this time, the node will be released for another test in the queue. This helps to clear client crashes without manual intervention. To remove the timeout completely, specify `-timeout 0` and the hub will never release the node.

Note: This is NOT the WebDriver timeout for all "wait for WebElement" type of commands.

- `-maxSession 5` (5 is default) The maximum number of browsers that can run in parallel on the node. This is different from the maxInstance of supported browsers (Example: For a node that supports Firefox 3.6, Firefox 4.0 and Internet Explorer 8, maxSession=1 will ensure that you never have more than 1 browser running. With maxSession=2 you can have 2 Firefox tests at the same time, or 1 Internet Explorer and 1 Firefox test).
- `-browser < params >` If `-browser` is not set, a node will start with 5 firefox, 1 chrome, and 1 internet explorer instance (assuming it's on a windows box). This parameter can be set multiple times on the same line to define multiple types of browsers. Parameters allowed for `-browser`:
`browserName={android, chrome, firefox, htmlunit, internet explorer, iphone, opera} version={browser version} firefox_binary={path to executable binary} chrome_binary={path to executable binary} maxInstances={maximum number of browsers of this type} platform={WINDOWS, LINUX, MAC}`
- `-registerCycle N` = how often in ms the node will try to register itself again. Allow to restart the hub without having to restart the nodes.
- Really large (>50 node) Hub installations may need to increase the jetty threads by setting `-DPOOL_MAX=512` (or larger) on the java command line.

Configuring timeouts (Version 2.21 required)

Timeouts in the grid should normally be handled through `webDriver.manage().timeouts()`, which will control how the different operations time out.

To preserve run-time integrity of a grid with selenium-servers, there are two other timeout values that can be set.

On the hub, setting the -timeout command line option to “30” seconds will ensure all resources are reclaimed 30 seconds after a client crashes. On the hub you can also set -browserTimeout 60 to make the maximum time a node is willing to hang inside the browser 60 seconds. This will ensure all resources are reclaimed slightly after 60 seconds. All the nodes use these two values from the hub if they are set. Locally set parameters on a single node has precedence, it is generally recommended not to set these timeouts on the node.

The browserTimeout **should** be:

- Higher than the socket lock timeout (45 seconds)
- Generally higher than values used in webDriver.manage().timeouts(), since this mechanism is a “last line of defense”.

Configuring the nodes by JSON

```
java -jar selenium-server-standalone.jar -role node -nodeConfig nodeconfig.json
```

A sample nodeconfig file for server version 3.x.x (>= beta4) can be seen at

<https://github.com/SeleniumHQ/selenium/blob/selenium-3.141.59/java/server/src/org/openqa/grid/common/defaults/DefaultNodeWebDriver.json>

A sample nodeconfig file for server version 2.x.x can be seen at

<https://github.com/SeleniumHQ/selenium/blob/selenium-2.53.0/java/server/src/org/openqa/grid/common/defaults/DefaultNode.json>

Note: the configuration { ... } object in version 2.x.x has been flattened in version 3.x.x (>= beta4)

Configuring the hub by JSON

```
java -jar selenium-server-standalone.jar -role hub -hubConfig hubconfig.json
```

A sample hubconfig.json file can be seen at <https://github.com/SeleniumHQ/selenium/blob/selenium-3.141.59/java/server/src/org/openqa/grid/common/defaults/DefaultHub.json>

Hub diagnostic messages

Upon detecting anomalous usage patterns, the hub can give the following message:

```
Client requested session XYZ that was terminated due to REASON
```

Reason	Cause/fix
TIMEOUT	The session timed out because the client did not access it within the timeout. If the client has been somehow suspended, this may happen when it wakes up
BROWSER_TIMEOUT	The node timed out the browser because it was hanging for too long (parameter browserTimeout)
ORPHAN	A client waiting in queue has given up once it was offered a new session
CLIENT_STOPPED_SESSION	The session was stopped using an ordinary call to stop/quit on the client. Why are you using it again??
CLIENT_GONE	The client process (<i>your code</i>) appears to have died or otherwise not responded to our requests, intermittent network issues may also cause

Reason	Cause/fix
FORWARDING_TO_NODE_FAILED	The hub was unable to forward to the node. Out of memory errors/node stability issues or network problems
CREATIONFAILED	The node failed to create the browser. This can typically happen when there are environmental/configuration problems on the node. Try using the node directly to track problem.
PROXY_REREGISTRATION	The session has been discarded because the node has re-registered on the grid (in mid-test)

Tips for running with grid

If your tests are running in parallel, make sure that each thread deallocates its webdriver resource independently of any other tests running on other threads. Starting 1 browser per thread at the start of the test-run and deallocating all browsers at the end is **not** a good idea. (If one test-case decides to consume abnormal amounts of time you may get timeouts on all the other tests because they're waiting for the slow test. This can be very confusing)

Selenium Grid Platforms

(previously located: <https://github.com/SeleniumHQ/selenium/wiki/Grid-Platforms>)

This section describes the PLATFORM option used in configuring Selenium Grid Nodes and [DesiredCapabilities] object.

History of Platforms

When requesting a new WebDriver session from the Grid, user can specify the [DesiredCapabilities] of the remote browser. Things such as the browser name, version, and platform are among the list of options that can be specified by the test. Specifying desired.

The following code demonstrates the DesiredCapability of Internet Explorer, version 9, on Windows XP platform:

```
[[DesiredCapabilities]] capability = DesiredCapabilities.internetExplorer();
capability.setVersion("8");
capability.setPlatform(Platform.XP);
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444/wd/hub"), capa
```

The request for a new session with specified DesiredCapability is sent to the Grid Hub, which will look through all of the registered nodes to see if any of them match the specification given by the test. If no node matches the specification, a CapabilityNotPresentOnTheGridException will be returned.

It is a common misconception that the PLATFORM determines the ability to choose the Operating System on which the new session will be created. In this situation, platform and operating system are not the same, thus specifying the platform to "Windows 2003 Server" will not allow you to choose between a Windows XP, Vista, and 2003 server. This misconception can be born from platforms such as Mac OSX and Linux, where the name of the platform matches the name of the Operating System.

In case of Selenium Grid, platform refers to the underlying interactions between the Driver Atoms and the web browser. Mac OSX and Linux based Operating Systems (Centos, Ubuntu, Debian, etc..) have a relatively stable communication with the web browsers such as Firefox and Chrome. Thus the platform names are simple to understand, as seen in the example bellow:

```
capability.setPlatform(Platform.MAC); //Set platform to OSX
capability.setPlatform(Platform.LINUX); // Set platform to Linux based systems
```

The prior to release of Vista, Windows based Operating Systems only had one platform, shown here:

```
capability.setPlatform(Platform.WINDOWS); //Set platform to Windows
```

However, with the introduction of UAC in Windows Vista, there were major changes done to the underlying interactions between WebDriver and Internet Explorer. To work around the UAC constraints a new platform was added to nodes with Windows based Operating systems:

```
capability.setPlatform(Platform.VISTA); //Set platform to VISTA
```

With the release of Windows 8, another major overhaul happened in how the WebDriver communicates with Internet Explorer, thus a new platform was added for Windows 8 based nodes:

```
capability.setPlatform(Platform.WIN8); //Set platform to Windows 8
```

Similar story happened with introduction of Windows 8.1, in this example the platform is set to Windows 8.1:

```
capability.setPlatform(Platform.WIN8_1); //Set platform to Windows 8.1
```

Operating System Platforms

The following list demonstrates some of the Operating Systems, and what Platform they are part of:

MAC**All OSX Operating Systems** LINUX Centos Ubuntu **UNIXSolarisBSD** XP Windows Server
2003 Windows XP Windows NT **VISTAWindows VistaWindows 2008 Server****Windows 7**
WIN8 Windows 2012 Server Windows 8 **WIN8_1****Windows 8.1**

Families

Different platforms are grouped into “Families” of platform. For example, Win8 and XP platforms are a part of the WINDOWS family. Similarly ANDROID and LINUX are part of the UNIX family.

Choosing Platform and Platform Family

When setting a platform on the [\[DesiredCapabilities\]](#) object, we can set an individual platform or family of platforms. For example:

```
capability.setPlatform(Platform.VISTA); //Will return a node with Windows Vista or 2  
capability.setPlatform(Platform.XP); //Will return a node with Windows XP or 2003  
capability.setPlatform(Platform.WINDOWS); //Will return a node with ANY Windows Oper
```

More Information

For more information on the latest platforms, please view this file:

[org.openqa.selenium.Platform.java](#)

7.2.5 - History of Grid Platforms

Information for working with platform names in Grid 2.

This documentation previously located [on the wiki](#)

You can read more about [Grid 2](#)

Selenium Grid Platforms

This section describes the PLATFORM option used in configuring Selenium Grid Nodes and [\[DesiredCapabilities\]](#) object.

History of Platforms

When requesting a new WebDriver session from the Grid, user can specify the [\[DesiredCapabilities\]](#) of the remote browser. Things such as the browser name, version, and platform are among the list of options that can be specified by the test. Specifying desired.

The following code demonstrates the DesiredCapability of Internet Explorer, version 9, on Windows XP platform:

```
[[DesiredCapabilities]] capability = DesiredCapabilities.internetExplorer();
capability.setVersion("8");
capability.setPlatform(Platform.XP);
WebDriver driver = new RemoteWebDriver(new URL("http://localhost:4444/wd/hub"), capa
```

The request for a new session with specified DesiredCapability is sent to the Grid Hub, which will look through all of the registered nodes to see if any of them match the specification given by the test. If no node matches the specification, a CapabilityNotPresentOnTheGridException will be returned.

It is a common misconception that the PLATFORM determines the ability to choose the Operating System on which the new session will be created. In this situation, platform and operating system are not the same, thus specifying the platform to “Windows 2003 Server” will not allow you to choose between a Windows XP, Vista, and 2003 server. This misconception can be born from platforms such as Mac OSX and Linux, where the name of the platform matches the name of the Operating System.

In case of Selenium Grid, platform refers to the underlying interactions between the Driver Atoms and the web browser. Mac OSX and Linux based Operating Systems (Centos, Ubuntu, Debian, etc..) have a relatively stable communication with the web browsers such as Firefox and Chrome. Thus the platform names are simple to understand, as seen in the example bellow:

```
capability.setPlatform(Platform.MAC); //Set platform to OSX
capability.setPlatform(Platform.LINUX); // Set platform to Linux based systems
```

The prior to release of Vista, Windows based Operating Systems only had one platform, shown here:

```
capability.setPlatform(Platform.WINDOWS); //Set platform to Windows
```

However, with the introduction of UAC in Windows Vista, there were major changes done to the underlying interactions between WebDriver and Internet Explorer. To work around the UAC constrains a new platform was added to nodes with Windows based Operating systems:

```
capability.setPlatform(Platform.VISTA); //Set platform to VISTA
```

With the release of Windows 8, another major overhaul happened in how the WebDriver communicates with Internet Explorer, thus a new platform was added for Windows 8 based nodes:

```
capability.setPlatform(Platform.WIN8); //Set platform to Windows 8
```

Similar story happened with introduction of Windows 8.1, in this example the platform is set to Windows 8.1:

```
capability.setPlatform(Platform.WIN8_1); //Set platform to Windows 8.1
```

Operating System Platforms

The following list demonstrates some of the Operating Systems, and what Platform they are part of:

MAC**All OSX Operating Systems** LINUX Centos Ubuntu **UNIXSolarisBSD** XP Windows Server
2003 Windows XP Windows NT **VISTAWindows VistaWindows 2008 Server****Windows 7**
WIN8 Windows 2012 Server Windows 8 **WIN8_1****Windows 8.1**

Families

Different platforms are grouped into “Families” of platform. For example, Win8 and XP platforms are a part of the WINDOWS family. Similarly ANDROID and LINUX are part of the UNIX family.

Choosing Platform and Platform Family

When setting a platform on the [[DesiredCapabilities](#)] object, we can set an individual platform or family of platforms. For example:

```
capability.setPlatform(Platform.VISTA); //Will return a node with Windows Vista or 2008  
capability.setPlatform(Platform.XP); //Will return a node with Windows XP or 2003  
capability.setPlatform(Platform.WINDOWS); //Will return a node with ANY Windows Operat
```

More Information

For more information on the latest platforms, please view this file:

[org.openqa.selenium.Platform.java](#)

7.2.6 - Remote WebDriver standalone server

Working with the Standalone Server.

This documentation previously located [on the wiki](#)

The server will always run on the machine with the browser you want to test. The server can be used either from the command line or through code configuration.

Starting the server from the command line

Once you have downloaded `selenium-server-standalone-{VERSION}.jar`, place it on the computer with the browser you want to test. Then, from the directory with the jar, run the following:

```
java -jar selenium-server-standalone-{VERSION}.jar
```

Considerations for running the server

The caller is expected to terminate each session properly, calling either `selenium#stop()` or `WebDriver#quit`.

The selenium-server keeps in-memory logs for each ongoing session, which are cleared when `Selenium#stop()` or `WebDriver#quit` is called. If you forget to terminate these sessions, your server may leak memory. If you keep extremely long-running sessions, you will probably need to stop/quit every now and then (or increase memory with `-Xmx` jvm option).

Timeouts (from version 2.21)

The server has two different timeouts, which can be set as follows:

```
java -jar selenium-server-standalone-{VERSION}.jar -timeout=20 -browserTimeout=6
```

- `browserTimeout`
 - Controls how long the browser is allowed to hang (value in seconds).
- `timeout`
 - Controls how long the client is allowed to be gone before the session is reclaimed (value in seconds).

The system property `selenium.server.session.timeout` is no longer supported as of 2.21.

Please note that the `browserTimeout` is intended as a backup timeout mechanism when the ordinary timeout mechanism fails, which should be used mostly in grid/server environments to ensure that crashed/lost processes do not stay around for too long, polluting the runtime environment.

Configuring the server programmatically

In theory, the process is as simple as mapping the `DriverServlet` to a URL, but it is also possible to host the page in a lightweight container, such as Jetty, configured entirely in code.

- Download the `selenium-server.zip` and unpack.
- Put the JARs on the CLASSPATH.

- Create a new class called `AppServer`. Here, we are using Jetty, so you will need to [download](#) that as well:

```
import org.mortbay.jetty.Connector;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.nio.SelectChannelConnector;
import org.mortbay.jetty.security.SslSocketConnector;
import org.mortbay.jetty.webapp.WebAppContext;

import javax.servlet.Servlet;
import java.io.File;

import org.openqa.selenium.remote.server.DriverServlet;

public class AppServer {
    private Server server = new Server();

    public AppServer() throws Exception {
        WebAppContext context = new WebAppContext();
        context.setContextPath("");
        context.setWar(new File("."));
        server.addHandler(context);

        context.addServlet(DriverServlet.class, "/wd/*");

        SelectChannelConnector connector = new SelectChannelConnector();
        connector.setPort(3001);
        server.addConnector(connector);

        server.start();
    }
}
```

7.2.7 - Limitations of scaling up tests in Selenium 2

Summary of additional constraints that arise when running Selenium2 in parallel.

This documentation previously located [on the wiki](#)

Running parallel Selenium2

This page tries to summarize additional constraints that arise when running Selenium2 in parallel.

WebDriver instantiation

While an individual WebDriver instance cannot be shared among threads, it is easy to create multiple WebDriver instances.

Ephemeral sockets

There is a general problem of TCP/IP v4, where the TCP/IP stack uses ephemeral ports when making a connection between two sockets. The typical symptom of this is that connection failures start appearing after a short time of running, often a minute or two. The message will vary somewhat but it always appears after some time, and if you reduce the number of browsers it will eventually work fine.

[Wikipedia on Ephemeral ports](#) or a quick google of “ephemeral sockets ” will tell you what your current OS delivers and how to set it.

Currently (2.13.0) it seems like a firefox running at full blast consumes something in the range of 2000 ephemeral ports per firefox; your mileage will vary here. This means you can run out of ephemeral port on Windows XP with as little as 2 browsers, maybe even 1 if you for instance iterate extremely quickly .

Will it be fixed ?

The solution to the ephemeral socket problem is HTTP1.1 keep alive on the connections. Firefox does not support keep-alive as of version 2.13.0.

Things that are fixed

- The java client.
- Selenium server (“rc”).
- Selenium grid hub & nodes
- The ruby bindings (see notes in [RubyBindings](#)).
- The IE driver.
- ChromeDriver

The means you can use the java client to scale out to remote boxes running selenium server and never have any problems on the central build server. You may need to solve socket problems on the remote boxes though.

Microsoft Windows

If you are using the old versions of Windows (<=2003, inc XP) you should not be waiting for port usage to get low enough to fit in this space. That may simply never happen, although some combinations probably will. See <http://support.microsoft.com/kb/196271> on how to adjust it.

If you for technical reasons cannot adjust the port range on your Windows machine you will not be able to run more than 2-3 firefox browsers.

Avoiding the socket lock

Starting new browsers between each test class/test method is slow, and the socket lock also uses Ephemeral sockets, worsening the problem described above.

If you're using a suite-less test setup (like many JUnit4 users), you often start/stop the browsers in @BeforeClass/@AfterClass methods. Another option is to start the browsers in @BeforeClass and use something like JUnit/TestNG run listeners to shut down all the browsers at the end of the test run. Maven surefire supports run listeners for both JUnit and TestNG.

(TODO: Strategies to disable the socket lock and manage the ports yourself)

Native events

Due to a shared file in the native events logic, the firefox driver should probably not be using native events when running concurrently. (Watch [this issue](#)).

7.2.8 - Stealing focus from Firefox in Linux

How to work with Native Events in the Legacy Firefox extension.

This documentation previously located [on the wiki](#)

This page describes an essential component of the native events implementation on Linux - focus maintaining. In order for native events to be processed in Firefox, it must always retain focus. In case the user decides to switch to another window (a thing which could be understood), Firefox must not know it lost focus.

Solution overview

Basic idea

The basic idea is to get between the XLib (X-Windows client library) layer and the application. X-Windows notifies the application of events (user input, windows being destroyed, mouse movements) by asynchronous events. The events that indicate loss of focus [FocusOut](#) are discarded. The idea is based on Jordan Sissel's implementation of a pre-loaded library that over-rides XNextEvent - see http://www.semicomplete.com/blog/geekery/xsendevent-xdotool-and-ld_preload.html.

Extension

This simple implementation works well as long as there's one browser window. When multiple windows are involved, several challenges arise:

- Even though new windows may be opened, native events must continue to flow to the active window. However, most window managers will give focus to newly-opened windows.
- Window Switching: When wishing to switch to another window, the focus has to be moved. This requires cooperation between WebDriver's Firefox extension and this component.
- Closing windows: When a window is closed, focus must move to another window. By design, WebDriver does not guarantee anything if the active window is closed - until a new window is being switched to. In this situation, special care must be taken.

Interaction with other components

The basic idea requires no interaction with other components of WebDriver. However, when multiple windows are involved - creating, switching or destroying, this component should be aware of it. New window creation cannot be tracked - as it may happen as a side effect of many operations. Switching and closing can be tracked.

Involved technologies

To understand this solution, one should be familiar with X-Windows and its events. Knowledge of the GDK event processing loop is also useful.

Implementation Details

All of this describes the code in `firefox/src/cpp/linux-specific/x_ignore_nofocus.c`.

The shared library

Hijacking the events is done by over-riding XNextEvent. A shared library containing a modified implementation of `xNextEvent` is loaded using `LD_PRELOAD`. The modified function opens `/usr/lib/libX11.so.6` and invokes the real function. Then the event that the real function returns (i.e. the real event) is inspected.

Identifying events

Under the basic idea, `FocusOut` events will be simply discarded. However, window switching complicates matters.

Data structure

There's a global data structure that remembers the following information:

- The active window ID (if there is such one at the moment)
- The ID of a new window that's being created (again, if exists)
- If window switching is in progress.
- If window closing is in progress.
- Was the focus given to another window and should be stolen back to the active one?
- Was a `FocusIn` event already received by the active window?
- Did we set the currently active window as a result of a close operation?

Firefox starts up

`FocusIn` event arrives and the active window ID is 0. A new active window is set. Note that during the creation of the main window, another sub-window is created and a `FocusOut` event is sent to the active window. Fortunately, this `FocusOut` event indicates that the focus is going to move to a sub-window (identified by `NotifyInferior`) so it is allowed.

The user has switched to another window

This is indicated by a `FocusOut` event with a detail field that is neither `NotifyAncestor` nor `NotifyInferior`. This event is simply discarded and replaced with a `KeymapNotify` event, which is promptly discarded by GDK.

A new window is being created

This condition is identified by a `ReparentNotify` event. When this happens, the `new_window` field will be set to the ID of the newly created window. Subsequent `FocusOut` events will be allowed - during the new window creation events will flow as usual (`FocusOut` event from the active window, `FocusIn` event to the new window, `FocusOut` to the new window and `FocusIn` to a sub-window of the new window). After the sub-window of the new window receives `FocusIn`, a call to `xSetInputFocus` will be issued to return the focus to the active window.

A window switch occurs

During a window switch events will flow as normal. A window switch is considered done when the sub-window of a window receives the `FocusIn` event. A window switch starts by identifying the file `/tmp/switch_window_started`. In this file, a `switch:` string following a window ID is written (the ID is just for debugging purpose). This will change the active window ID to 0 and the state to "during switch". During a switch (or when there's no active window) no events are discarded.

A window is being closed

Very similar to window switching (also identified by reading the file). However, it is indicated that the window is being closed - in case it was closed, no focus stealing will take place. In addition, the `DestroyNotify` event is being identified to find out when the active window is being closed (explicitly by the user or implicitly by some other operation that is not an explicit call to close). In this case, the active window ID will be set to 0 as well.

Important Links

- Jordan Sissel's original [XSendEvent hack](#)
- [XLib events](#) and the [XLib programming manual](#)
- [The X programming manual / specification](#)

7.2.9 - Untrusted SSL Certificates

Details on how Selenium 2 accepted untrusted SSL certificates

This documentation previously located [on the wiki](#)

Introduction

This page details how WebDriver is able to accept untrusted SSL certificates, allowing users to test trusted sites in a testing environment, where valid certificates usually do not exist. This feature is turned on by default for all supported browsers (Currently Firefox).

Firefox

Outline of solution

Firefox has an interface for overriding invalid certificates, called nsICertOverrideService. Implement this interface as a proxy to the original service - **unless** untrusted certificates are allowed. In that case, when asked about a certificate (a call to hasMatchingOverride for an invalid certificate) - indicate it's trusted.

Implementation details

Implementing the idea is mostly straightforward - badCertListener.js is a stand-alone module, that, when loaded, registers a factory for returning an instance of the service. The interesting function is hasMatchingOverride:

```
WdCertOverrideService.prototype.hasMatchingOverride = function(
    aHostName, aPort, aCert, aOverrideBits, aIsTemporary)
```

The aOverrideBits and aIsTemporary are output arguments. This is where things get a bit tricky: There are three possible override bits:

```
ERROR_UNTRUSTED: 1,
ERROR_MISMATCH: 2,
ERROR_TIME: 4
```

It's impossible to just set them all, since Firefox expects a perfect match between the offences generated by the certificate and the function's return value:
(security/manager/ssl/src/SSLServerCertVerification.cpp:302):

```

if (overrideService)
{
    PRBool haveOverride;
    PRBool isTemporaryOverride; // we don't care

    nsrv = overrideService->HasMatchingOverride(hostString, port,
                                                   ix509,
                                                   &overrideBits,
                                                   &isTemporaryOverride,
                                                   &haveOverride);

    if (NS_SUCCEEDED(nsr) && haveOverride)
    {
        // remove the errors that are already overridden
        remaining_display_errors -= overrideBits;
    }
}

if (!remaining_display_errors) {
    // all errors are covered by override rules, so let's accept the cert
    return SECSuccess;
}

```

The exact mapping of violation to error code can be easily seen at `security/manager/pki/resources/content/exceptionDialog.js` (in Firefox source):

```

var flags = 0;
if(gSSLStatus.isUntrusted)
    flags |= overrideService.ERROR_UNTRUSTED;
if(gSSLStatus.isDomainMismatch)
    flags |= overrideService.ERROR_MISMATCH;
if(gSSLStatus.isNotValidAtThisTime)
    flags |= overrideService.ERROR_TIME;

```

The SSL status can be obtained from `"@mozilla.org/security/recentbadcerts;1"` usually - However, the certificate (and its status) are added to this service only **after** the call to `hasMatchingOverride`, so there is no easy way to find out the certificate's SSLStatus. Instead, the checks have to be executed manually.

Two checks are carried out:

- Calling `nsIX509Cert.verifyForUsage`
- Comparing hostname against `nsIX509Cert.commonName`. If those are not equal, `ERROR_MISMATCH` is set.

The second check indicates whether `ERROR_MISMATCH` should be set. The first check should indicate whether `ERROR_UNTRUSTED` and `ERROR_TIME` should be set. Unfortunately, it does not work reliably when the certificate expired **and** is from an untrusted issuer. When the certificate has expired, the return code would be `CERT_EXPIRED` even if it is also untrusted. For this reason, the FirefoxDriver assumes that certificates will be untrusted - it **always** sets the `ERROR_UNTRUSTED` bit - the other two will be set only if the conditions for them are met.

This could pose a problem for someone testing a site with a valid certificate that does not match the host name it's served from (e.g. test environment serving production certificates). An additional feature for `FirefoxProfile` was added: `FirefoxProfile.setAssumeUntrustedCertificateIssuer`. Calling this function with `false` will turn the `ERROR_UNTRUSTED` bit off and allow a user to work in such situation.

HTMLUnit

Not tested yet.

IE

Not implemented yet.

Chrome

Not implemented yet.

7.2.10 - WebDriver For Mobile Browsers

Describes how Selenium 2 supported Android and iOS before Appium was created

This documentation previously located [on the wiki](#)

Introduction

We provide mobile drivers for two major mobile platforms: Android and iOS (iPhone & iPad).

They can be run on real devices and in an Android emulator or in the iOS Simulator, as appropriate. They are packaged as an app. The app needs to be installed on the emulator or device. The app embeds a [RemoteWebDriver server](#) and a light-weight HTTP server which receive, and respond to, requests from WebDriver Clients i.e. from your automated tests.

The connection between the server on the mobile platform and your tests uses an IP connection. The connection may need to be configured. For Android you can connect establish an IP connection over USB.

In some cases your existing WebDriver tests may run successfully e.g. where a common web site serves mobile and desktop users and where the UI is relatively straight-forward. However in other cases you may end up needing to create specific tests for the mobile site; particularly when the site provides specific capabilities, user interfaces, etc. for mobile browsers.

Even when a common web site serves both desktop and mobile browsers, you may want to consider writing specific tests that incorporate factors such as the screen-size of the mobile devices, and different ways users are likely to interact with your web site or web app.

Getting Started

[Android Setup](#)

[iPhone & iPad Setup](#)

Additional Mobile Platforms

There are several related opensource projects that include support for other Mobile platforms. These include:

[Blackberry WebDriver](#), for BlackBerry 5.0 and onward.

[Headless WebKit WebDriver](#). Many mobile browsers are WebKit based. Headless WebKit provides a fast light-weight solution.

These projects don't appear to be active, however they may provide a starting point for future work on these platforms.

7.2.11 - Frequently Asked Questions for Selenium 2

Items of interest for moving from Selenium 1 to Selenium 2

This documentation previously located [on the wiki](#) \

Q: What is WebDriver?

A: WebDriver is a tool for writing automated tests of websites. It aims to mimic the behaviour of a real user, and as such interacts with the HTML of the application.

Q: So, is it like [Selenium](#)? Or [Sahi](#)?

A: The aim is the same (to allow you to test your webapp), but the implementation is different. Rather than running as a Javascript application within the browser (with the limitations this brings, such as the “[same origin](#)” problem), WebDriver controls the browser itself. This means that it can take advantage of any facilities offered by the native platform.

Q: What is Selenium 2.0?

A: WebDriver is part of [Selenium](#). The main contribution that WebDriver makes is its API and the native drivers.

Q: How do I migrate from using the original Selenium APIs to the new WebDriver APIs?

A: The process is described in the Selenium documentation at
http://seleniumhq.org/docs/appendix_migrating_from_rc_to_webdriver.html

Q: Which browsers does WebDriver support?

A: The existing drivers are the ChromeDriver, InternetExplorerDriver, FirefoxDriver, OperaDriver and HtmlUnitDriver. For more information about each of these, including their relative strengths and weaknesses, please follow the links to the relevant pages. There is also support for mobile testing via the AndroidDriver, OperaMobileDriver and iPhoneDriver.

Q: What does it mean to be “developer focused”?

A: We believe that within a software application’s development team, the people who are best placed to build the tools that everyone else can use are the developers. Although it should be easy to use WebDriver directly, it should also be easy to use it as a building block for more sophisticated tools. Because of this, WebDriver has a small API that’s easy to explore by hitting the “autocomplete” button in your favourite IDE, and aims to work consistently no matter which browser implementation you use.

Q: How do I execute Javascript directly?

A: We believe that most of the time there is a requirement to execute Javascript there is a failing in the tool being used: it hasn’t emitted the correct events, has not interacted with a page correctly, or has failed to react when an XMLHttpRequest returns. We would rather fix WebDriver to work consistently and correctly than rely on testers working out which Javascript method to call.

We also realise that there will be times when this is a limitation. As a result, for those browsers that support it, you can execute Javascript by casting the WebDriver instance to a [JavascriptExecutor](#). In Java, this looks like:

```
WebDriver driver; // Assigned elsewhere
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("return document.title");
```

Other language bindings will follow a similar approach. Take a look at the [UsingJavascript](#) page for more information.

Q: Why is my Javascript execution always returning null?

A: You need to return from your javascript snippet to return a value, so:

```
js.executeScript("document.title");
```

will return null, but:

```
js.executeScript("return document.title");
```

will return the title of the document.

Q: My XPath finds elements in one browser, but not in others. Why is this?

A: The short answer is that each supported browser handles XPath slightly differently, and you're probably running into one of these differences. The long answer is on the [XpathInWebDriver](#) page.

Q: The InternetExplorerDriver does not work well on Vista. How do I get it to work as expected?

A: The InternetExplorerDriver requires that all security domains are set to the same value (either trusted or untrusted) If you're not in a position to modify the security domains, then you can override the check like this:

```
DesiredCapabilities capabilities = DesiredCapabilities.internetExplorer();
capabilities.setCapability(InternetExplorerDriver.INTRODUCE_FLAKINESS_BY_IGNORING_SECURITY_DOMAINS, true);
WebDriver driver = new InternetExplorerDriver(capabilities);
```

As can be told by the name of the constant, this may introduce flakiness in your tests. If all sites are in the same protection domain, you *should* be okay.

Q: What about support for languages other than Java?

A: Python, Ruby, C# and Java are all supported directly by the development team. There are also webdriver implementations for PHP and Perl. Support for a pure JS API is also planned.

Q: How do I handle pop up windows?

A: WebDriver offers the ability to cope with multiple windows. This is done by using the "WebDriver.switchTo().window()" method to switch to a window with a known name. If the name is not known, you can use "WebDriver.getWindowHandles()" to obtain a list of known windows. You may pass the handle to "switchTo().window()".

Q: Does WebDriver support Javascript alerts and prompts?

A: Yes, using the [Alerts API](#):

```
// Get a handle to the open alert, prompt or confirmation
Alert alert = driver.switchTo().alert();
// Get the text of the alert or prompt
alert.getText();
// And acknowledge the alert (equivalent to clicking "OK")
alert.accept();
```

Q: Does WebDriver support file uploads?

A: Yes.

You can't interact with the native OS file browser dialog directly, but we do some magic so that if you call WebElement#sendKeys("/path/to/file") on a file upload element, it does the right thing. Make sure you don't WebElement#click() the file upload element, or the browser will probably hang.

Handy hint: You can't interact with hidden elements without making them un-hidden. If your element is hidden, it can probably be un-hidden with some code like:

```
((JavascriptExecutor)driver).executeScript("arguments[0].style.visibility = 'visible'; argun
```

Q: The “onchange” event doesn’t fire after a call “sendKeys”

A: WebDriver leaves the focus in the element you called “sendKeys” on. The “onchange” event will only fire when focus leaves that element. As such, you need to move the focus, perhaps using a “click” on another element.

Q: Can I run multiple instances of the WebDriver sub-classes?

A: Each instance of an HtmlUnitDriver, ChromeDriver and FirefoxDriver is completely independent of every other instance (in the case of firefox and chrome, each instance has its own anonymous profile it uses). Because of the way that Windows works, there should only ever be a single InternetExplorerDriver instance at one time. If you need to run more than one instance of the InternetExplorerDriver at a time, consider using the Remote!WebDriver and virtual machines.

Q: I need to use a proxy. How do I configure that?

A: Proxy configuration is done via the `org.openqa.selenium.Proxy` class like so:

```
Proxy proxy = new Proxy();
proxy.setProxyAutoconfigUrl("http://youdomain/config");

// We use firefox as an example here.
DesiredCapabilities capabilities = DesiredCapabilities.firefox();
capabilities.setCapability(CapabilityType.PROXY, proxy);

// You could use any webdriver implementation here
WebDriver driver = new FirefoxDriver(capabilities);
```

Q: How do I handle authentication with the HtmlUnitDriver?

A: When creating your instance of the HtmlUnitDriver, override the “modify WebClient” method, for example:

```
WebDriver driver = new HtmlUnitDriver() {
    protected WebClient modifyWebClient(WebClient client) {
        // This class ships with HtmlUnit itself
        DefaultCredentialsProvider creds = new DefaultCredentialsProvider();

        // Set some example credentials
        creds.addCredentials("username", "password");

        // And now add the provider to the webClient instance
        client.setCredentialsProvider(creds);

        return client;
    }
};
```

Q: Is WebDriver thread-safe?

A: WebDriver is not thread-safe. Having said that, if you can serialise access to the underlying driver instance, you can share a reference in more than one thread. This is not advisable. You /can/ on the other hand instantiate one WebDriver instance for each thread.

Q: How do I type into a contentEditable iframe?

A: Assuming that the iframe is named “foo”:

```
driver.switchTo().frame("foo");
WebElement editable = driver.switchTo().activeElement();
editable.sendKeys("Your text here");
```

Sometimes this doesn’t work, and this is because the iframe doesn’t have any content. On Firefox you can execute the following before “sendKeys”:

```
((JavascriptExecutor) driver).executeScript("document.body.innerHTML = '<br>'");
```

This is needed because the iframe has no content by default: there’s nothing to send keyboard input to. This method call inserts an empty tag, which sets everything up nicely.

Remember to switch out of the frame once you’re done (as all further interactions will be with this specific frame):

```
driver.switchTo().defaultContent();
```

Q: WebDriver fails to start Firefox on Linux due to java.net.SocketException

A: If, when running WebDriver on Linux, Firefox fails to start and the error looks like:

```
Caused by: java.net.SocketException: Invalid argument
    at java.net.PlainSocketImpl.socketBind(Native Method)
    at java.net.PlainSocketImpl.bind(PlainSocketImpl.java:365)
    at java.net.Socket.bind(Socket.java:571)
    at org.openqa.selenium.firefox.internal.SocketLock.isLockFree(SocketLock.java:99)
    at org.openqa.selenium.firefox.internal.SocketLock.lock(SocketLock.java:63)
```

It may be caused due to IPv6 settings on the machine. Execute:

```
sudo sysctl net.ipv6.bindv6only=0
```

To get the socket to bind both to IPv6 and IPv4 addresses of the host with the same calls. More permanent solution is disabling this behaviour by editing /etc/sysctl.d/bindv6only.conf

Q: WebDriver fails to find elements / Does not block on page loads

A: This problem can manifest itself in various ways:

- Using WebDriver.findElement(...) throws ElementNotFoundException, but the element is clearly there - inspecting the DOM (using Firebug, etc) clearly shows it.
- Calling Driver.get returns once the HTML has been loaded - but Javascript code triggered by the onload event was not done, so the page is incomplete and some elements cannot be found.
- Clicking on an element / link triggers an operation that creates new element. However, calling findElement(s) after click returns does not find it. Isn't click supposed to be blocking?
- How do I know when a page has finished loading?

Explanation: WebDriver has a blocking API, generally. However, under some conditions it is possible for a get call to return before the page has finished loading. The classic example is Javascript starting to run after the page has loaded (triggered by onload). Browsers (e.g. Firefox) will notify WebDriver when the basic HTML content has been loaded, which is when WebDriver returns. It's difficult (if not impossible) to know when Javascript has finished executing, since JS code may schedule functions to be called in the future, depend on server response, etc. This is also true for clicking - when the platform supports native events (Windows, Linux) clicking is done by sending a mouse click event with the element's coordinates at the OS level - WebDriver cannot track the exact sequence of operations this click creates. For this reason, the blocking API is imperfect - WebDriver cannot wait for all conditions to be met before the test proceeds because it does not know them. Usually, the important matter is whether the element involved in the next interaction is present and ready.

Solution: Use the Wait class to wait for a specific element to appear. This class simply calls findElement over and over, discarding the NoSuchElementException each time, until the element is found (or a timeout has expired). Since this is the behaviour desired by default for many users, a mechanism for implicitly-waiting for elements to appear has been implemented. This is accessible through the [WebDriver.manage\(\).timeouts\(\)](#) call. (This was previously tracked on issue [26](#)).

Q: How can I trigger arbitrary events on the page?

A: WebDriver aims to emulate user interaction - so the API reflects the ways a user can interact with various elements.

Triggering a specific event cannot be achieved directly using the API, but one can use the Javascript execution abilities to call methods on an element.

Q: Why is it not possible to interact with hidden elements?

A: Since a user cannot read text in a hidden element, WebDriver will not allow access to it as well.

However, it is possible to use Javascript execution abilities to call getText directly from the element:

```
WebElement element = ...;
((JavascriptExecutor) driver).executeScript("return arguments[0].getText();", element);
```

Q: How do I start Firefox with an extension installed?

A:

```
FirefoxProfile profile = new FirefoxProfile()
profile.addExtension(...);

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

Q: I'd like it if WebDriver did....

A: If there's something that you'd like WebDriver to do, or you've found a bug, then please add an [add an issue](#) to the WebDriver project page.

Q: Selenium server sometimes takes a long time to start a new session ?

A: If you're running on linux, you will need to increase the amount of entropy available for secure random number generation. Most linux distros can install a package called "randomsound" to do this.

On Windows (XP), you may be running into http://bugs.sun.com/view_bug.do?bug_id=6705872, which usually means clearing out a lot of files from your temp directory. temp directory.

Q: What's the Selenium WebDriver API equivalent to **TextPresent ?**

A:

```
driver.findElement(By.tagName("body")).getText()
```

will get you the text of the page. To verifyTextPresent/assertTextPresent, from that you can use your favourite test framework to assert on the text. To waitForTextPresent, you may want to investigate the [WebDriverWait](#) class.

Q: The socket lock seems like a bad design. I can make it better

A: the socket lock that guards the starting of firefox is constructed with the following design constraints:

- It is shared among all the language bindings; ruby, java and any of the other bindings can coexist at the same time on the same machine.
- Certain critical parts of starting firefox must be exclusive-locked on the machine in question.
- The socket lock itself is not the primary bottleneck, starting firefox is.

The SocketLock is an implementation of the Lock interface. This allows for a pluggable strategy for your own implementation of it. To switch to a different implementation, subclass the FirefoxDriver and override the "obtainLock" method.

Q: Why do I get a UnicodeEncodeError when I send_keys in python

A: You likely don't have a Locale set on your system. Please set a `locale LANG=en_US.UTF-8` and `LC_CTYPE="en_US.UTF-8"` for example.

7.2.12 - Selenium 2.0 Team

This is who you can blame for the Selenium 2 release.

(Previously located: <https://github.com/SeleniumHQ/selenium/wiki/The-Team>)

If you've ever wondered who to thank (or blame!) for Selenium, then you've come to the right place. This page introduces you to the contributors and shows you what they're working on.

Simon Stewart: Original WebDriver developer and leading the Selenium 2 effort. He works mainly with Java and can be seen all over the code base, patching holes and adding features. By day he works as a Software Engineer in Test at Google. By night, he hacks on the crazy fun build grammar.

Julian Harty: Dabbled with `WebDriver` since 2007 mainly finding ways to make the code real and useful by testing it, and by documenting it. Currently working at eBay to find ways to make software testing more efficient and effective. He's also involved in various [open source initiatives](#), accessibility software, and writing material. Search for "Julian Harty" in your favorite search engine to track down his public work.

Jari Bakken: Has been working on WebDriver since late 2009, developed and now maintaining all things Ruby. Lead developer of [Celerity](#) and [watir-webdriver](#) and a committer on the [Watir](#) project. Day job is as a test engineer for classified ads website [FINN.no](#), and by night I try to make use of my degree in jazz guitar.

David Burns: Has been working with Selenium 1 for about 4 years and with WebDriver since the beginning of 2010 and now maintaining the .NET and Python bindings. Senior Software Engineer in Test at [Mozilla](#) helping lead the Test Automation on Web projects from within WebQA.

Anthony Long: Has been working with Selenium since 2008, and is currently working to improve the Selenium Python bindings. Anthony is the organizer of [Quality Assurance](#) and author of numerous python modules for use in the Quality Assurance field. He has used selenium extensively as a QA Lead at [HUGE](#) and most recently and currently, at [AdMeld](#).

Jim Evans: Started working with the WebDriver and Selenium since the end of 2009, working mostly on the .NET bindings. His test automation experience includes 12 years at [Microsoft](#), and has worked for the past 7 years as a Senior QA Engineer at [Numara Software](#). When he's not hacking code, he enjoys spending time with his family and performing as a singer and songwriter.

7.3 - Selenium 3

Selenium 3 was the implementation of WebDriver without the Selenium RC Code. It has since been replaced with Selenium 4, which implements the W3C WebDriver specification.

7.3.1 - Grid 3

Selenium Grid 3 supported WebDriver without Selenium RC code. Grid 3 was completely rewritten for the new Grid 4.

You can read our documentation for more information about [Grid 4](#)

Selenium Grid is a smart proxy server that allows Selenium tests to route commands to remote web browser instances. Its aim is to provide an easy way to run tests in parallel on multiple machines.

With Selenium Grid, one server acts as the hub that routes JSON formatted test commands to one or more registered Grid nodes. Tests contact the hub to obtain access to remote browser instances. The hub has a list of registered servers that it provides access to, and allows control of these instances.

Selenium Grid allows us to run tests in parallel on multiple machines, and to manage different browser versions and browser configurations centrally (instead of in each individual test).

Selenium Grid is not a silver bullet. It solves a subset of common delegation and distribution problems, but will for example not manage your infrastructure, and might not suit your specific needs.

7.3.2 - Setting up your own Grid 3

Quick start guide for setting up Grid 3.

To use Selenium Grid, you need to maintain your own infrastructure for the nodes. As this can be a cumbersome and time intense effort, many organizations use IaaS providers such as Amazon EC2 and Google Compute to provide this infrastructure.

Other options include using providers such as Sauce Labs or Testing Bot who provide a Selenium Grid as a service in the cloud. It is certainly possible to also run nodes on your own hardware. This chapter will go into detail about the option of running your own grid, complete with its own node infrastructure.

Quick start

This example will show you how to start the Selenium 2 Grid Hub, and register both a WebDriver node and a Selenium 1 RC legacy node. We will also show you how to call the grid from Java. The hub and nodes are shown here running on the same machine, but of course you can copy the selenium-server-standalone to multiple machines.

The `selenium-server-standalone` package includes the hub, WebDriver, and legacy RC needed to run the Grid, `ant` is not required anymore. You can download the `selenium-server-standalone.jar` from <https://selenium.dev/downloads/>.

Step 1: Start the Hub

The Hub is the central point that will receive test requests and distribute them to the right nodes. The distribution is done on a capabilities basis, meaning a test requiring a set of capabilities will only be distributed to nodes offering that set or subset of capabilities.

Because a test's desired capabilities are just what the name implies, *desired*, the hub cannot guarantee that it will locate a node fully matching the requested desired capabilities set.

Open a command prompt and navigate to the directory where you copied the `selenium-server-standalone.jar` file. You start the hub by passing the `-role hub` flag to the standalone server:

```
java -jar selenium-server-standalone.jar -role hub
```

The Hub will listen to port 4444 by default. You can view the status of the hub by opening a browser window and navigating to <http://localhost:4444/grid/console>.

To change the default port, you can add the optional `-port` flag with an integer representing the port to listen to when you run the command. Also, all of the other options you see in the JSON config file (seen below) are possible command-line flags.

You certainly can get by with only the simple command shown above, but if you need more advanced configuration, you can also specify a JSON format config file, for convenience, to configure the hub when you start it. You can do it like so:

```
java -jar selenium-server-standalone.jar -role hub -hubConfig hubConfig.json -de
```

Below you will see an example of a `hubConfig.json` file. We will go into more detail on how to provide node configuration files in step 2.

```
{  
  "_comment" : "Configuration for Hub - hubConfig.json",  
  "capabilities": [
```

```
"host": ip,
"maxSession": 5,
"port": 4444,
"cleanupCycle": 5000,
"timeout": 300000,
"newSessionWaitTimeout": -1,
" servlets": [],
"prioritizer": null,
"capabilityMatcher": "org.openqa.grid.internal.utils.DefaultCapabilityMatcher"
"throwOnCapabilityNotPresent": true,
"nodePolling": 180000,
"platform": "WINDOWS"}
```

Step 2: Start the Nodes

Regardless of whether you want to run a grid with new WebDriver functionality, or a grid with Selenium 1 RC functionality, or both at the same time, you use the same `selenium-server-standalone.jar` file to start the nodes:

```
java -jar selenium-server-standalone.jar -role node -hub http://localhost:4444
```

If a port is not specified through the `-port` flag, a free port will be chosen. You can run multiple nodes on one machine but if you do so, you need to be aware of your systems memory resources and problems with screenshots if your tests take them.

Configuration of Node with options

As mentioned, for backwards compatibility “wd” and “rc” roles are still a valid subset of the “node” role. But those roles limit the types of remote connections to their corresponding API, while “node” allows both RC and WebDriver remote connections.

Passing JVM properties (using the `-D` flag *before the -jar argument*) on the command line as well, and these will be picked up and propagated to the nodes:

```
-Dwebdriver.chrome.driver=chromedriver.exe
```

Configuration of Node with JSON

You can also start grid nodes that are configured with a JSON configuration file

```
java -Dwebdriver.chrome.driver=chromedriver.exe -jar selenium-server-standalone.
```

And here is an example of a `nodeConfig.json` file:

```
{
  "capabilities": [
    {
      "browserName": "firefox",
      "acceptSslCerts": true,
      "javascriptEnabled": true,
      "takesScreenshot": false,
      "firefox_profile": "",
      "browser-version": "27",
      "platform": "WINDOWS",
      "maxInstances": 5,
```

```

    "firefox_binary": "",
    "cleanSession": true
},
{
  "browserName": "chrome",
  "maxInstances": 5,
  "platform": "WINDOWS",
  "webdriver.chrome.driver": "C:/Program Files (x86)/Google/Chrome/Application/chromedriver.exe"
},
{
  "browserName": "internet explorer",
  "maxInstances": 1,
  "platform": "WINDOWS",
  "webdriver.ie.driver": "C:/Program Files (x86)/Internet Explorer/iexplore.exe"
}
],
"configuration": {
  "_comment" : "Configuration for Node",
  "cleanUpCycle": 2000,
  "timeout": 30000,
  "proxy": "org.openqa.grid.selenium.proxy.WebDriverRemoteProxy",
  "port": 5555,
  "host": ip,
  "register": true,
  "hubPort": 4444,
  "maxSession": 5
}
}

```

A note about the `-host` flag

For both hub and node, if the `-host` flag is not specified, `0.0.0.0` will be used by default. This will bind to all the public (non-loopback) IPv4 interfaces of the machine. If you have a special network configuration or any component that creates extra network interfaces, it is advised to set the `-host` flag with a value that allows the hub/node to be reachable from a different machine.

Specifying the port

The default TCP/IP port used by the hub is 4444. If you need to change the port please use above mentioned configurations.

Troubleshooting

Using Log file

For advanced troubleshooting you can specify a log file to log system messages. Start Selenium GRID hub or node with `-log` argument. Please see the below example:

```
java -jar selenium-server-standalone.jar -role hub -log log.txt
```

Use your favorite text editor to open log file (log.txt in the example above) to find “ERROR” logs if you get issues.

Using `-debug` argument

Also you can use `-debug` argument to print debug logs to console. Start Selenium Grid Hub or Node with `-debug` argument. Please see the below example:

```
java -jar selenium-server-standalone.jar -role hub -debug
```

Warning

The Selenium Grid must be protected from external access using appropriate firewall permissions.

Failure to protect your Grid could result in one or more of the following occurring:

- You provide open access to your Grid infrastructure
- You allow third parties to access internal web applications and files
- You allow third parties to run custom binaries

See this blog post on [Detectify](#), which gives a good overview of how a publicly exposed Grid could be misused: [Don't Leave your Grid Wide Open](#).

Docker Selenium

[Docker](#) provides a convenient way to provision and scale Selenium Grid infrastructure in a unit known as a container. Containers are standardised units of software that contain everything required to run the desired application, including all dependencies, in a reliable and repeatable way on different machines.

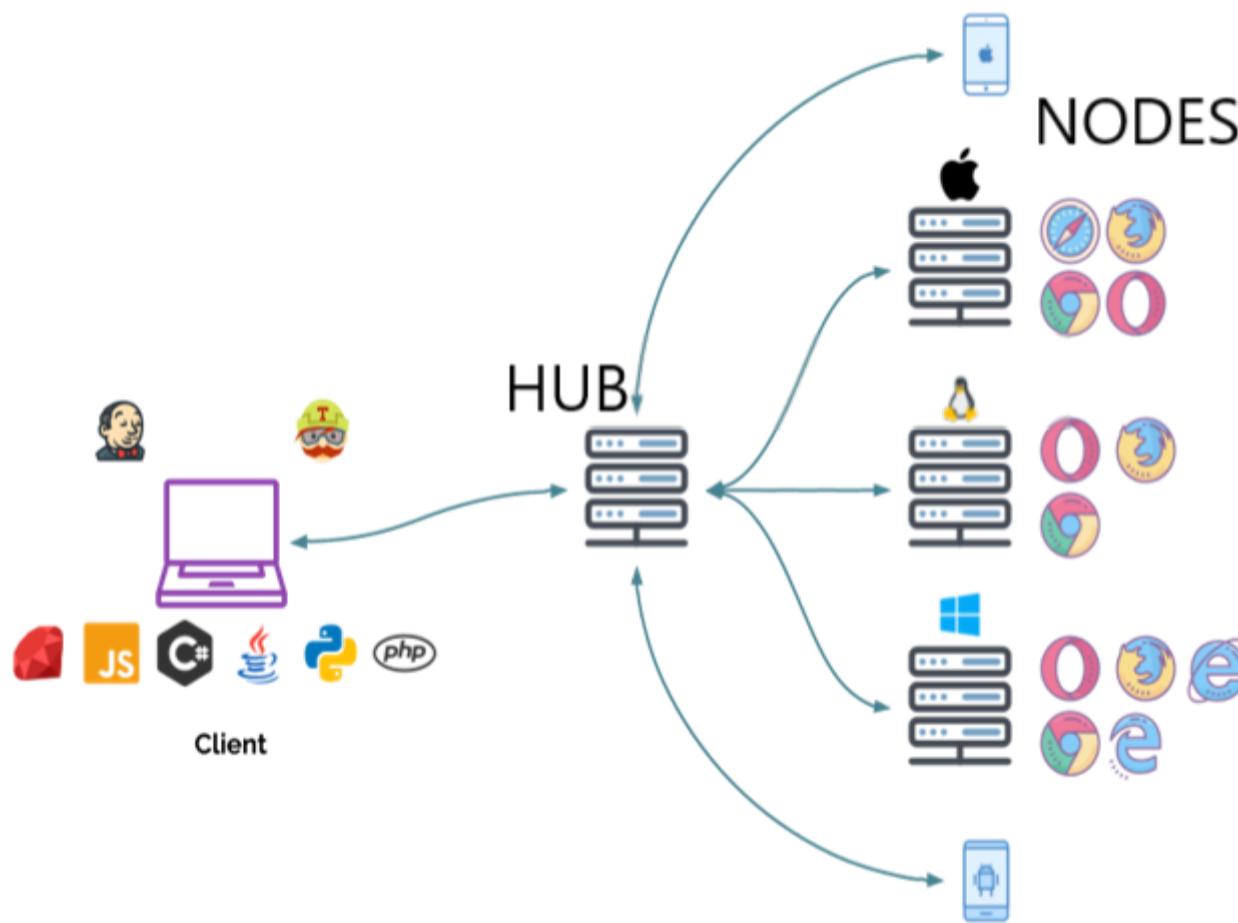
The Selenium project maintains a set of Docker images which you can download and run to get a working grid up and running quickly. Nodes are available for both Firefox and Chrome. Full details of how to provision a grid can be found within the [Docker Selenium](#) repository.

Prerequisite

The only requirement to run a Grid is to have Docker installed and working. [Install Docker](#).

7.3.3 - Components of Grid 3

Description of Hub and Nodes for Grid 3.



Hub

- Intermediary and manager
- Accepts requests to run tests
- Takes instructions from client and executes them remotely on the nodes
- Manages threads

A *Hub* is a central point where all your tests are sent. Each Selenium Grid consists of exactly one hub. The hub needs to be reachable from the respective clients (i.e. CI server, Developer machine etc.) The hub will connect one or more nodes that tests will be delegated to.

Nodes

- Where the browsers live
- Registers itself to the hub and communicates its capabilities
- Receives requests from the hub and executes them

Nodes are different Selenium instances that will execute tests on individual computer systems. There can be many nodes in a grid. The machines which are nodes do not need to be the same platform or have the same browser selection as that of the hub or the other nodes. A node on Windows might have the capability of offering Internet Explorer as a browser option, whereas this wouldn't be possible on Linux or Mac.

7.4 - Legacy Selenium IDE

Selenium IDE was the original Firefox extension for Record and Playback.

Introduction

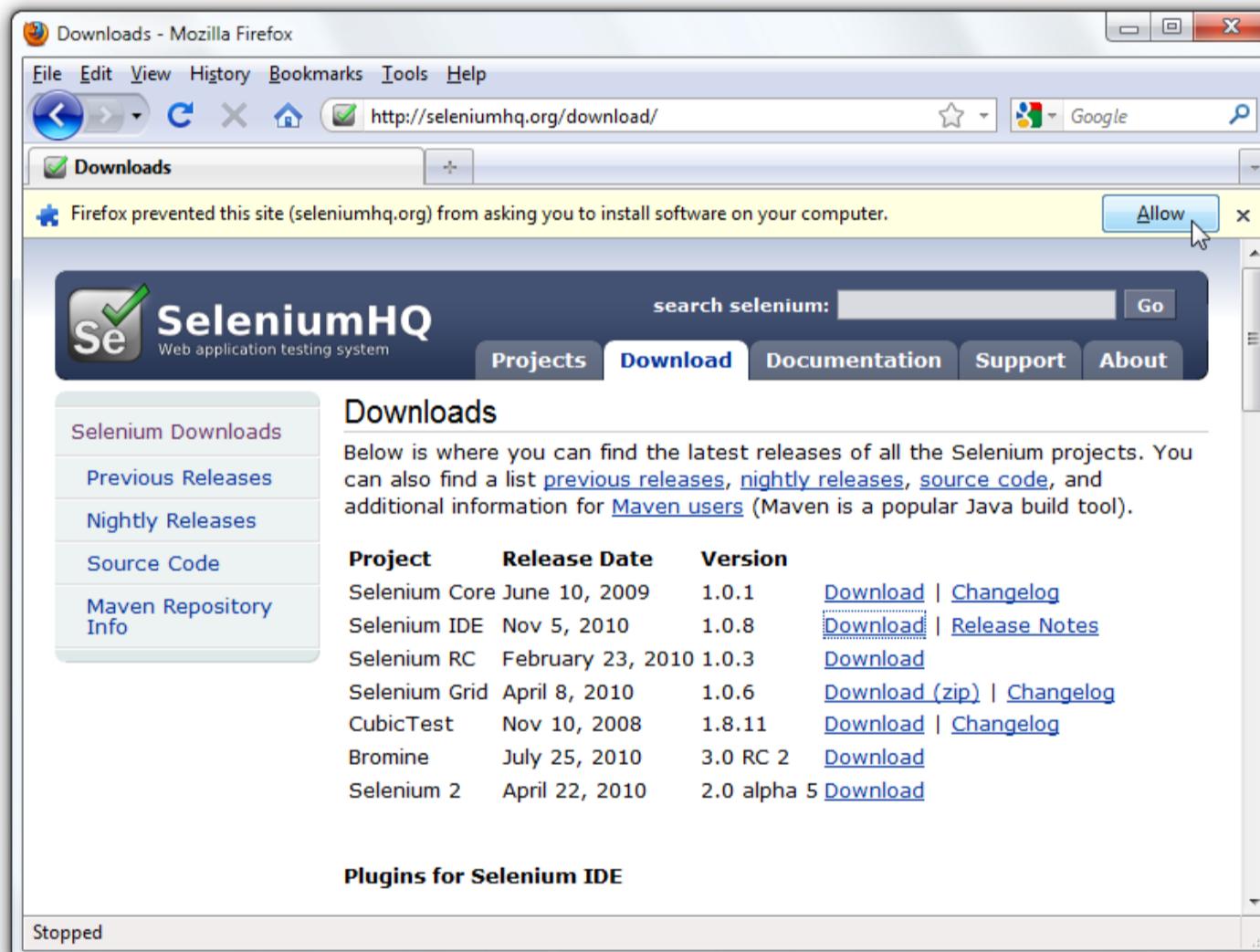
The Selenium-IDE (Integrated Development Environment) is the tool you use to develop your Selenium test cases. It's an easy-to-use Firefox plug-in and is generally the most efficient way to develop test cases. It also contains a context menu that allows you to first select a UI element from the browser's currently displayed page and then select from a list of Selenium commands with parameters pre-defined according to the context of the selected UI element. This is not only a time-saver, but also an excellent way of learning Selenium script syntax.

This chapter is all about the Selenium IDE and how to use it effectively.

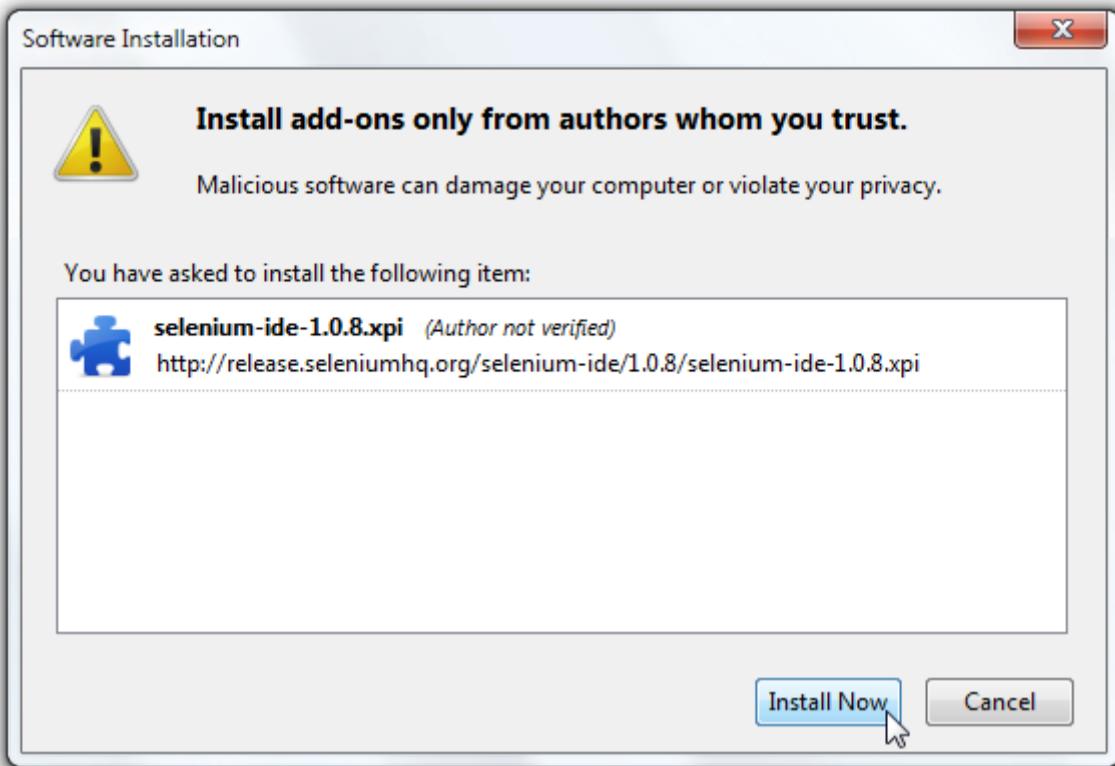
Installing the IDE

Using Firefox, first, download the IDE from the SeleniumHQ [downloads page](#)

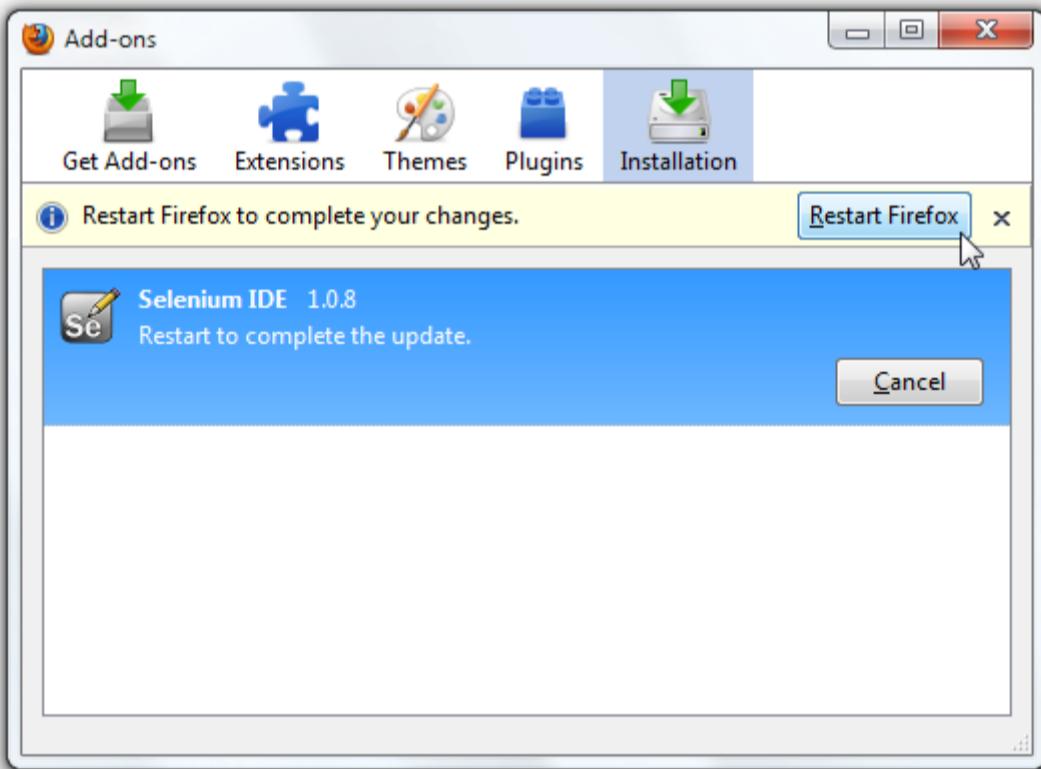
Firefox will protect you from installing addons from unfamiliar locations, so you will need to click 'Allow' to proceed with the installation, as shown in the following screenshot.



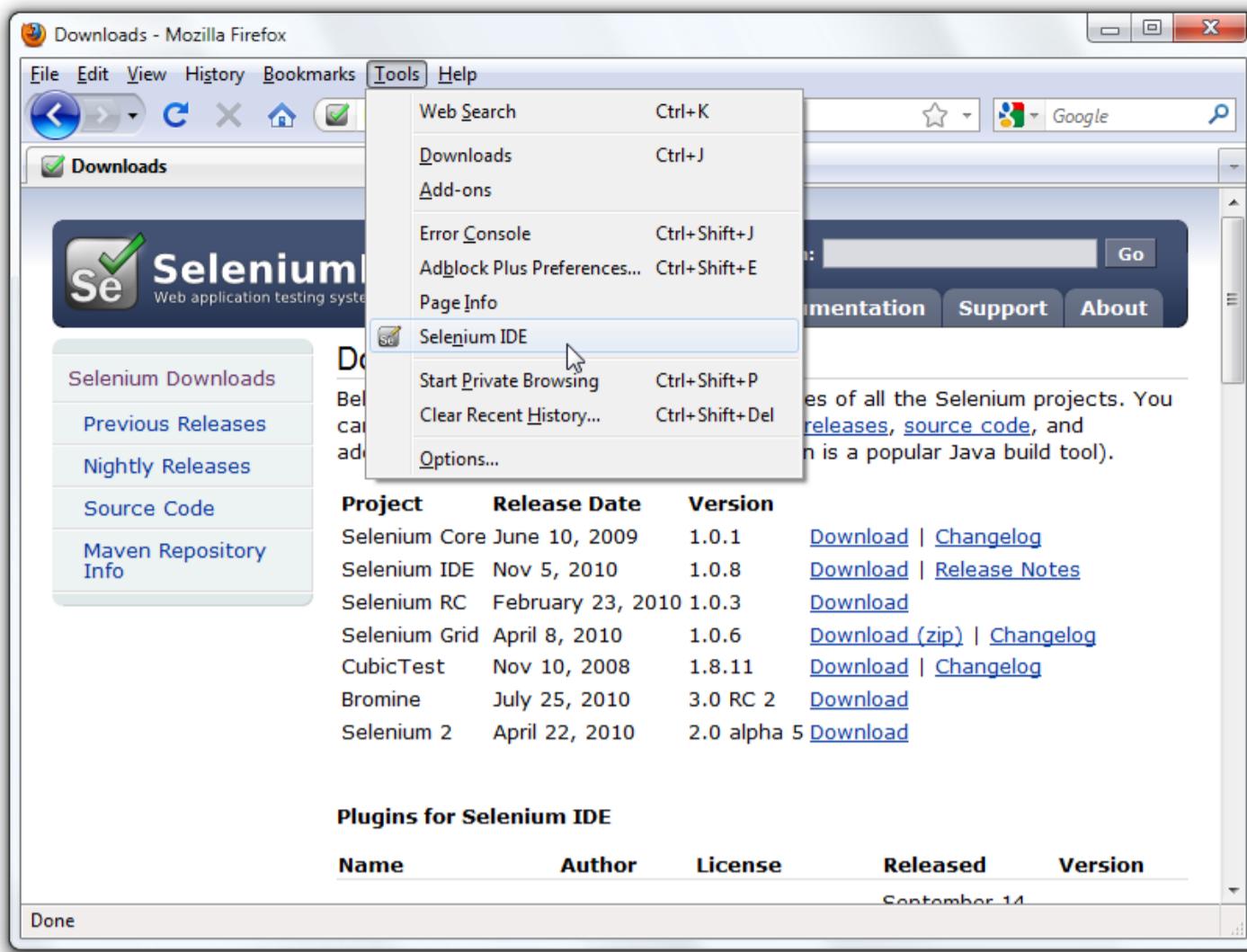
When downloading from Firefox, you'll be presented with the following window.



Select Install Now. The Firefox Add-ons window pops up, first showing a progress bar, and when the download is complete, displays the following.

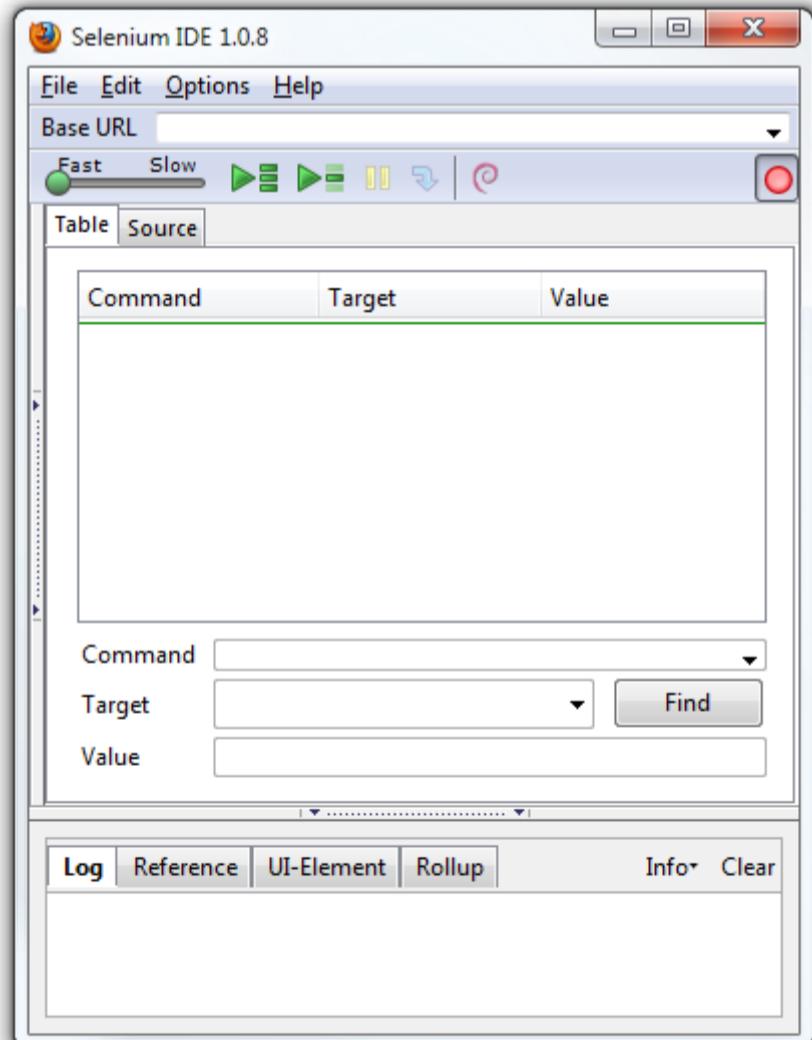


Restart Firefox. After Firefox reboots you will find the Selenium-IDE listed under the Firefox Tools menu.



Opening the IDE

To run the Selenium-IDE, simply select it from the Firefox Tools menu. It opens as follows with an empty script-editing window and a menu for loading, or creating new test cases.



IDE Features

Menu Bar

The File menu has options for Test Case and Test Suite (suite of Test Cases). Using these you can add a new Test Case, open a Test Case, save a Test Case, export Test Case in a language of your choice. You can also open the recent Test Case. All these options are also available for Test Suite.

The Edit menu allows copy, paste, delete, undo, and select all operations for editing the commands in your test case. The Options menu allows the changing of settings. You can set the timeout value for certain commands, add user-defined user extensions to the base set of Selenium commands, and specify the format (language) used when saving your test cases. The Help menu is the standard Firefox Help menu; only one item on this menu—UI-Element Documentation—pertains to Selenium-IDE.

Toolbar

The toolbar contains buttons for controlling the execution of your test cases, including a step feature for debugging your test cases. The right-most button, the one with the red-dot, is the record button.



Speed Control: controls how fast your test case runs.



Run All: Runs the entire test suite when a test suite with multiple test cases is loaded.



Run: Runs the currently selected test. When only a single test is loaded this button and the Run All button have the same effect.



Pause/Resume: Allows stopping and re-starting of a running test case.



Step: Allows you to “step” through a test case by running it one command at a time. Use for debugging test cases.



TestRunner Mode: Allows you to run the test case in a browser loaded with the Selenium-Core TestRunner. The TestRunner is not commonly used now and is likely to be deprecated. This button is for evaluating test cases for backwards compatibility with the TestRunner. Most users will probably not need this button.



Apply Rollup Rules: This advanced feature allows repetitive sequences of Selenium commands to be grouped into a single action. Detailed documentation on rollup rules can be found in the UI-Element Documentation on the Help menu.



Test Case Pane

Your script is displayed in the test case pane. It has two tabs, one for displaying the command and their parameters in a readable “table” format.

Command	Target	Value
open	/	
waitForPageToLoad		
clickAndWait	xpath=id('menu_download')/a	
assertTitle	Downloads	
verifyText	xpath=id('mainContent')/h2	Downloads

The other tab - Source displays the test case in the native format in which the file will be stored. By default, this is HTML although it can be changed to a programming language such as Java or C#, or a scripting language like Python. See the Options menu for details. The Source view also allows one to edit the test case in its raw form, including copy, cut and paste operations.

The Command, Target, and Value entry fields display the currently selected command along with its parameters. These are entry fields where you can modify the currently selected command. The first parameter specified for a command in the Reference tab of the bottom pane always goes in the Target field. If a second parameter is specified by the Reference tab, it always goes in the Value field.

A screenshot of the Selenium IDE interface. At the top left, there are three input fields: 'Command' containing 'clickAndWait', 'Target' containing 'xpath=id('menu_download')/a', and 'Value' which is empty. To the right of the Target field is a 'Find' button. Below these fields is a dropdown menu.

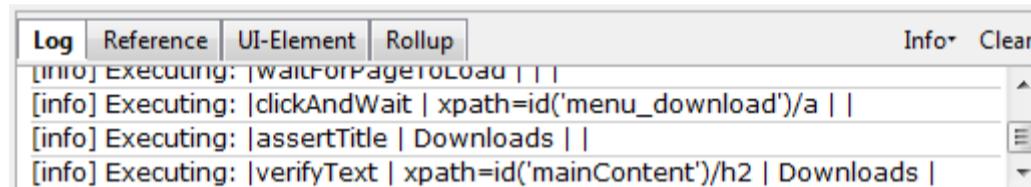
If you start typing in the Command field, a drop-down list will be populated based on the first characters you type; you can then select your desired command from the drop-down.

Log/Reference/UI-Element/Rollup Pane

The bottom pane is used for four different functions—Log, Reference, UI-Element, and Rollup—depending on which tab is selected.

Log

When you run your test case, error messages and information messages showing the progress are displayed in this pane automatically, even if you do not first select the Log tab. These messages are often useful for test case debugging. Notice the Clear button for clearing the Log. Also notice the Info button is a drop-down allowing selection of different levels of information to log.



Reference

The Reference tab is the default selection whenever you are entering or modifying Selenese commands and parameters in Table mode. In Table mode, the Reference pane will display documentation on the current command. When entering or modifying commands, whether from Table or Source mode, it is critically important to ensure that the parameters specified in the Target and Value fields match those specified in the parameter list in the Reference pane. The number of parameters provided must match the number specified, the order of parameters provided must match the order specified, and the type of parameters provided must match the type specified. If there is a mismatch in any of these three areas, the command will not run correctly.



While the Reference tab is invaluable as a quick reference, it is still often necessary to consult the Selenium Reference document.

UI-Element and Rollup

Detailed information on these two panes (which cover advanced features) can be found in the UI-Element Documentation on the Help menu of Selenium-IDE.

Building Test Cases

There are three primary methods for developing test cases. Frequently, a test developer will require all three techniques.

Recording

Many first-time users begin by recording a test case from their interactions with a website. When Selenium-IDE is first opened, the record button is ON by default. If you do not want Selenium-IDE to begin recording automatically you can turn this off by going under Options > Options... and deselecting "Start recording immediately on open."

During recording, Selenium-IDE will automatically insert commands into your test case based on your actions. Typically, this will include:

- clicking a link - click or clickAndWait commands
- entering values - type command
- selecting options from a drop-down listbox - select command
- clicking checkboxes or radio buttons - click command

Here are some "gotchas" to be aware of:

- The type command may require clicking on some other area of the web page for it to record.
- Following a link usually records a click command. You will often need to change this to clickAndWait to ensure your test case pauses until the new page is completely loaded. Otherwise, your test case will continue running commands before the page has loaded all its UI elements. This will cause unexpected test case failures.

Adding Verifications and Asserts With the Context Menu

Your test cases will also need to check the properties of a web-page. This requires assert and verify commands. We won't describe the specifics of these commands here; that is in the chapter on Selenium Commands – "Selenese". Here we'll simply describe how to add them to your test case.

With Selenium-IDE recording, go to the browser displaying your test application and right click anywhere on the page. You will see a context menu showing verify and/or assert commands.

The first time you use Selenium, there may only be one Selenium command listed. As you use the IDE however, you will find additional commands will quickly be added to this menu. Selenium-IDE will attempt to predict what command, along with the parameters, you will need for a selected UI element on the current web-page.

Let's see how this works. Open a web-page of your choosing and select a block of text on the page. A paragraph or a heading will work fine. Now, right-click the selected text. The context menu should give you a verifyTextPresent command and the suggested parameter should be the text itself.

Also, notice the Show All Available Commands menu option. This shows many, many more commands, again, along with suggested parameters, for testing your currently selected UI element.

Try a few more UI elements. Try right-clicking an image, or a user control like a button or a checkbox. You may need to use Show All Available Commands to see options other than verifyTextPresent. Once you select these other options, the more commonly used ones will show up on the primary context menu. For example, selecting verifyElementPresent for an image should later cause that command to be available on the primary context menu the next time you select an image and right-click.

Again, these commands will be explained in detail in the chapter on Selenium commands. For now though, feel free to use the IDE to record and select commands into a test case and then run it. You can learn a lot about the Selenium commands simply by experimenting with the IDE.

Editing

Insert Command

Table View

Select the point in your test case where you want to insert the command. To do this, in the Test Case Pane, left-click on the line where you want to insert a new command. Right-click and select Insert Command; the IDE will add a blank line just ahead of the line you selected. Now use the command editing text fields to enter your new command and its parameters.

Source View

Select the point in your test case where you want to insert the command. To do this, in the Test Case Pane, left-click between the commands where you want to insert a new command, and enter the HTML tags needed to create a 3-column row containing the Command, first parameter (if one is required by the Command), and second parameter (again, if one is required to locate an element) and third parameter (again, if one is required to have a value). Example:

```
<tr>
  <td>Command</td>
  <td>target (locator)</td>
  <td>Value</td>
</tr>
```

Insert Comment

Comments may be added to make your test case more readable. These comments are ignored when the test case is run.

Comments may also be used to add vertical white space (one or more blank lines) in your tests; just create empty comments. An empty command will cause an error during execution; an empty comment won't.

Table View

Select the line in your test case where you want to insert the comment. Right-click and select Insert Comment. Now use the Command field to enter the comment. Your comment will appear in purple text.

Source View

Select the point in your test case where you want to insert the comment. Add an HTML-style comment, i.e., `<!-- your comment here -->`.

Edit a Command or Comment

Table View

Simply select the line to be changed and edit it using the Command, Target, and Value fields.

Source View

Since Source view provides the equivalent of a WYSIWYG (What You See is What You Get) editor, simply modify which line you wish—command, parameter, or comment.

Opening and Saving a Test Case

Like most programs, there are Save and Open commands under the File menu. However, Selenium distinguishes between test cases and test suites. To save your Selenium-IDE tests for later use you can either save the individual test cases, or save the test suite. If the test cases of your test suite have not been saved, you'll be prompted to save them before saving the test suite.

When you open an existing test case or suite, Selenium-IDE displays its Selenium commands in the Test Case Pane.

Running Test Cases

The IDE allows many options for running your test case. You can run a test case all at once, stop and start it, run it one line at a time, run a single command you are currently developing, and you can do a batch run of an entire test suite. Execution of test cases is very flexible in the IDE.

Run a Test Case

Click the Run button to run the currently displayed test case.

Run a Test Suite

Click the Run All button to run all the test cases in the currently loaded test suite.

Stop and Start

The Pause button can be used to stop the test case while it is running. The icon of this button then changes to indicate the Resume button. To continue click Resume.

Stop in the Middle

You can set a breakpoint in the test case to cause it to stop on a particular command. This is useful for debugging your test case. To set a breakpoint, select a command, right-click, and from the context menu select Toggle Breakpoint.

Start from the Middle

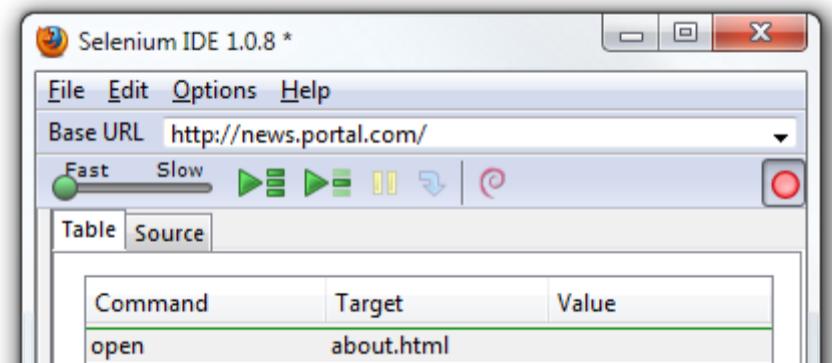
You can tell the IDE to begin running from a specific command in the middle of the test case. This also is used for debugging. To set a startpoint, select a command, right-click, and from the context menu select Set/Clear Start Point.

Run Any Single Command

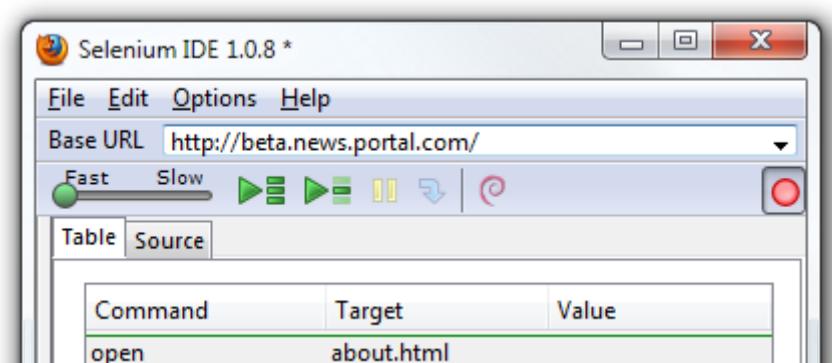
Double-click any single command to run it by itself. This is useful when writing a single command. It lets you immediately test a command you are constructing, when you are not sure if it is correct. You can double-click it to see if it runs correctly. This is also available from the context menu.

Using Base URL to Run Test Cases in Different Domains

The Base URL field at the top of the Selenium-IDE window is very useful for allowing test cases to be run across different domains. Suppose that a site named <http://news.portal.com> had an in-house beta site named <http://beta.news.portal.com>. Any test cases for these sites that begin with an open statement should specify a relative URL as the argument to open rather than an absolute URL (one starting with a protocol such as http: or https:). Selenium-IDE will then create an absolute URL by appending the open command's argument onto the end of the value of Base URL. For example, the test case below would be run against <http://news.portal.com/about.html>:



This same test case with a modified Base URL setting would be run against <http://beta.news.portal.com/about.html>:



Selenium Commands – “Selenese”

Selenium commands, often called selenese, are the set of commands that run your tests. A sequence of these commands is a test script. Here we explain those commands in detail, and we present the many choices you have in testing your web application when using Selenium.

Selenium provides a rich set of commands for fully testing your web-app in virtually any way you can imagine. The command set is often called selenese. These commands essentially create a testing language.

In selenese, one can test the existence of UI elements based on their HTML tags, test for specific content, test for broken links, input fields, selection list options, submitting forms, and table data among other things. In addition Selenium commands support testing of window size, mouse position, alerts, Ajax functionality, pop up windows, event handling, and many other web-application features. The Command Reference lists all the available commands.

A command tells Selenium what to do. Selenium commands come in three “flavors”: **Actions**, **Accessors**, and **Assertions**.

- **Actions** are commands that generally manipulate the state of the application. They do things like “click this link” and “select that option”. If an Action fails, or has an error, the execution of the current test is stopped.

Many Actions can be called with the “AndWait” suffix, e.g. “clickAndWait”. This suffix tells Selenium that the action will cause the browser to make a call to the server, and that Selenium should wait for a new page to load.

- **Accessors** examine the state of the application and store the results in variables, e.g. “storeTitle”. They are also used to automatically generate Assertions.
- **Assertions** are like Accessors, but they verify that the state of the application conforms to what is expected. Examples include “make sure the page title is X” and “verify that this checkbox is checked”.

All Selenium Assertions can be used in 3 modes: “assert”, “verify”, and ”waitFor”. For example, you can “assertText”, “verifyText” and “waitForText”. When an “assert” fails, the test is aborted. When a “verify” fails, the test will continue execution, logging the failure. This allows a single “assert” to ensure that the application is on the correct page, followed by a bunch of “verify” assertions to test form field values, labels, etc.

“waitFor” commands wait for some condition to become true (which can be useful for testing Ajax applications). They will succeed immediately if the condition is already true. However, they will fail and halt the test if the condition does not become true within the current timeout setting (see the setTimeout action below).

Script Syntax

Selenium commands are simple, they consist of the command and two parameters. For example:

```
verifyText //div//a[2] Login
```

The parameters are not always required; it depends on the command. In some cases both are required, in others one parameter is required, and in still others the command may take no parameters at all. Here are a couple more examples:

```
goBackAndWait
```

```
verifyTextPresent Welcome to My Home Page
```

type	id=phone	(555) 666-7066
type	id=address1	\${myVariableAddress}

The command reference describes the parameter requirements for each command.

Parameters vary, however they are typically:

- a locator for identifying a UI element within a page.
- a text pattern for verifying or asserting expected page content
- a text pattern or a Selenium variable for entering text in an input field or for selecting an option from an option list.

Locators, text patterns, Selenium variables, and the commands themselves are described in considerable detail in the section on Selenium Commands.

Selenium scripts that will be run from Selenium-IDE will be stored in an HTML text file format. This consists of an HTML table with three columns. The first column identifies the Selenium command, the second is a target, and the final column contains a value. The second and third columns may not require values depending on the chosen Selenium command, but they should be present. Each table row represents a new Selenium command. Here is an example of a test that opens a page, asserts the page title and then verifies some content on the page:

```
<table>
  <tr><td>open</td><td>/download/</td></tr>
  <tr><td>assertTitle</td><td></td><td>Downloads</td></tr>
  <tr><td>verifyText</td><td>//h2</td><td>Downloads</td></tr>
</table>
```

Rendered as a table in a browser this would look like the following:

open	/download/
assertTitle	Downloads
verifyText	//h2

The Selenese HTML syntax can be used to write and run tests without requiring knowledge of a programming language. With a basic knowledge of selenese and Selenium-IDE you can quickly produce and run testcases.

Test Suites

A test suite is a collection of tests. Often one will run all the tests in a test suite as one continuous batch-job.

When using Selenium-IDE, test suites also can be defined using a simple HTML file. The syntax again is simple. An HTML table defines a list of tests where each row defines the filesystem path to each test. An example tells it all.

```
<html>
<head>
<title>Test Suite Function Tests - Priority 1</title>
</head>
<body>
<table>
```

```
<tr><td><b>Suite Of Tests</b></td></tr>
<tr><td><a href=". /Login.html">Login</a></td></tr>
<tr><td><a href=". /SearchValues.html">Test Searching for Values</a></td></tr>
<tr><td><a href=". /SaveValues.html">Test Save</a></td></tr>
</table>
</body>
</html>
```

A file similar to this would allow running the tests all at once, one after another, from the Selenium-IDE.

Test suites can also be maintained when using Selenium-RC. This is done via programming and can be done a number of ways. Commonly Junit is used to maintain a test suite if one is using Selenium-RC with Java. Additionally, if C# is the chosen language, Nunit could be employed. If using an interpreted language like Python with Selenium-RC then some simple programming would be involved in setting up a test suite. Since the whole reason for using Selenium-RC is to make use of programming logic for your testing this usually isn't a problem.

Commonly Used Selenium Commands

To conclude our introduction of Selenium, we'll show you a few typical Selenium commands. These are probably the most commonly used commands for building tests.

open

opens a page using a URL.

click/clickAndWait

performs a click operation, and optionally waits for a new page to load.

verifyTitle/assertTitle

verifies an expected page title.

verifyTextPresent

verifies expected text is somewhere on the page.

verifyElementPresent

verifies an expected UI element, as defined by its HTML tag, is present on the page.

verifyText

verifies expected text and its corresponding HTML tag are present on the page.

verifyTable

verifies a table's expected contents.

waitForPageToLoad

pauses execution until an expected new page loads. Called automatically when clickAndWait is used.

waitForElementPresent

pauses execution until an expected UI element, as defined by its HTML tag, is present on the page.

Verifying Page Elements

Verifying UI elements on a web page is probably the most common feature of your automated tests. Selenese allows multiple ways of checking for UI elements. It is important that you understand these different methods because these methods define what you are actually testing.

For example, will you test that...

1. an element is present somewhere on the page?
2. specific text is somewhere on the page?
3. specific text is at a specific location on the page?

For example, if you are testing a text heading, the text and its position at the top of the page are probably relevant for your test. If, however, you are testing for the existence of an image on the home page, and the web designers frequently change the specific image file along with its position on the page, then you only want to test that an image (as opposed to the specific image file) exists somewhere on the page.

Assertion or Verification?

Choosing between “assert” and “verify” comes down to convenience and management of failures. There’s very little point checking that the first paragraph on the page is the correct one if your test has already failed when checking that the browser is displaying the expected page. If you’re not on the correct page, you’ll probably want to abort your test case so that you can investigate the cause and fix the issue(s) promptly. On the other hand, you may want to check many attributes of a page without aborting the test case on the first failure as this will allow you to review all failures on the page and take the appropriate action. Effectively an “assert” will fail the test and abort the current test case, whereas a “verify” will fail the test and continue to run the test case.

The best use of this feature is to logically group your test commands, and start each group with an “assert” followed by one or more “verify” test commands. An example follows:

Command	Target	Value
open	/download/	
assertTitle		Downloads
verifyText	//h2	Downloads
assertTable	1.2.1	Selenium IDE
verifyTable	1.2.2	June 3, 2008
verifyTable	1.2.3	1.0 beta 2

The above example first opens a page and then “asserts” that the correct page is loaded by comparing the title with the expected value. Only if this passes will the following command run and “verify” that the text is present in the expected location. The test case then “asserts” the first column in the second row of the first table contains the expected value, and only if this passed will the remaining cells in that row be “verified”.

verifyTextPresent

The command `verifyTextPresent` is used to verify specific text exists somewhere on the page. It takes a single argument—the text pattern to be verified. For example:

Command	Target	Value
verifyTextPresent	Marketing Analysis	

This would cause Selenium to search for, and verify, that the text string “Marketing Analysis” appears somewhere on the page currently being tested. Use `verifyTextPresent` when you are interested in only the text itself being present on the page. Do not use this when you also need to test where the text occurs on the page.

verifyElementPresent

Use this command when you must test for the presence of a specific UI element, rather than its content. This verification does not check the text, only the HTML tag. One common use is to check for the presence of an image.

Command	Target	Value
verifyElementPresent	//div/p/img	

This command verifies that an image, specified by the existence of an HTML tag, is present on the page, and that it follows a

tag and a tag. The first (and only) parameter is a locator for telling the Selenese command how to find the element. Locators are explained in the next section.

`verifyElementPresent` can be used to check the existence of any HTML tag within the page. You can check the existence of links, paragraphs, divisions

, etc. Here are a few more examples.

Command	Target	Value
verifyElementPresent	//div/p	
verifyElementPresent	//div/a	
verifyElementPresent	id=Login	
verifyElementPresent	link=Go to Marketing Research	
verifyElementPresent	//a[2]	
verifyElementPresent	//head/title	

These examples illustrate the variety of ways a UI element may be tested. Again, locators are explained in the next section.

verifyText

Use `verifyText` when both the text and its UI element must be tested. `verifyText` must use a locator. If you choose an *XPath* or *DOM* locator, you can verify that specific text appears at a specific location on the page relative to other UI components on the page.

Command	Target	Value
verifyText	//table/tr/td/div/p	This is my text and it occurs right after the div inside the table.

Locating Elements

For many Selenium commands, a target is required. This target identifies an element in the content of the web application, and consists of the location strategy followed by the location in the format `locatorType=location`. The locator type can be omitted in many cases. The various locator types are explained below with examples for each.

Locating by Identifier

This is probably the most common method of locating elements and is the catch-all default when no recognized locator type is used. With this strategy, the first element with the id attribute value matching the location will be used. If no element has a matching id attribute, then the first element with a name attribute matching the location will be used.

For instance, your page source could have id and name attributes as follows:

```
<html>
<body>
<form id="loginForm">
  <input name="username" type="text" />
  <input name="password" type="password" />
  <input name="continue" type="submit" value="Login" />
</form>
```

```
</body>
<html>
```

The following locator strategies would return the elements from the HTML snippet above indicated by line number:

- identifier=loginForm (3)
- identifier=password (5)
- identifier=continue (6)
- continue (6)

Since the `identifier` type of locator is the default, the `identifier=` in the first three examples above is not necessary.

Locating by Id

This type of locator is more limited than the identifier locator type, but also more explicit. Use this when you know an element's id attribute.

```
<html>
  <body>
    <form id="loginForm">
      <input name="username" type="text" />
      <input name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
      <input name="continue" type="button" value="Clear" />
    </form>
  </body>
<html>
```

- id=loginForm (3)

Locating by Name

The name locator type will locate the first element with a matching name attribute. If multiple elements have the same value for a name attribute, then you can use filters to further refine your location strategy. The default filter type is value (matching the value attribute).

```
<html>
  <body>
    <form id="loginForm">
      <input name="username" type="text" />
      <input name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
      <input name="continue" type="button" value="Clear" />
    </form>
  </body>
<html>
```

- name=username (4)
- name=continue value=Clear (7)
- name=continue Clear (7)
- name=continue type=button (7)

Note: Unlike some types of XPath and DOM locators, the three types of locators above allow Selenium to test a UI element independent of its location on the page. So if the page structure and organization is altered, the test will still pass. You may or may not want to also test whether the page structure changes. In the case where web designers frequently alter the page, but its functionality must be regression tested, testing via id and name attributes, or really via any HTML property, becomes very important.

Locating by XPath

XPath is the language used for locating nodes in an XML document. As HTML can be an implementation of XML (XHTML), Selenium users can leverage this powerful language to target elements in their web applications. XPath extends beyond (as well as supporting) the simple methods of locating by id or name attributes, and opens up all sorts of new possibilities such as locating the third checkbox on the page.

One of the main reasons for using XPath is when you don't have a suitable id or name attribute for the element you wish to locate. You can use XPath to either locate the element in absolute terms (not advised), or relative to an element that does have an id or name attribute. XPath locators can also be used to specify elements via attributes other than id and name.

Absolute XPaths contain the location of all elements from the root (html) and as a result are likely to fail with only the slightest adjustment to the application. By finding a nearby element with an id or name attribute (ideally a parent element) you can locate your target element based on the relationship. This is much less likely to change and can make your tests more robust.

Since only `xpath` locators start with “//”, it is not necessary to include the `xpath=` label when specifying an XPath locator.

```
<html>
  <body>
    <form id="loginForm">
      <input name="username" type="text" />
      <input name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
      <input name="continue" type="button" value="Clear" />
    </form>
  </body>
<html>
```

- `xpath=/html/body/form[1]` (3) - *Absolute path (would break if the HTML was changed only slightly)*
- `//form[1]` (3) - *First form element in the HTML*
- `xpath=//form[@id='loginForm']` (3) - *The form element with attribute named 'id' and the value 'loginForm'*
- `xpath=//form[input/@name='username']` (3) - *First form element with an input child element with attribute named 'name' and the value 'username'*
- `//input[@name='username']` (4) - *First input element with attribute named 'name' and the value 'username'*
- `//form[@id='loginForm']/input[1]` (4) - *First input child element of the form element with attribute named 'id' and the value 'loginForm'*
- `//input[@name='continue'][@type='button']` (7) - *Input with attribute named 'name' and the value 'continue' and attribute named 'type' and the value 'button'*
- `//form[@id='loginForm']/input[4]` (7) - *Fourth input child element of the form element with attribute named 'id' and value 'loginForm'*

These examples cover some basics, but in order to learn more, the following references are recommended:

- [W3Schools XPath Tutorial](#)
- [W3C XPath Recommendation](#)

There are also a couple of very useful Firefox Add-ons that can assist in discovering the XPath of an element:

- [XPath Checker](#) XPath and can be used to test XPath results.
- [Firebug] (<https://addons.mozilla.org/en-US/firefox/addon/1843>) - XPath suggestions are just one of the many powerful features of this very useful add-on.

Locating Hyperlinks by Link Text

This is a simple method of locating a hyperlink in your web page by using the text of the link. If two links with the same text are present, then the first match will be used.

```
<html>
  <body>
    <p>Are you sure you want to do this?</p>
    <a href="continue.html">Continue</a>
    <a href="cancel.html">Cancel</a>
  </body>
<html>
```

- link=Continue (4)
- link=Cancel (5)

Locating by DOM

The Document Object Model represents an HTML document and can be accessed using JavaScript. This location strategy takes JavaScript that evaluates to an element on the page, which can be simply the element's location using the hierarchical dotted notation.

Since only `dom` locators start with “document”, it is not necessary to include the `dom=` label when specifying a DOM locator.

```
<html>
  <body>
    <form id="loginForm">
      <input name="username" type="text" />
      <input name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
      <input name="continue" type="button" value="Clear" />
    </form>
  </body>
<html>
```

- `dom=document.getElementById('loginForm')` (3)
- `dom=document.forms['loginForm']` (3)
- `dom=document.forms[0]` (3)
- `document.forms[0].username` (4)
- `document.forms[0].elements['username']` (4)
- `document.forms[0].elements[0]` (4)
- `document.forms[0].elements[3]` (7)

You can use Selenium itself as well as other sites and extensions to explore the DOM of your web application. A good reference exists on [W3Schools](#).

Locating by CSS

CSS (Cascading Style Sheets) is a language for describing the rendering of HTML and XML documents. CSS uses Selectors for binding style properties to elements in the document. These Selectors can be used by Selenium as another locating strategy.

```
<html>
  <body>
    <form id="loginForm">
      <input class="required" name="username" type="text" />
      <input class="required passfield" name="password" type="password" />
      <input name="continue" type="submit" value="Login" />
      <input name="continue" type="button" value="Clear" />
    </form>
```

```
</body>
<html>
```

- css=form#loginForm (3)
- css=input[name="username"] (4)
- css=input.required[type="text"] (4)
- css=input.passfield (5)
- css=#loginForm input[type="button"] (7)
- css=#loginForm input:nth-child(2) (5)

For more information about CSS Selectors, the best place to go is [the W3C publication](#). You'll find additional references there.

Implicit Locators

You can choose to omit the locator type in the following situations:

- Locators without an explicitly defined locator strategy will default to using the identifier locator strategy. See [Locating by Identifier](#) .
- Locators starting with “//” will use the XPath locator strategy. See [Locating by XPath](#) .
- Locators starting with “document” will use the DOM locator strategy. See [Locating by DOM](#) .

Matching Text Patterns

Like locators, *patterns* are a type of parameter frequently required by Selenese commands. Examples of commands which require patterns are **verifyTextPresent**, **verifyTitle**, **verifyAlert**, **assertConfirmation**, **verifyText**, and **verifyPrompt**. And as has been mentioned above, link locators can utilize a pattern. Patterns allow you to *describe*, via the use of special characters, what text is expected rather than having to specify that text exactly.

There are three types of patterns: *globbing*, *regular expressions*, and *exact*.

Globbing Patterns

Most people are familiar with globbing as it is utilized in filename expansion at a DOS or Unix/Linux command line such as `ls *.c`. In this case, globbing is used to display all the files ending with a `.c` extension that exist in the current directory. Globbing is fairly limited.

Only two special characters are supported in the Selenium implementation:

* which translates to “match anything,” i.e., nothing, a single character, or many characters.

[] (*character class*) which translates to “match any single character found inside the square brackets.” A dash (hyphen) can be used as a shorthand to specify a range of characters (which are contiguous in the ASCII character set). A few examples will make the functionality of a character class clear:

`[aeiou]` matches any lowercase vowel

`[0-9]` matches any digit

`[a-zA-Z0-9]` matches any alphanumeric character

In most other contexts, globbing includes a third special character, the **?**. However, Selenium globbing patterns only support the asterisk and character class.

To specify a globbing pattern parameter for a Selenese command, you can prefix the pattern with a **glob:** label. However, because globbing patterns are the default, you can also omit the label and specify just the pattern itself.

Below is an example of two commands that use globbing patterns. The actual link text on the page being tested was “Film/Television Department”; by using a pattern rather than the exact text, the **click** command will work even if the link text is changed to “Film & Television Department” or “Film and Television Department”. The glob pattern’s asterisk will match “anything or nothing” between the word “Film” and the word “Television”.

Command Target

Value

Command	Target	Value
click	link=glob:Film*Television Department	
verifyTitle	glob:*Film*Television*	The actual title of the page reached by clicking on the link was “De Anza Film And Television Department - Menu”. By using a pattern rather than the exact text, the <code>verifyTitle</code> will pass as long as the two words “Film” and “Television” appear (in that order) anywhere in the page’s title. For example, if the page’s owner should shorten the title to just “Film & Television Department,” the test would still pass. Using a pattern for both a link and a simple test that the link worked (such as the <code>verifyTitle</code> above does) can greatly reduce the maintenance for such test cases.

Regular Expression Patterns

Regular expression patterns are the most powerful of the three types of patterns that Selenese supports. Regular expressions are also supported by most high-level programming languages, many text editors, and a host of tools, including the Linux/Unix command-line utilities **grep**, **sed**, and **awk**. In Selenese, regular expression patterns allow a user to perform many tasks that would be very difficult otherwise. For example, suppose your test needed to ensure that a particular table cell contained nothing but a number. `regexp: [0-9]+` is a simple pattern that will match a decimal number of any length.

Whereas Selenese globbing patterns support only the `*` and `[]` (character class) features, Selenese regular expression patterns offer the same wide array of special characters that exist in JavaScript. Below are a subset of those special characters:

PATTERN MATCH

.	any single character
[]	character class: any single character that appears inside the brackets
*	quantifier: 0 or more of the preceding character (or group)
+	quantifier: 1 or more of the preceding character (or group)
?	quantifier: 0 or 1 of the preceding character (or group)
{1,5}	quantifier: 1 through 5 of the preceding character (or group)
	alternation: the character/group on the left or the character/group on the right
()	grouping: often used with alternation and/or quantifier

Regular expression patterns in Selenese need to be prefixed with either `regexp:` or `regexpi:`. The former is case-sensitive; the latter is case-insensitive.

A few examples will help clarify how regular expression patterns can be used with Selenese commands. The first one uses what is probably the most commonly used regular expression pattern—`*` (“dot star”). This two-character sequence can be translated as “0 or more occurrences of any character” or more simply, “anything or nothing.” It is the equivalent of the one-character globbing pattern `*` (a single asterisk).

Command	Target	Value
click	link=glob:Film*Television Department	
verifyTitle	regexp:.*Film.*Television.*	The example above is functionally equivalent to the earlier example that used globbing patterns for this same test. The only differences are the prefix (<code>regexp:</code> instead of <code>glob:</code>) and the “anything or nothing” pattern (<code>.*</code> instead of just <code>*</code>).

The more complex example below tests that the Yahoo! Weather page for Anchorage, Alaska contains info on the sunrise time:

Command	Target	Value
open	http://weather.yahoo.com/forecast/USA0012.html	
verifyTextPresent	regexp:Sunrise: *[0-9]{1,2}:[0-9]{2} [ap]m	

Let’s examine the regular expression above one part at a time:

`Sunrise: *` The string **Sunrise:** followed by 0 or more spaces
`[0-9]{1,2}` 1 or 2 digits (for the hour of the day)
`:` The character `:` (no special characters involved)
`[0-9]{2}` 2 digits (for the minutes) followed by a space
`[ap]m` “a” or “p” followed by “m” (am or pm)

Exact Patterns

The **exact** type of Selenium pattern is of marginal usefulness. It uses no special characters at all. So, if you needed to look for an actual asterisk character (which is special for both globbing and regular expression patterns), the **exact** pattern would be one way to do that. For example, if you wanted to select an item labeled “Real *” from a dropdown, the following code might work or it might not. The asterisk in the `glob:Real *` pattern will match anything or nothing. So, if there was an earlier select option labeled “Real Numbers,” it would be the option selected rather than the “Real *” option.

Command Target Value

```
select //select glob:Real *
```

In order to ensure that the “Real *” item would be selected, the `exact:` prefix could be used to create an **exact** pattern as shown below:

Command Target Value

```
select //select exact:Real *
```

But the same effect could be achieved via escaping the asterisk in a regular expression pattern:

Command Target Value

```
select //select regexp:Real \*
```

It's rather unlikely that most testers will ever need to look for an asterisk or a set of square brackets with characters inside them (the character class for globbing patterns). Thus, globbing patterns and regular expression patterns are sufficient for the vast majority of us.

The “AndWait” Commands

The difference between a command and its *AndWait* alternative is that the regular command (e.g. `click`) will do the action and continue with the following command as fast as it can, while the *AndWait* alternative (e.g. `clickAndWait`) tells Selenium to **wait** for the page to load after the action has been done.

The *AndWait* alternative is always used when the action causes the browser to navigate to another page or reload the present one.

Be aware, if you use an *AndWait* command for an action that does not trigger a navigation/refresh, your test will fail. This happens because Selenium will reach the *AndWait*'s timeout without seeing any navigation or refresh being made, causing Selenium to raise a timeout exception.

The `waitFor` Commands in AJAX applications

In AJAX driven web applications, data is retrieved from server without refreshing the page. Using `andWait` commands will not work as the page is not actually refreshed. Pausing the test execution for a certain period of time is also not a good approach as web element might appear later or earlier than the stipulated period depending on the system's responsiveness, load or other uncontrolled factors of the moment, leading to test failures. The best approach would be to wait for the needed element in a dynamic period and then continue the execution as soon as the element is found.

This is done using `waitFor` commands, as `waitForElementPresent` or `waitForVisible`, which wait dynamically, checking for the desired condition every second and continuing to the next command in the script as soon as the condition is met.

Sequence of Evaluation and Flow Control

When a script runs, it simply runs in sequence, one command after another.

Selenese, by itself, does not support condition statements (if-else, etc.) or iteration (for, while, etc.). Many useful tests can be conducted without flow control. However, for a functional test of dynamic content, possibly involving multiple pages, programming logic is often needed.

When flow control is needed, there are three options:

- a) Run the script using Selenium-RC and a client library such as Java or PHP to utilize the programming language's flow control features.
- b) Run a small JavaScript snippet from within the script using the `storeEval` command.
- c) Install the `goto_sel_ide.js` extension .

Most testers will export the test script into a programming language file that uses the Selenium-RC API (see the Selenium-IDE chapter). However, some organizations prefer to run their scripts from Selenium-IDE whenever possible (for instance, when they have many junior-level people running tests

for them, or when programming skills are lacking). If this is your case, consider a JavaScript snippet or the `goto_sel_ide.js` extension.

Store Commands and Selenium Variables

You can use Selenium variables to store constants at the beginning of a script. Also, when combined with a data-driven test design (discussed in a later section), Selenium variables can be used to store values passed to your test program from the command-line, from another program, or from a file.

The plain `store` command is the most basic of the many store commands and can be used to simply store a constant value in a Selenium variable. It takes two parameters, the text value to be stored and a Selenium variable. Use the standard variable naming conventions of only alphanumeric characters when choosing a name for your variable.

Command	Target	Value
store		paul@mysite.org

Later in your script, you'll want to use the stored value of your variable. To access the value of a variable, enclose the variable in curly brackets (`{}`) and precede it with a dollar sign like this.

Command Target Value

`verifyText //div/p \${userName}`

A common use of variables is for storing input for an input field.

Command Target Value

`type id=login \${userName}`

Selenium variables can be used in either the first or second parameter and are interpreted by Selenium prior to any other operations performed by the command. A Selenium variable may also be used within a locator expression.

An equivalent store command exists for each verify and assert command. Here are a couple more commonly used store commands.

storeElementPresent

This corresponds to `verifyElementPresent`. It simply stores a boolean value—“true” or “false”—depending on whether the UI element is found.

storeText

`StoreText` corresponds to `verifyText`. It uses a locator to identify specific page text. The text, if found, is stored in the variable. `StoreText` can be used to extract text from the page being tested.

storeEval

This command takes a script as its first parameter. Embedding JavaScript within Selenese is covered in the next section. `StoreEval` allows the test to store the result of running the script in a variable.

JavaScript and Selenese Parameters

JavaScript can be used with two types of Selenese parameters: script and non-script (usually expressions). In most cases, you'll want to access and/or manipulate a test case variable inside the JavaScript snippet used as a Selenese parameter. All variables created in your test case are stored in a JavaScript *associative array*. An associative array has string indexes rather than sequential numeric indexes. The associative array containing your test case's variables is named **storedVars**. Whenever you wish to access or manipulate a variable within a JavaScript snippet, you must refer to it as `storedVars['yourVariableName']`.

JavaScript Usage with Script Parameters

Several Selenese commands specify a `script` parameter including `assertEval`, `verifyEval`, `storeEval`, and `waitForEval`. These parameters require no special syntax. A Selenium-IDE user would simply place a snippet of JavaScript code into the appropriate field, normally the **Target** field (because a `script` parameter is normally the first or only parameter).

The example below illustrates how a JavaScript snippet can be used to perform a simple numerical calculation:

Command	Target	Value
---------	--------	-------

Command	Target	Value
store	10	hits
storeXpathCount//blockquote		blockquotes
storeEval	storedVars['hits'].storedVars['blockquotes']	paragraphs

This next example illustrates how a JavaScript snippet can include calls to methods, in this case the JavaScript String object's `toUpperCase` method and `toLowerCase` method.

Command	Target	Value
store	Edith Wharton	name
storeEval	storedVars['name'].toUpperCase()	uc
storeEval	storedVars['name'].toLowerCase()	lc

JavaScript Usage with Non-Script Parameters

JavaScript can also be used to help generate values for parameters, even when the parameter is not specified to be of type **script**.

However, in this case, special syntax is required—the *entire* parameter value must be prefixed by `javascript{` with a trailing `}`, which encloses the JavaScript snippet, as in `javascript{*yourCodeHere*}`. Below is an example in which the `type` command's second parameter `value` is generated via JavaScript code using this special syntax:

Command	Target	Value
store	league of nations searchString	
type	q	<code>javascript{storedVars['searchString'].toUpperCase()}</code>

echo - The Selenese Print Command

Selenese has a simple command that allows you to print text to your test's output. This is useful for providing informational progress notes in your test which display on the console as your test is running. These notes also can be used to provide context within your test result reports, which can be useful for finding where a defect exists on a page in the event your test finds a problem. Finally, echo statements can be used to print the contents of Selenium variables.

Command	Target	Value
echo	Testing page footer now.	
echo	Username is \${userName}	

Alerts, Popups, and Multiple Windows

Suppose that you are testing a page that looks like this.

```
<!DOCTYPE HTML>
<html>
<head>
<script type="text/javascript">
    function output(resultText){
        document.getElementById('output').childNodes[0].nodeValue=resultText;
    }

    function show_confirm(){
        var confirmation=confirm("Chose an option.");
        if (confirmation==true){
            output("Confirmed.");
        }
        else{
            output("Rejected!");
        }
    }

    function show_alert(){
        alert("I'm blocking!");
        output("Alert is gone.");
    }

    function show_prompt(){
        var response = prompt("What's the best web QA tool?","Selenium");
    }
</script>
</head>
<body>
<div id="output"></div>
</body>
</html>
```

```

        output(response);
    }
    function open_window(windowName){
        window.open("newWindow.html",windowName);
    }
    </script>
</head>
<body>

<input type="button" id="btnConfirm" onclick="show_confirm()" value="Show co
<input type="button" id="btnAlert" onclick="show_alert()" value="Show alert"
<input type="button" id="btnPrompt" onclick="show_prompt()" value="Show prom
<a href="newWindow.html" id="lnkNewWindow" target="_blank">New Window Link</
<input type="button" id="btnNewNamelessWindow" onclick="open_window()" value
<input type="button" id="btnNewNamedWindow" onclick="open_window('Mike')" va

<br />
<span id="output">
</span>
</body>
</html>

```

The user must respond to alert/confirm boxes, as well as moving focus to newly opened popup windows. Fortunately, Selenium can cover JavaScript pop-ups.

But before we begin covering alerts/confirmations/prompts in individual detail, it is helpful to understand the commonality between them. Alerts, confirmation boxes and prompts all have variations of the following

Command	Description
assertFoo(pattern)	throws error if pattern doesn't match the text of the pop-up
assertFooPresent	throws error if pop-up is not available
assertFooNotPresent	throws error if any pop-up is present
storeFoo(variable)	stores the text of the pop-up in a variable
storeFooPresent(variable)	stores the text of the pop-up in a variable and returns true or false
When running under Selenium, JavaScript pop-ups will not appear. This is because the function calls are actually being overridden at runtime by Selenium's own JavaScript. However, just because you cannot see the pop-up doesn't mean you don't have to deal with it. To handle a pop-up, you must call its <code>assertFoo(pattern)</code> function. If you fail to assert the presence of a pop-up your next command will be blocked and you will get an error similar to the following [error] Error: There was an unexpected Confirmation! [Choose an option.]	

Alerts

Let's start with alerts because they are the simplest pop-up to handle. To begin, open the HTML sample above in a browser and click on the "Show alert" button. You'll notice that after you close the alert the text "Alert is gone." is displayed on the page. Now run through the same steps with Selenium IDE recording, and verify the text is added after you close the alert. Your test will look something like this:

Command	Target	Value
open	/	
click	btnAlert	
assertAlert	I'm blocking!	
verifyTextPresent	Alert is gone.	

You may be thinking "That's odd, I never tried to assert that alert." But this is Selenium-IDE handling and closing the alert for you. If you remove that step and replay the test you will get the following error [error] Error: There was an unexpected Alert! [I'm blocking!] . You must include an assertion of the alert to acknowledge its presence.

If you just want to assert that an alert is present but either don't know or don't care what text it contains, you can use `assertAlertPresent`. This will return true or false, with false halting the test.

Confirmations

Confirmations behave in much the same way as alerts, with `assertConfirmation` and `assertConfirmationPresent` offering the same characteristics as their alert counterparts. However, by default Selenium will select OK when a confirmation pops up. Try recording clicking on the “Show confirm box” button in the sample page, but click on the “Cancel” button in the popup, then assert the output text. Your test may look something like this:

Command	Target	Value
open	/	
click	btnConfirm	
chooseCancelOnNextConfirmation		
assertConfirmation	Choose an option.	
verifyTextPresent	Rejected	

The `chooseCancelOnNextConfirmation` function tells Selenium that all following confirmation should return false. It can be reset by calling `chooseOkOnNextConfirmation`.

You may notice that you cannot replay this test, because Selenium complains that there is an unhandled confirmation. This is because the order of events Selenium-IDE records causes the click and `chooseCancelOnNextConfirmation` to be put in the wrong order (it makes sense if you think about it, Selenium can't know that you're cancelling before you open a confirmation) Simply switch these two commands and your test will run fine.

Prompts

Prompts behave in much the same way as alerts, with `assertPrompt` and `assertPromptPresent` offering the same characteristics as their alert counterparts. By default, Selenium will wait for you to input data when the prompt pops up. Try recording clicking on the “Show prompt” button in the sample page and enter “Selenium” into the prompt. Your test may look something like this:

Command	Target	Value
open	/	
answerOnNextPrompt	Selenium!	
click	id=btnPrompt	
assertPrompt	What's the best web QA tool?	
verifyTextPresent	Selenium!	

If you choose cancel on the prompt, you may notice that `answerOnNextPrompt` will simply show a target of blank. Selenium treats cancel and a blank entry on the prompt basically as the same thing.

Debugging

Debugging means finding and fixing errors in your test case. This is a normal part of test case development.

We won't teach debugging here as most new users to Selenium will already have some basic experience with debugging. If this is new to you, we recommend you ask one of the developers in your organization.

Breakpoints and Startpoints

The Sel-IDE supports the setting of breakpoints and the ability to start and stop the running of a test case, from any point within the test case. That is, one can run up to a specific command in the middle of the test case and inspect how the test case behaves at that point. To do this, set a breakpoint on the command just before the one to be examined.

To set a breakpoint, select a command, right-click, and from the context menu select *Toggle Breakpoint*. Then click the Run button to run your test case from the beginning up to the breakpoint.

It is also sometimes useful to run a test case from somewhere in the middle to the end of the test case or up to a breakpoint that follows the starting point.

For example, suppose your test case first logs into the website and then performs a series of tests and you are trying to debug one of those tests.

However, you only need to login once, but you need to keep rerunning your tests as you are developing them. You can login once, then run your test case from a startpoint placed after the login portion of your test case. That will prevent you from having to manually logout each time you rerun your test case.

To set a startpoint, select a command, right-click, and from the context menu select *Set/Clear Start Point*. Then click the Run button to execute the test case beginning at that startpoint.

Stepping Through a Testcase

To execute a test case one command at a time (“step through” it), follow these steps:

1. Start the test case running with the Run button from the toolbar.
2. Immediately pause the executing test case with the Pause button.
3. Repeatedly select the Step button.

Find Button

The Find button is used to see which UI element on the currently displayed webpage (in the browser) is used in the currently selected Selenium command.

This is useful when building a locator for a command’s first parameter (see the section on :ref: locators <locators-section> in the Selenium Commands chapter). It can be used with any command that identifies a UI element on a webpage, i.e. *click*, *clickAndWait*, *type*, and certain *assert* and *verify* commands, among others.

From Table view, select any command that has a locator parameter. Click the Find button.

Now look on the webpage: There should be a bright green rectangle enclosing the element specified by the locator parameter.

Page Source for Debugging

Often, when debugging a test case, you simply must look at the page source (the HTML for the webpage you’re trying to test) to determine a problem. Firefox makes this easy. Simply right-click the webpage and select ‘View->Page Source’.

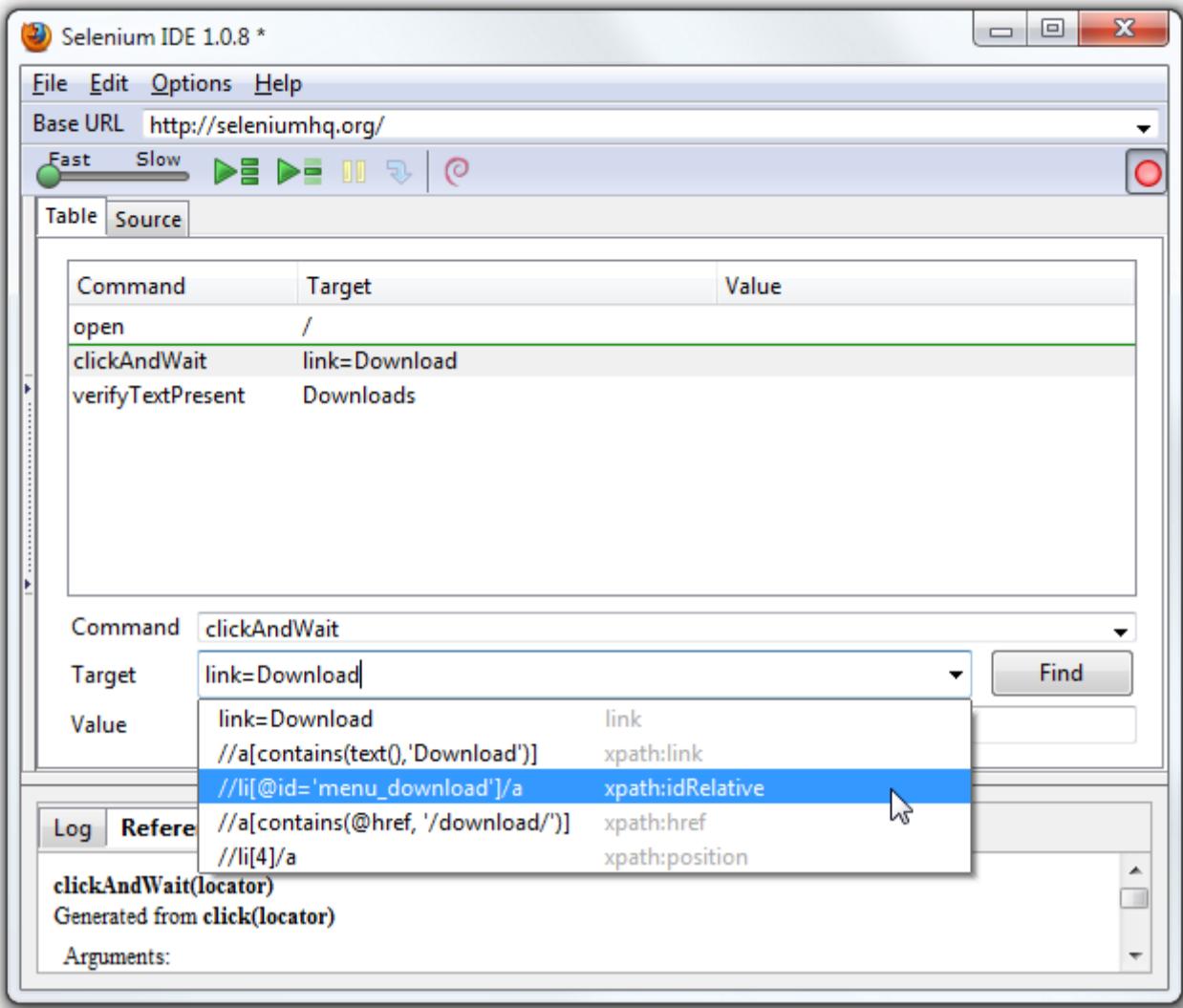
The HTML opens in a separate window. Use its Search feature (Edit=>Find) to search for a keyword to find the HTML for the UI element you’re trying to test.

Alternatively, select just that portion of the webpage for which you want to see the source. Then right-click the webpage and select View Selection Source. In this case, the separate HTML window will contain just a small amount of source, with highlighting on the portion representing your selection.

Locator Assistance

Whenever Selenium-IDE records a locator-type argument, it stores additional information which allows the user to view other possible locator-type arguments that could be used instead. This feature can be very useful for learning more about locators, and is often needed to help one build a different type of locator than the type that was recorded.

This locator assistance is presented on the Selenium-IDE window as a drop-down list accessible at the right end of the Target field (only when the Target field contains a recorded locator-type argument). Below is a snapshot showing the contents of this drop-down for one command. Note that the first column of the drop-down provides alternative locators, whereas the second column indicates the type of each alternative.



Writing a Test Suite

A test suite is a collection of test cases which is displayed in the leftmost pane in the IDE.

The test suite pane can be manually opened or closed via selecting a small dot halfway down the right edge of the pane (which is the left edge of the entire Selenium-IDE window if the pane is closed).

The test suite pane will be automatically opened when an existing test suite is opened *or* when the user selects the New Test Case item from the File menu. In the latter case, the new test case will appear immediately below the previous test case.

Selenium-IDE also supports loading pre-existing test cases by using the File -> Add Test Case menu option. This allows you to add existing test cases to a new test suite.

A test suite file is an HTML file containing a one-column table. Each cell of each row in the section contains a link to a test case. The example below is of a test suite containing four test cases:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Sample Selenium Test Suite</title>
  </head>
  <body>
    <table cellpadding="1" cellspacing="1" border="1">
      <thead>
        <tr><td>Test Cases for De Anza A-Z Directory Links</td></tr>
      </thead>
      <tbody>
        <tr><td><a href=".//a.html">A Links</a></td></tr>
        <tr><td><a href=".//b.html">B Links</a></td></tr>
        <tr><td><a href=".//c.html">C Links</a></td></tr>
        <tr><td><a href=".//d.html">D Links</a></td></tr>
      </tbody>
    </table>
  </body>
</html>
```

Note: Test case files should not have to be co-located with the test suite file that invokes them. And on Mac OS and Linux systems, that is indeed the case. However, at the time of this writing, a bug prevents Windows users from being able to place the test cases elsewhere than with the test suite that invokes them.

User Extensions

User extensions are JavaScript files that allow one to create his or her own customizations and features to add additional functionality. Often this is in the form of customized commands although this extensibility is not limited to additional commands.

There are a number of useful extensions_ created by users.

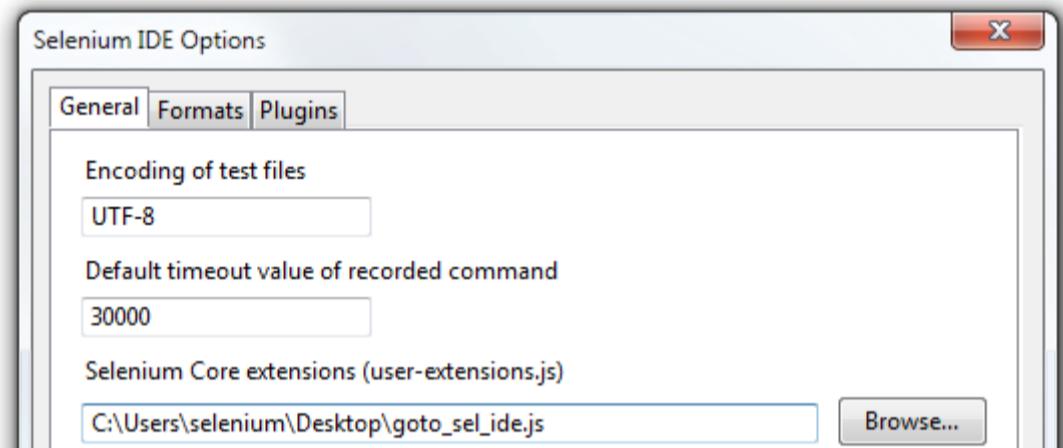
IMPORTANT: THIS SECTION IS OUT OF DATE–WE WILL BE REVISING THIS SOON.

- _extensions: <http://wiki.openqa.org/display/SEL/Contributed+User-Extensions>

.. – [goto_sel_ide.js extension](#) :

Perhaps the most popular of all Selenium-IDE extensions is one which provides flow control in the form of while loops and primitive conditionals. This extension is the `goto_sel_ide.js`. For an example of how to use the functionality provided by this extension, look at the page_ created by its author.

To install this extension, put the pathname to its location on your computer in the **Selenium Core extensions** field of Selenium-IDE's Options=>Options=>General tab.



After selecting the **OK** button, you must close and reopen Selenium-IDE in order for the extensions file to be read. Any change you make to an extension will also require you to close and reopen Selenium-IDE.

Information on writing your own extensions can be found near the bottom of the Selenium Reference_ document.

Sometimes it can prove very useful to debug step by step Selenium IDE and your User Extension. The only debugger that appears able to debug XUL/Chrome based extensions is Venkman which is supported in Firefox until version 32 included. The step by step debug has been verified to work with Firefox 32 and Selenium IDE 2.9.0.

Format

Format, under the Options menu, allows you to select a language for saving and displaying the test case. The default is HTML.

If you will be using Selenium-RC to run your test cases, this feature is used to translate your test case into a programming language. Select the language, e.g. Java, PHP, you will be using with Selenium-RC for developing your test programs. Then simply save the test case using File=>Export Test Case As. Your test case will be translated into a series of functions in the language you choose. Essentially, program code supporting your test is generated for you by Selenium-IDE.

Also, note that if the generated code does not suit your needs, you can alter it by editing a configuration file which defines the generation process.

Each supported language has configuration settings which are editable. This is under the Options=>Options=>Formats tab.

Executing Selenium-IDE Tests on Different Browsers

While Selenium-IDE can only run tests against Firefox, tests developed with Selenium-IDE can be run against other browsers, using a simple command-line interface that invokes the Selenium-RC server. This topic is covered in the :ref: Run Selenese tests <html-suite> section on Selenium-RC chapter. The `-htmlSuite` command-line option is the particular feature of interest.

Troubleshooting

Below is a list of image/explanation pairs which describe frequent sources of problems with Selenium-IDE:

Table view is not available with this format.

This message can be occasionally displayed in the Table tab when Selenium IDE is launched. The workaround is to close and reopen Selenium IDE. See [issue 1008](#), for more information. If you are able to reproduce this reliably then please provide details so that we can work on a fix.

error loading test case: no command found

You've used **File=>Open** to try to open a test suite file. Use **File=>Open Test Suite** instead.

An enhancement request has been raised to improve this error message. See [issue 1010](#).

Log	Reference	UI-Element	Rollup	Info	Clear
[info]	Executing: open /				
[info]	Executing: waitForPageToLoad				
[info]	Executing: click xpath=id('menu_download')/a				
[info]	Executing: assertTitle Downloads				
[info]	Executing: verifyText xpath=id('mainContent')/h2 Downloads				
[error]	Element xpath=id('mainContent')/h2 not found				

This type of **error** may indicate a timing problem, i.e., the element specified by a locator in your command wasn't fully loaded when the command was executed. Try putting a **pause 5000** before the command to determine whether the problem is indeed related to timing. If so, investigate using an appropriate **waitFor*** or ***AndWait** command before the failing command.

Log	Reference	UI-Element	Rollup	Info	Clear
[info]	Executing: store URL http://seleniumhq.org/				
[info]	Executing: open \${URL}				

Whenever your attempt to use variable substitution fails as is the case for the **open** command above, it indicates that you haven't actually created the variable whose value you're trying to access. This is sometimes due to putting the variable in the **Value** field when it should be in the **Target** field or vice versa. In the example above, the two parameters for the **store** command have been erroneously placed in the reverse order of what is required. For any Selenese command, the first required parameter must go in the **Target** field, and the second required parameter (if one exists) must go in the **Value** field.

*error loading test case: [Exception... “Component returned failure code: 0x80520012
(NS_ERROR_FILE_NOT_FOUND) [nsIFileInputStream.init]” nresult: “0x80520012
(NS_ERROR_FILE_NOT_FOUND)” location: “JS frame :: chrome://selenium-ide/content/file-utils.js ::
anonymous :: line 48” data: no]*

One of the test cases in your test suite cannot be found. Make sure that the test case is indeed located where the test suite indicates it is located. Also, make sure that your actual test case files have the .html extension both in their filenames, and in the test suite file where they are referenced.

An enhancement request has been raised to improve this error message. See [issue 1011](#).

Log	Reference	UI-Element	Rollup	Info	Clear
[info]	Executing:	open	/ / /		
[info]	Executing:	storeEval	3 numLinks		
[info]	Executing:	storeExpression	0 index		
[info]	Executing:	while	\${index} < \${numLinks})		
[error]	Unknown command:	'while'			

Your extension file's contents have not been read by Selenium-IDE. Be sure you have specified the proper pathname to the extensions file via **Options=>Options=>General** in the **Selenium Core extensions** field. Also, Selenium-IDE must be restarted after any change to either an extensions file *or* to the contents of the **Selenium Core extensions** field.

7.4.1 - HTML runner

Execute HTML Selenium IDE exports from command line

Selenium HTML-runner allows you to run Test Suites from a command line. Test Suites are HTML exports from Selenium IDE or compatible tools.

Common information

- Combination of releases of geckodriver / firefox / selenium-html-runner matters. There might be a software compatibility matrix somewhere.
- selenium-html-runner runs only Test Suite (not Test Case - for example an export from Monitis Transaction Monitor). Be sure you comply with this.
- For Linux users with no DISPLAY - you need to start html-runner with Virtual display (search for xvfb)

Example Linux environment

Install / download following software packages:

```
[user@localhost ~]$ cat /etc/redhat-release
CentOS Linux release 7.4.1708 (Core)

[user@localhost ~]$ rpm -qa | egrep -i "xvfb|java-1.8|firefox"
xorg-x11-server-Xvfb-1.19.3-11.el7.x86_64
firefox-52.4.0-1.el7.centos.x86_64
java-1.8.0-openjdk-1.8.0.151-1.b12.el7_4.x86_64
java-1.8.0-openjdk-headless-1.8.0.151-1.b12.el7_4.x86_64
```

Test Suite example:

```
[user@localhost ~]$ cat testsuite.html
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type" />
  <title>Test Suite</title>
</head>
<body>
  <table id="suiteTable" cellpadding="1" cellspacing="1" border="1" class="selenium">
    <tr><td><b>Test Suite</b></td></tr>
    <tr><td><a href="YOUR-TEST-SCENARIO.html">YOUR-TEST-SCENARIO</a></td></tr>
  </tbody></table>
</body>
</html>
```

How to run selenium-html-runner headless

Now, the most important part, an example of how to run the selenium-html-runner! Your experience might vary depending on software combinations - geckodriver/FF/html-runner releases.

```
xvfb-run java -Dwebdriver.gecko.driver=/home/mmasek/geckodriver.0.18.0 -jar sele
```

```
[user@localhost ~]$ xvfb-run java -Dwebdriver.gecko.driver=/home/mmasek/geckodri
Multi-window mode is longer used as an option and will be ignored.

1510061109691 geckodriver    INFO    geckodriver 0.18.0
1510061109708 geckodriver    INFO    Listening on 127.0.0.1:2885
1510061110162 geckodriver::marionette INFO    Starting browser /usr/bin/firefo
151006111084 Marionette      INFO    Listening on port 43229
1510061111187 Marionette      WARN    TLS certificate errors will be ignored f
Nov 07, 2017 1:25:12 PM org.openqa.selenium.remote.ProtocolHandshake createSessi
INFO: Detected dialect: W3C
2017-11-07 13:25:12.714:INFO::main: Logging initialized @3915ms to org.seleniumh
2017-11-07 13:25:12.804:INFO:osjs.Server:main: jetty-9.4.z-SNAPSHOT
2017-11-07 13:25:12.822:INFO:osjsh.ContextHandler:main: Started o.s.j.s.h.Contex
2017-11-07 13:25:12.843:INFO:osjs.AbstractConnector:main: Started ServerConnecto
2017-11-07 13:25:12.843:INFO:osjs.Server:main: Started @4045ms
Nov 07, 2017 1:25:13 PM org.openqa.selenium.server.htmlrunner.CoreTestCase run
INFO: |open|/auth_mellon.php|||
Nov 07, 2017 1:25:14 PM org.openqa.selenium.server.htmlrunner.CoreTestCase run
INFO: |waitForPageToLoad|3000|||
.
.
.

.etc

<td>result:</td>
<td>PASS</td>
```

7.4.2 - Legacy Selenium IDE Release Notes

Selenium IDE was the original Firefox extension for Record and Playback. Version 2.x was updated to support WebDriver.

This documentation previously located [on the wiki](#)

2.9.1 - to be released

- Fix - Fixes <https://github.com/SeleniumHQ/selenium/issues/396>
- Fix - Changed Google code links to GitHub.
- Enh - Merged official language plugins into the main xpi eliminating the need for multi-xpi with the main xpi and multiple language plugin xpis.
- Fix - Fixes <https://github.com/SeleniumHQ/selenium/issues/570>

2.9.0

- Enh - Schedule tests for automatic playback at a certain time or periodic intervals.
(<http://blog.reallysimplethoughts.com/2015/03/09/selenium-ide-scheduler-has-arrived-part-1/>)
- Enh - Allow submission of diagnostic information via a gist.
- Enh - Improved health logging, including alerts normally hidden.

2.8.0

- New - Added visual assist option to help users requiring stronger contrast in colors, turned off by default. Turn it on from the Options dialog. - Issue 7696 (on Google Code)
- New - Health Service to catch unhandled exceptions, statistics, metrics and diagnostics
- Enh - Added Search Issues menu item in Help menu to make it easier to search all issues so that we do not get so many duplicate reports of the same issue
- Fix - Fixed broken autocomplete - issue 7928 (on Google Code)
- Fix - Fixed cancelling of select button when page is reloaded - issue 7793 (on Google Code)
- Fix - Adding select button to the sidebar and reduced button size - issue 7815 (on Google Code)

2.7.0

- Fix - Fixed switching between tabs in the bottom info panel in FF32 - issue 7824 (on Google Code)
- Fix - Fixes for https://bugzilla.mozilla.org/show_bug.cgi?id=1016305
- Enh - Let comments (and commands) span the full width of the commands table
- Enh - Show the result of the test case in the log after it has been played
- Enh - Group items in the Action menu by function
- Enh - Collect more statistics about test case and suite including running time for reporting purposes
- Enh - Improved listboxes supporting drag and drop reordering
- Enh - Provide common utility function for plugin authors to deal with files
- Enh - Allow pressing tab in the command text box to accept the current autocomplete and move to the target text box
- Enh - Select an autocomplete match when typing in the command text box to speed up manual entry of commands
- Enh - Make promises implementation available via deferred.js for plugin developers
- Enh - Make simple http functions available for plugin developers

- Enh - Easier to use confirmations for internal use and for plugins
- Fix - Disable autocomplete when editing comments
- Fix - Fixed error TypeError: command.isRollup is not a function
- Fix - Fixed TypeError: debugContext.currentCommand is undefined
- Fix - Fixed TypeError: this.treebox is undefined treeView.js
- Fix - Various errors when selecting a comment (usually hidden from the user)
- Fix - Incorrect doctype in overlay
- Fix - Adding Selenium IDE item under Settings->Developer menu - issue 7268 (on Google Code)
- Fix - Ignore Firefox developer tools while recording

2.6.0

- Fix - Fixed broken autocompletion in FF31+ - issue 7645 (on Google Code)
- Fix - Fixed options validation on options reset - issue 1050 (on Google Code)
- Fix - Fixed C# code formatting for select elements

2.5.0

- Enh - Select an element for a command by clicking on the element in the browser window (<http://blog.reallysimplethoughts.com/2014/01/05/manually-adding-and-updating-element-locators-the-easy-way/>)
- Enh - Start playing a test suite from any test case (Using right click menu) - issue 1987 (on Google Code)
- Enh - Add a new test case using a keyboard shortcut (ctrl-N or cmd+N)
- Fix - Fixed delete test case through right click menu was sometimes disabled - issue 5003 (on Google Code)
- Fix - Fixed Selenium IDE icon is sometimes not visible - issue 5712 (on Google Code)
- Fix - Fixed selectWindow using a variable - issue 3270 (on Google Code)
- Some minor changes

2.4.0

- Enh - Base URL history, recent test cases and recent test suites can be cleared - issue 6135 (on Google Code)
- Enh - Special key now have shorter names (<http://blog.reallysimplethoughts.com/2013/09/25/using-special-keys-in-selenium-ide-part-2/>)
- Enh - Support for user extensions in Webdriver playback - issue 5675 (on Google Code)
- Fix - The recording of entering text in fields uses type instead of sendKeys.
- Enh - When developer tools are active, the last open test case or suite is automatically opened
- Fix - Fixed is * commands in Webdriver playback in Selenium IDE - issue 6118 (on Google Code)
- Enh - Adding ability to show commands as deprecated in Selenium IDE and smartness to show the correct alternative command
- Enh - Deprecating Selenium IDE commands * TextPresent, typeKeys, keyUp, keyDown and keyPress
- Enh - Import json library in exported Ruby Webdriver tests
- Enh - Adding support for waitFor * and waitForNot * commands in Webdriver playback - issue 5913 (on Google Code)

2.3.0

- New - Added support for HTML5 input fields recording - issue 3765 (on Google Code)
- New - Recording for sendKeys command
- Enh - Removal of deprecated * TextPresent commands from right click menu

- Fix - Dead object error in recording IDE tests - issue 4761 (on Google Code)
- Fix - Fixed could not continue in recording - issue 5820 (on Google Code)
- Enh - UTF-8 encoded user-extensions.js support - issue 1646 (on Google Code)
- New special keys support for sendKeys in Selenium IDE and webdriver playback - issue #6052 (on Google Code)
- New - Special keys support to sendKeys in all official formatters - issue 6053 (on Google Code) (<http://blog.reallysimplethoughts.com/2013/09/25/using-special-keys-in-selenium-ide-part-1/>)
- Enh - Plugin api enhancement for specifying formatter type + documentaton comments
- Fix - Invalid XPath error in Firefox 23 - issue 6055 (on Google Code)
- New - Added support for Firefox 23

2.2.0

- Fix - keyUp, keyDown, keyPress, typeKeys fixed on Firefox 22 - issue 5883 (on Google Code), issue 5884 (on Google Code)

2.1.0

- Enh - Plugin system changed (<http://blog.reallysimplethoughts.com/2013/07/07/changes-to-selenium-ide-plugin-system/>)
- New - Added support for Firefox 22 + 23 beta
- Fix - Click fixed for Firefox 22 - issue 5841 (on Google Code)

2.0.0

- New - WebDriver playback support (<http://blog.reallysimplethoughts.com/2013/02/18/webdriver-playback-in-selenium-ide-is-here/>)
- New - Added support for Firefox 19 & 20
- New - Selenium IDE icon on toolbar is added on first install

1.10.0

- New - Added support for Firefox 16 & 17
- New - Implemented formatting for alert handling commands
- Bug - Fixed options for Java 4 WebDriver formatter
- Bug - Processing locators before use in getCssCount and getXpathCount. Fixes issue 4784 (on Google Code)

1.9.1

- New - Added support for Firefox 15
- New - Added support for assertTextPresent, verifyTextPresent, waitForTextPresent, assertTextNotPresent, verifyTextNotPresent, waitForTextNotPresent commands to WebDriver formatters. (<http://blog.reallysimplethoughts.com/2012/08/26/selenium-ide-webdriver-formatters-updated-to-support-textpresent-commands/>)
- New - Added the target and value parameters in comments when the WebDriver formatters do not support the command

1.9.0

- New - Added Selenese command sendKeys (<http://blog.reallysimplethoughts.com/2012/07/19/new-selenese-command-sendkeys/>)

- New - Better naming of formatters
- New - Added support for Firefox 14

1.8.1

- New - Added support for Firefox 13

1.8.0

- New - Added support for Firefox 12

1.7.2

- Bug - Fixed regression with typing into file input fields - issue 3549 (on Google Code)

1.7.1

- Bug - Fixed regression with stored variables - issue 3520 (on Google Code)

1.7.0

- New - Added additional useful menu items to the help menu
- New - Added support for Firefox 11
- Bug - Stored variables can safely contain consecutive dollar signs - issue 834 (on Google Code)
- Bug - Don't trim whitespace when decoding HTML testcases - issue 755 (on Google Code)
- New - Formatter menu items are now context sensitive - issue 3327 (on Google Code) and issue 3385 (on Google Code)
- Bug - Fixed Ruby WebDriver test suite export - issue 3243 (on Google Code)
- Bug - File extensions being added to all file pickers - issue 3336 (on Google Code)
- Bug - Record interactions with elements with an id of 'id' - issue 3273 (on Google Code)

1.6.0

- New - Added support for Firefox 10
- New - Added keyboard shortcuts to launch Selenium IDE - issue 3028 (on Google Code)
- Bug - Added break command to autocomplete list - issue 3046 (on Google Code)
- Bug - Incorrect tooltip displayed in sidebar - issue 3098 (on Google Code)
- Bug - Improved XPath locator recording when there are multiple matches - issue 3056 (on Google Code)
- Bug - Locators can now be reordered on Mac - issue 3267 (on Google Code)

1.5.0

- New - Added support for Firefox 9
- Bug - Changes to user extensions weren't being updated in Firefox 8 - issue 2801 (on Google Code)
- Bug - Security error was thrown when trying to type into file (upload) input fields in Firefox 8 - issue 2826 (on Google Code)
- Bug - Improved French locale - issue 1912 (on Google Code)
- Bug - break command was failing - issue 725 (on Google Code)
- Bug - source view is now fixed width (monospace) - issue 522 (on Google Code)

- New - Implemented 'select' formatting for WebDriver bindings (Java, C#, Python, Ruby)
- Bug - Fixed compile-time and run-time errors in the code formatted for WebDriverBackedSelenium
- Bug - Fixed 'baseUrl' and 'get' formatting errors in various formatters to handle relative and absolute URLs

1.4.1

- Bug - Apparently I shipped without switching all the version numbers correctly. (Adam)

1.4.0

- New - Firefox 8 support (again, just a version max version bump)

1.3.0

Was going to be just a quick release to get

- New - Firefox 7 support (again, just a version max version bump)

in, but then I got busy and didn't push it when I had planned and so now

- New - Order of locators can be controlled through a panel in options.

has leaked in. Most people will want to just leave this the way it is by default. This is brand-spanking-new and allows you to do visually what you could before using a somewhat arcane bit of JS in an extension.

1.2.0

Just a quick release primarily for

- New - Firefox 6 support (which really was just changing the max version number)

But we also snuck in

- Bug - Recorded CSS locator was not W3C clean wrt attributes
- Bug - Deleting of cookies works properly if the cookie name is escaped (such as will ASP sites)
- Bug - If the cookie value has an = in it, the whole cookie is now returned instead of just up to the =

You will also notice that the bundle now only has formatters for the officially supported languages of the project (Java, C#, Python, Ruby). If anyone from the Perl, Groovy or PHP camps wants to take on ownership of those formats we'll happily help you out.

1.1.0

Hey! Look at that! A slightly more significant version bump! Any why is that? Well...

- New - WebDriver exports for Ruby, Python, C# and Java

Which are the four supported languages of the Selenium project. This also means that Se-IDE is officially deprecating inclusion of the Groovy, Perl and PHP format plugins in the main release bundle. It would be outstanding if the community around those languages picks up their development and maintenance. Read more about the WebDriver exporters on [Samit's blog](#).

Of course, format switching is still in Experimental purgatory for at least this release. Losing people's scripts because of bugs is not acceptable and we're working on it. To 'goal' is to have them back for the next release.

Also included in this release are

- New - setIndent(n) is now available to formats for greater control over formatting of export formats
- Bug - There was a performance regression in deep in some shared code that has been addressed.
- New - Rather than recording 'foo' for an element which has an id of 'foo' it is captured as 'id=foo' to be very specific as to which element would be interacted with
- New - Same with 'name'
- New - Popups (alerts, confirms, prompts) and new windows work again

1.0.12

This is a minor release with nothing too huge included. But because the last one didn't get pushed to the world, it is important to make a note of a big change introduced in 1.0.11.

We have marked the changing of formats as *Experimental* due to a couple lose-all-your-data bugs. As a result it is disabled in the toolbar by default. To enable it, click the checkbox in the Options menu. And because we **really** don't want you to lose your data, when you switch formats you will get a big warning box. This too can be disabled in the Options menu. But if you do both of these things and your script gets sent to the abyss, you have been warned. :)

Changes in this release include the following:

- New - Firefox 5 support
- New - When upgrading Se-IDE, the release notes (these) are shown on first start
- Bug - some Java format changes
- Bug - some PHP format changes
- Bug - the 'Find' button works again
- Bug - generated CSS is standards compliant
- New - dropped support for FF 3.5 or older

1.0.11

It has been half a year since our last release of 1.0.10 and we have put a lot of effort to bring you this release. The summary of the contributions to this release is as follows:-

73% (22) Samit Badle

16% (5) Adam Goucher

6% (2) Dave Hunt

3% (1) Santiago Suarez Ordoñez

3% (1) Simon Stewart

Here is the list of changes excluding some minor fixes and code refactoring.

Main Features:

- Firefox 4 support (Issue 1470 (on Google Code), Simon Stewart and Samit Badle)
- New CSS locator builder! Selenium IDE will now create locators using CSS when recording (Santiago Suarez Ordoñez)
- Added more power to the plugin developers through the new Util command builders support (Issue 442 (on Google Code), Samit Badle)
- New command getCssCount (Adam Goucher)

Usability Improvements:

- Selenium IDE is now available from the Web developer menu in Firefox 4 (Issue 1467 (on Google Code), Samit Badle)
- Camel Case search in command text box has been improved allowing you to type vTP for verifyTextPresent command (Samit Badle with Dave Hunt)
- Most actions in Selenium IDE are now accessible through the new Actions menu (Issue 1266 (on Google Code), Samit Badle and Dave Hunt)
- Removed help menu items related to Firefox from Selenium IDE help menu (Issue 1704 (on Google Code), Samit Badle)
- Less prompting when saving test suite (Issue 967 (on Google Code), Samit Badle)
- A method to Reset IDE Window is now available through the Options menu for people having trouble when switching from multiple monitors (Issue 1249 (on Google Code), Samit Badle)
- Show the name of the test case in save dialog (Issue 984 (on Google Code), Samit Badle)
- The preferences for the current format will be automatically shown in options dialog (Samit Badle)
- The plugins pane in the Options dialog now has a splitter (Samit Badle)
- Default Timeout Value field in the Options dialog now mentions a unit (Issue 896 (on Google Code), Adam Goucher)
- Introduced experimental features option to hide some unstable features (Samit Badle)

Bug Fixes:

- Format changing is now marked as experimental due to possible issues, you can turn it on from the options dialog (Samit Badle)
- Fixed the header issue on saving test case in another format (Issue 1164 (on Google Code), Samit Badle)
- Improved alert on changing the format (Issue 1244 (on Google Code), Samit Badle)
- Find button is back on Macs and uses a new way to highlight (Issue 1052 (on Google Code), Samit Badle)
- Recording is possible in the middle of a script again (Issue 968 (on Google Code), Samit Badle)
- Fixed the annoying skip over one command when recording in the middle of the script (Issue 745 (on Google Code), Samit Badle)
- While recording, “clickAndWait” command becomes “click” is now fixed (Issue 419 (on Google Code), Samit Badle)
- Selenium IDE bottom pane folding now works correctly (Issue 614 (on Google Code), Samit Badle)
- Changed the ID of Selenium IDE menu from generic name to avoid clashes with other addons. (Issue 969 (on Google Code), Samit Badle)

Improvements/Fixes Related to Formatters:

- Fixed support for stored variables in PHP formatter (Issue 970 (on Google Code), Samit Badle)
- Allow formatters to customise how set * is handled (Adam Goucher)
- Some bug fixes in PHP formatter (Issue 1281 (on Google Code), Adam Goucher)
- Number type fix (Jeremy Herault)
- New Java formatter: Webdriver backed Junit 4 formatter
- New PHP formatter: Testing selenium formatter (Adam Goucher)

Known Issues:

- Issue 1728 (on Google Code) - Firefox 4 eliminated support for the highlight. So the Find button has stopped working under Firefox 4 on Windows.
- Issue 1729 (on Google Code) - The Plugin pane in the Options dialog is not showing any text in Firefox 4 on Windows 7.
- Issues have been reported in Selenium IDE on Ubuntu 11, which are not related to Selenium IDE. See comments on issue 1642 (on Google Code).

Another packaging problem broke the various things that used getText(). Which of course is one of the most commonly used bits of the API.

- BUG - properly including se-core atoms

As a result, we've started to rebuild the test suite for things. It's going to take awhile to get the coverage we're hoping for, but it'll be worth it if we can go at least 2 days after a release before becoming embarrassed.

Upgrade Notes:

- Due to the atoms being included properly, some of the behaviour around accessing boolean attributes has changed. See <http://seleniumhq.wordpress.com/2010/12/09/atoms-have-come-to-selenium-ide/> for details.

1.0.9

What started out as a pretty major change in terms of packaging ended up including two significant bug fixes as well. Hopefully we avoid that sort of thing with the release. Not that I don't expect it. :)

- BUG - Sizzle CSS library not included
- BUG - Recording works with FF 4.0b7

What 1.0.9 was supposed to only have was...

- NEW - Formatters are **all** plugins. This effectively separates the development of an individual format from the development of the editor. Now, this means that when you install things for the first time you get a tonne of addons. That is ok. Don't panic. Oh, and it also means if you don't want them you have the option to. Not only does this mean fixes to formats get distributed sooner (PHP, I'm looking at you) but 3rd parties will be able to make better packaging choices by having the editor plus their formatters.

Other stuff

- BUG - the JUnit 4 formatter doesn't try to use a string as the port number
- BUG - the window when creating new formats properly closes now
- BUG - removed the 'find' button if on OSX since it doesn't do anything on this platform (its a FF bug)
- BUG - some hard coded strings have been internationalized
- NEW - autocomplete has been enhanced somewhat - see <http://code.google.com/p/selenium/issues/detail?id=992>
- BUG - when switching build systems, the icons for menus and such got left out of the package
- BUG - commands are trimmed of whitespace before executing which was sometimes a source of great confusion
- BUG - now preserves whitespace when displaying diffs in the log

1.0.8

This release is primarily to get FF4 support out into the wild since it is getting to the advanced beta phase, but there is also a fair bit of other bug fixes in there as well. About 75% of the fixes in the release are directly the work of Samit Badle and the vast remainder by Jérémie Héault.

- BUG - There was an annoying bug where 'clickAndWait' would be saved as click, but has been fixed. see <http://code.google.com/p/selenium/issues/detail?id=419>
- NEW -This could arguably be considered a bug fix, but if you changed format from HTML to something else then made an edit and switched back again to HTML your script contents would be lost. At its heart, the HTML -> something conversion is one way and so there is now a warning about possibly losing your code. The warning only happens the first time though so you can still shoot yourself in the foot; its just harder
- BUG - element locator works for table rows. see <http://code.google.com/p/selenium/issues/detail?id=485>

- BUG - the default timeout setting of se-ide is now actually used. see <http://code.google.com/p/selenium/issues/detail?id=552>
- NEW - the ‘run in the selenium testrunner’ option has been removed. The supported methods in se-ide are the play single, play suite and if you need more there is always se-rc with a language binding or -htmlSuite
- BUG - the base url wouldn’t change on occasion, much to the frustration of many
- NEW - a JUnit 4 formatter was added
- BUG - the RSpec formatter had some additional tweaks
- BUG - test suite html can now have tests from different folders
- BUG - test suite saving triggers got a bit of attention so add/delete/modify is a little more robust
- NEW - if you resize your se-ide and/or move it around your screen, the size and position are saved between sessions
- BUG - the logic around when to prompt for saving wasn’t really that nice, but its been fixed
- NEW - uses ‘browser atoms’ like the rest of Selenium
- NEW - CSS locator execution is handled through Sizzle
- NEW - can now add multiple test cases to a suite at once
- NEW - addition to the se-ide plugin api to add se-ide extensions to manipulate how recording is done - <http://reallysimplethings.wordpress.com/2010/10/11/the-selenium-ide-1-x-plugin-api-part-12-adding-locator-builders/>
- NEW - the case of the missing log messages is now solved
- NEW - Firefox 4 support

1.0.7

Only a couple of things of note in this release to end-users which is somewhat silly since it is a month overdue, but that was due to some build changes that took a bit of work to get the kinks worked out. Should be ok now though.

- NEW - you can now drag-and-drop command around instead of the cut-insert-paste dance that you used to have to do (Jérémie Héault)
- NEW - same thing with tests in the test suite panel (Jérémie Héault)
- NEW - an new optional parameter when registering your se-ide plugin to allow for command exporting. see <http://adam.goucher.ca/?p=1456> for details (Adam Goucher)
- NEW - Swedish locale sv-SE now has translations (Olle Jonsson)
- BUG - Some people were reporting an annoying popup when starting se-ide without any plugins installed (Adam Goucher)

1.0.6

The big thing in this release is that the scary log message that was showing up on ‘open’ is fixed. The other big things are:

- BUG - The scary log message that was happening when you used ‘open’ has had its underlying cause fixed (Adam Goucher, Jérémie Héault)
- BUG - fixed a build issue with FF 3.6 and type-ahead for commands (Jérémie Héault)
- BUG - fixed some PHP export issues - see <http://jira.openqa.org/browse/SIDE-346> and <http://jira.openqa.org/browse/SIDE-183> (Adam Goucher)
- BUG - there was a packaging issue around user-extensions (Adam Goucher)
- BUG - ide won’t put ‘name=’ as the Target when recording a selectWindow (David Burns)
- BUG - to avoid confusion, when viewing formatter source, if it is read-only the button says ‘ok’ and if it is editable then it is ‘save’ (Jérémie Héault)
- NEW - you can now set a preference on whether you want record to be on or off when you start ide (Adam Goucher)
- NEW - se-ide plugin information is read from the plugin’s install.rdf (most people won’t care about this, but it’s pretty cool from a geek perspective)

1.0.5

One thing that does not really fit the BUG or NEW label is that the code for Se-IDE is now in the main repo rather than tucked away in a somewhat hidden location.

- BUG - user formats were not appearing in the list (Adam Goucher)
- BUG - constrained how iframes were loaded; which is why AMO was unhappy (Adam Goucher)
- BUG - a whole bunch of tweaks to the existing formats (Dave Hunt)
- BUG - a bunch of French translation fixes / additions (Jérémie Héault)
- BUG - the reload user extensions button only shows up if you have the developer tool checkbox checked (Jérémie Héault)
- BUG - labelling access keys on test runner (Olle Jonsson)
- BUG - cleaned up a bunch of references from OpenQA to SeleniumHQ (Olle Jonsson)
- BUG - had an = instead of == (Olle Jonsson)
- BUG - adding a bunch of ;'s to make jslint shut up (Olle Jonsson)
- BUG - getting rid of the 'setting something that only has a getter' message in Firefox 3.6 (Dan Fabulich)
- NEW - self hosting of updates to avoid delays at AMO (Adam Goucher)
- NEW - the version of se-ide is now in the title bar (Adam Goucher)
- NEW - added some Se-IDE specific icons here and there (Adam Goucher, Dave Hunt)
- NEW - preferences can now be Bool's as well (Adam Goucher)
- NEW - added addPlugin(id) to the plugin API (Adam Goucher)
- NEW - added a new panel to the Options screen around plugins. It doesn't do much now other than list the plugins that registered themselves through addPlugin, but should do more for 1.0.6 (Adam Goucher)

1.0.4

Selenium IDE 1.0.4 marks a resurgence in the project with releases planned for the middle of each month. Here are the changes that have happened between versions 1.0.2 and 1.0.4 of Selenium IDE. (Don't ask what happened to version 1.0.3)

- BUG - Supported Firefox version increased to include the 3.6 series (Santiago Suarez Ordoñez)
- BUG - Removed the Ruby formatter that was flagged as 'deprecated' (Adam Goucher)
- NEW - Ruby formatter updated to use the selenium-client gem (<http://selenium-client.rubyforge.org/>) (Adam Goucher)
- NEW - Ability to add custom user-extensions to extend the Selenium API through plugins to Selenium IDE (Adam Goucher)
- NEW - Ability to add custom formatters to extend which languages are available to users through plugins to Selenium IDE (Adam Goucher)
- NEW - Can now load changes to user extensions without having to restart Selenium IDE (Jérémie Héault)
- NEW - RSpec formatter

Acknowledgements

Version 1.0.4 would not have happened without the following assistance

- Sauce Labs' sponsoring of Adam Goucher to work on it
- Jérémie Héault and the SERLI team for their Helium plugin (which was the proof an API could / should be developed for Se-IDE)
- Dave Hunt for his feedback on pre-release versions

For issues with this release or features you would like to see in future releases, please log them in the Google Code Issue tracker (<https://github.com/SeleniumHQ/selenium/issues>) using the *ide* label so they don't get lost.

-adam

7.5 - JSON Wire Protocol Specification

The endpoints and payloads for the now-obsolete open source protocol that was the precursor to the [W3C specification](#).

This documentation previously located [on the wiki](#)

All implementations of WebDriver that communicate with the browser, or a RemoteWebDriver server shall use a common wire protocol. This wire protocol defines a [RESTful web service](#) using [JSON](#) over HTTP.

The protocol will assume that the WebDriver API has been “flattened”, but there is an expectation that client implementations will take a more Object-Oriented approach, as demonstrated in the existing Java API. The wire protocol is implemented in request/response pairs of “commands” and “responses”.

Terms and Concepts

Client

The machine on which the WebDriver API is being used.

Session

The machine running the RemoteWebDriver. This term may also refer to a specific browser that implements the wire protocol directly, such as the FirefoxDriver or iPhoneDriver.

The server should maintain one browser per session. Commands sent to a session will be directed to the corresponding browser.

WebElement

An object in the WebDriver API that represents a DOM element on the page.

WebElement JSON Object

The JSON representation of a WebElement for transmission over the wire. This object will have the following properties:

Key	Type	Description
ELEMENT	string	The opaque ID assigned to the element by the server. This ID should be used in all subsequent commands issued against the element.

Capabilities JSON Object

Not all server implementations will support every WebDriver feature. Therefore, the client and server should use JSON objects with the properties listed below when describing which features a session supports.

Key	Type	Description

Key	Type	Description
browserName	string	The name of the browser being used; should be one of {android, chrome, firefox, htmlunit, internet explorer, iPhone, iPad, opera, safari} .
version	string	The browser version, or the empty string if unknown.
platform	string	A key specifying which platform the browser is running on. This value should be one of {WINDOWS XP VISTA MAC LINUX UNIX} . When requesting a new session, the client may specify ANY to indicate any available platform may be used.
javascriptEnabled	boolean	Whether the session supports executing user supplied JavaScript in the context of the current page.
takesScreenshot	boolean	Whether the session supports taking screenshots of the current page.
handlesAlerts	boolean	Whether the session can interact with modal popups, such as <code>window.alert</code> and <code>window.confirm</code> .
databaseEnabled	boolean	Whether the session can interact database storage.
locationContextEnabled	boolean	Whether the session can set and query the browser's location context.
applicationCacheEnabled	boolean	Whether the session can interact with the application cache.
browserConnectionEnabled	boolean	Whether the session can query for the browser's connectivity and disable it if desired.
cssSelectorsEnabled	boolean	Whether the session supports CSS selectors when searching for elements.
webStorageEnabled	boolean	Whether the session supports interactions with storage objects .
rotatable	boolean	Whether the session can rotate the current page's current layout between portrait and landscape orientations (only applies to mobile platforms).
acceptSslCerts	boolean	Whether the session should accept all SSL certs by default.
nativeEvents	boolean	Whether the session is capable of generating native events when simulating user input.
proxy	proxy object	Details of any proxy to use. If no proxy is specified, whatever the system's current or default state is used. The format is specified under Proxy JSON Object.
unexpectedAlertBehaviour	string	What the browser should do with an unhandled alert before throwing out the UnhandledAlertException. Possible values are "accept", "dismiss" and "ignore"

Key	Type	Description
elementScrollBehavior	integer	Allows the user to specify whether elements are scrolled into the viewport for interaction to align with the top (0) or bottom (1) of the viewport. The default value is to align with the top of the viewport. Supported in IE and Firefox (since 2.36)

Desired Capabilities

A Capabilities JSON Object sent by the client describing the capabilities a new session created by the server should possess. Any omitted keys implicitly indicate the corresponding capability is irrelevant. More at [DesiredCapabilities](#).

Actual Capabilities

A Capabilities JSON Object returned by the server describing what features a session actually supports. Any omitted keys implicitly indicate the corresponding capability is not supported.

Cookie JSON Object

A JSON object describing a Cookie.

Key	Type	Description
name	string	The name of the cookie.
value	string	The cookie value.
path	string	(Optional) The cookie path. ¹
domain	string	(Optional) The domain the cookie is visible to. ¹
secure	boolean	(Optional) Whether the cookie is a secure cookie. ¹
httpOnly	boolean	(Optional) Whether the cookie is an httpOnly cookie. ¹
expiry	number	(Optional) When the cookie expires, specified in seconds since midnight, January 1, 1970 UTC. ¹

¹ When returning Cookie objects, the server should only omit an optional field if it is incapable of providing the information.

Log Entry JSON Object

A JSON object describing a log entry.

Key	Type	Description
timestamp	number	The timestamp of the entry.
level	string	The log level of the entry, for example, "INFO" (see log levels).
message	string	The log message.

Log Levels

Log levels in order, with finest level on top and coarsest level at the bottom.

Level	Description
ALL	All log messages. Used for fetching of logs and configuration of logging.
DEBUG	Messages for debugging.
INFO	Messages with user information.
WARNING	Messages corresponding to non-critical problems.
SEVERE	Messages corresponding to critical errors.
OFF	No log messages. Used for configuration of logging.

Log Type

The table below lists common log types. Other log types, for instance, for performance logging may also be available.

Log Type	Description
client	Logs from the client.
driver	Logs from the webdriver.
browser	Logs from the browser.
server	Logs from the server.

Proxy JSON Object

A JSON object describing a Proxy configuration.

Key	Type	Description
proxyType	string	(Required) The type of proxy being used. Possible values are: direct - A direct connection - no proxy in use, manual - Manual proxy settings configured, e.g. setting a proxy for HTTP, a proxy for FTP, etc, pac - Proxy autoconfiguration from a URL, autodetect - Proxy autodetection, probably with WPAD, system - Use system settings
proxyAutoconfigUrl	string	(Required if proxyType == pac , Ignored otherwise) Specifies the URL to be used for proxy autoconfiguration. Expected format example: http://hostname.com:1234/pacfile
ftpProxy, httpProxy, sslProxy, socksProxy	string	(Optional, Ignored if proxyType != manual) Specifies the proxies to be used for FTP, HTTP, HTTPS and SOCKS requests respectively. Behaviour is undefined if a request is made, where the proxy for the particular protocol is undefined, if proxyType is manual . Expected format example: hostname.com:1234

Key	Type	Description
socksUsername	string	(Optional, Ignored if proxyType != manual and socksProxy is not set) Specifies SOCKS proxy username.
socksPassword	string	(Optional, Ignored if proxyType != manual and socksProxy is not set) Specifies SOCKS proxy password.
noProxy	string	(Optional, Ignored if proxyType != manual) Specifies proxy bypass addresses. Format is driver specific.

Messages

Commands

WebDriver command messages should conform to the [HTTP/1.1 request specification](#). Although the server may be extended to respond to other content-types, the wire protocol dictates that all commands accept a content-type of `application/json; charset=UTF-8`. Likewise, the message bodies for POST and PUT request must use an `application/json; charset=UTF-8` content-type.

Each command in the WebDriver service will be mapped to an HTTP method at a specific path. Path segments prefixed with a colon (:) indicate that segment is a variable used to further identify the underlying resource. For example, consider an arbitrary resource mapped as:

```
GET /favorite/color/:name
```

Given this mapping, the server should respond to GET requests sent to “/favorite/color/Jack” and “/favorite/color/Jill”, with the variable `:name` set to “Jack” and “Jill”, respectively.

Responses

Command responses shall be sent as [HTTP/1.1 response messages](#). If the remote server must return a 4xx response, the response body shall have a Content-Type of text/plain and the message body shall be a descriptive message of the bad request. For all other cases, if a response includes a message body, it must have a Content-Type of application/json; charset=UTF-8 and will be a JSON object with the following properties:

Key	Type	Description
sessionId	string	null
status	number	A status code summarizing the result of the command. A non-zero value indicates that the command failed.
value	*	The response JSON value.

Response Status Codes

The wire protocol will inherit its status codes from those used by the InternetExplorerDriver:

Code	Summary	Detail
0	Success	The command executed successfully.
6	NoSuchDriver	A session is either terminated or not started
7	NoSuchElement	An element could not be located on the page using the given search parameters.

Code	Summary	Detail
8	NoSuchFrame	A request to switch to a frame could not be satisfied because the frame could not be found.
9	UnknownCommand	The requested resource could not be found, or a request was received using an HTTP method that is not supported by the mapped resource.
10	StaleElementReference	An element command failed because the referenced element is no longer attached to the DOM.
11	ElementNotVisible	An element command could not be completed because the element is not visible on the page.
12	InvalidElementState	An element command could not be completed because the element is in an invalid state (e.g. attempting to click a disabled element).
13	UnknownError	An unknown server-side error occurred while processing the command.
15	ElementIsNotSelectable	An attempt was made to select an element that cannot be selected.
17	JavaScriptError	An error occurred while executing user supplied JavaScript.
19	XPathLookupError	An error occurred while searching for an element by XPath.
21	Timeout	An operation did not complete before its timeout expired.
23	NoSuchWindow	A request to switch to a different window could not be satisfied because the window could not be found.
24	InvalidCookieDomain	An illegal attempt was made to set a cookie under a different domain than the current page.
25	UnableToSetCookie	A request to set a cookie's value could not be satisfied.
26	UnexpectedAlertOpen	A modal dialog was open, blocking this operation
27	NoAlertOpenError	An attempt was made to operate on a modal dialog when one was not open.
28	ScriptTimeout	A script did not complete before its timeout expired.
29	InvalidElementCoordinates	The coordinates provided to an interactions operation are invalid.
30	IMENotAvailable	IME was not available.
31	IMEEngineActivationFailed	An IME engine could not be started.
32	InvalidSelector	Argument was an invalid selector (e.g. XPath/CSS).
33	SessionNotCreatedException	A new session could not be created.
34	MoveTargetOutOfBounds	Target provided for a move action is out of bounds.

The client should interpret a 404 Not Found response from the server as an “Unknown command” response. All other 4xx and 5xx responses from the server that do not define a status field should be interpreted as “Unknown error” responses.

Error Handling

There are two levels of error handling specified by the wire protocol: invalid requests and failed commands.

Invalid Requests

All invalid requests should result in the server returning a 4xx HTTP response. The response Content-Type should be set to text/plain and the message body should be a descriptive error message. The categories of invalid requests are as follows:

Unknown Commands

If the server receives a command request whose path is not mapped to a resource in the REST service, it should respond with a [404 Not Found](#) message.

Unimplemented Commands

Every server implementing the WebDriver wire protocol must respond to every defined command. If an individual command has not been implemented on the server, the server should respond with a [501 Not Implemented](#) error message. Note this is the only error in the Invalid Request category that does not return a [4xx](#) status code.

Variable Resource Not Found

If a request path maps to a variable resource, but that resource does not exist, then the server should respond with a [404 Not Found](#). For example, if ID `my-session` is not a valid session ID on the server, and a command is sent to `GET /session/my-session HTTP/1.1`, then the server should gracefully return a [404](#).

Invalid Command Method

If a request path maps to a valid resource, but that resource does not respond to the request method, the server should respond with a [405 Method Not Allowed](#). The response must include an Allows header with a list of the allowed methods for the requested resource.

Missing Command Parameters

If a POST/PUT command maps to a resource that expects a set of JSON parameters, and the response body does not include one of those parameters, the server should respond with a [400 Bad Request](#). The response body should list the missing parameters.

Failed Commands

If a request maps to a valid command and contains all of the expected parameters in the request body, yet fails to execute successfully, then the server should send a 500 Internal Server Error. This response should have a Content-Type of `application/json; charset=UTF-8` and the response body should be a well formed JSON response object.

The response status should be one of the defined status codes and the response value should be another JSON object with detailed information for the failing command:

Key	Type	Description
message	string	A descriptive message for the command failure.
screen	string	(Optional) If included, a screenshot of the current page as a base64 encoded string.

Key	Type	Description
class	string	(Optional) If included, specifies the fully qualified class name for the exception that was thrown when the command failed.
stackTrace	array	(Optional) If included, specifies an array of JSON objects describing the stack trace for the exception that was thrown when the command failed. The zeroeth element of the array represents the top of the stack.

Each JSON object in the stackTrace array must contain the following properties:

Key	Type	Description
fileName	string	The name of the source file containing the line represented by this frame.
className	string	The fully qualified class name for the class active in this frame. If the class name cannot be determined, or is not applicable for the language the server is implemented in, then this property should be set to the empty string.
methodName	string	The name of the method active in this frame, or the empty string if unknown/not applicable.
lineNumber	number	The line number in the original source file for the frame, or 0 if unknown.

Resource Mapping

Resources in the WebDriver REST service are mapped to individual URL patterns. Each resource may respond to one or more HTTP request methods. If a resource responds to a GET request, then it should also respond to HEAD requests. All resources should respond to OPTIONS requests with an `Allow` header field, whose value is a list of all methods that resource responds to.

If a resource is mapped to a URL containing a variable path segment name, that path segment should be used to further route the request. Variable path segments are indicated in the resource mapping by a colon-prefix. For example, consider the following:

```
/favorite/color/:person
```

A resource mapped to this URL should parse the value of the `:person` path segment to further determine how to respond to the request. If this resource received a request for `/favorite/color/Jack`, then it should return Jack's favorite color. Likewise, the server should return Jill's favorite color for any requests to `/favorite/color/Jill`.

Two resources may only be mapped to the same URL pattern if one of those resources' patterns contains variable path segments, and the other does not. In these cases, the server should always route requests to the resource whose path is the best match for the request. Consider the following two resource paths:

1. `/session/:sessionId/element/active`
2. `/session/:sessionId/element/:id`

Given these mappings, the server should always route requests whose final path segment is active to the first resource. All other requests should be routed to second.

Command Reference

Command Summary

HTTP		Method Path	Summary
GET	<u>/status</u>		Query the server's current status.
POST	<u>/session</u>		Create a new session.
GET	<u>/sessions</u>		Returns a list of the currently active sessions.
GET	<u>/session/:sessionId</u>		Retrieve the capabilities of the specified session.
DELETE	<u>/session/:sessionId</u>		Delete the session.
POST	<u>/session/:sessionId/timeouts</u>		Configure the amount of time the particular type of operation can execute for before they are aborted and a
POST	<u>/session/:sessionId/timeouts/async_script</u>		Set the amount of time, in milliseconds, that asynchronous scripts executed by /session/:sessionId/execute_async are permitted to run before they are aborted and a
POST	<u>/session/:sessionId/timeouts/implicit_wait</u>		Set the amount of time the driver should wait when searching for elements.
GET	<u>/session/:sessionId/window_handle</u>		Retrieve the current window handle.
GET	<u>/session/:sessionId/window_handles</u>		Retrieve the list of all window handles available to the session.
GET	<u>/session/:sessionId/url</u>		Retrieve the URL of the current page.
POST	<u>/session/:sessionId/url</u>		Navigate to a new URL.
POST	<u>/session/:sessionId/forward</u>		Navigate forwards in the browser history, if possible.
POST	<u>/session/:sessionId/back</u>		Navigate backwards in the browser history, if possible.
POST	<u>/session/:sessionId/refresh</u>		Refresh the current page.
POST	<u>/session/:sessionId/execute</u>		Inject a snippet of JavaScript into the page for execution in the context of the currently selected frame.
POST	<u>/session/:sessionId/execute_async</u>		Inject a snippet of JavaScript into the page for execution in the context of the currently selected frame.
GET	<u>/session/:sessionId/screenshot</u>		Take a screenshot of the current page.

HTTP

Method	Path	Summary
GET	<code>/session/:sessionId/ime/available_engines</code>	List all available engines on the machine.
GET	<code>/session/:sessionId/ime/active_engine</code>	Get the name of the active IME engine.
GET	<code>/session/:sessionId/ime/activated</code>	Indicates whether IME input is active at the moment (not if it's available).
POST	<code>/session/:sessionId/ime/deactivate</code>	De-activates the currently-active IME engine.
POST	<code>/session/:sessionId/ime/activate</code>	Make an engines that is available (appears on the list returned by getAvailableEngines) active.
POST	<code>/session/:sessionId/frame</code>	Change focus to another frame on the page.
POST	<code>/session/:sessionId/frame/parent</code>	Change focus to the parent container.
POST	<code>/session/:sessionId/window</code>	Change focus to another window.
DELETE	<code>/session/:sessionId/window</code>	Close the current window.
POST	<code>/session/:sessionId/window/:windowHandle/size</code>	Change the size of the specified window.
GET	<code>/session/:sessionId/window/:windowHandle/size</code>	Get the size of the specified window.
POST	<code>/session/:sessionId/window/:windowHandle/position</code>	Change the position of the specified window.
GET	<code>/session/:sessionId/window/:windowHandle/position</code>	Get the position of the specified window.
POST	<code>/session/:sessionId/window/:windowHandle/maximize</code>	Maximize the specified window if not already maximized.
GET	<code>/session/:sessionId/cookie</code>	Retrieve all cookies visible to the current page.
POST	<code>/session/:sessionId/cookie</code>	Set a cookie.
DELETE	<code>/session/:sessionId/cookie</code>	Delete all cookies visible to the current page.
DELETE	<code>/session/:sessionId/cookie/:name</code>	Delete the cookie with the given name.
GET	<code>/session/:sessionId/source</code>	Get the current page source.
GET	<code>/session/:sessionId/title</code>	Get the current page title.

HTTP

Method	Path	Summary
POST	<code>/session/:sessionId/element</code>	Search for an element on the page, starting from the document root.
POST	<code>/session/:sessionId/elements</code>	Search for multiple elements on the page, starting from the document root.
POST	<code>/session/:sessionId/element/active</code>	Get the element on the page that currently has focus.
GET	<code>/session/:sessionId/element/:id</code>	Describe the identified element.
POST	<code>/session/:sessionId/element/:id/element</code>	Search for an element on the page, starting from the identified element.
POST	<code>/session/:sessionId/element/:id/elements</code>	Search for multiple elements on the page, starting from the identified element.
POST	<code>/session/:sessionId/element/:id/click</code>	Click on an element.
POST	<code>/session/:sessionId/element/:id/submit</code>	Submit a FORM element.
GET	<code>/session/:sessionId/element/:id/text</code>	Returns the visible text for the element.
POST	<code>/session/:sessionId/element/:id/value</code>	Send a sequence of key strokes to an element.
POST	<code>/session/:sessionId/keys</code>	Send a sequence of key strokes to the active element.
GET	<code>/session/:sessionId/element/:id/name</code>	Query for an element's tag name.
POST	<code>/session/:sessionId/element/:id/clear</code>	Clear a TEXTAREA or text INPUT element's value.
GET	<code>/session/:sessionId/element/:id/selected</code>	Determine if an OPTION element or an INPUT element of type checkbox or radiobutton is currently selected.
GET	<code>/session/:sessionId/element/:id/enabled</code>	Determine if an element is currently enabled.
GET	<code>/session/:sessionId/element/:id/attribute/:name</code>	Get the value of an element's attribute.
GET	<code>/session/:sessionId/element/:id>equals/:other</code>	Test if two element IDs refer to the same DOM element.
GET	<code>/session/:sessionId/element/:id/displayed</code>	Determine if an element is currently displayed.
GET	<code>/session/:sessionId/element/:id/location</code>	Determine an element's location on the page.

HTTP

Method	Path	Summary
GET	<code>/session/:sessionId/element/:id/location_in_view</code>	Determine an element's location on the screen once it has been scrolled into view.
GET	<code>/session/:sessionId/element/:id/size</code>	Determine an element's size in pixels.
GET	<code>/session/:sessionId/element/:id/css/:propertyName</code>	Query the value of an element's computed CSS property.
GET	<code>/session/:sessionId/orientation</code>	Get the current browser orientation.
POST	<code>/session/:sessionId/orientation</code>	Set the browser orientation.
GET	<code>/session/:sessionId/alert_text</code>	Gets the text of the currently displayed JavaScript <code>alert()</code> , <code>confirm()</code> , or <code>prompt()</code> dialog.
POST	<code>/session/:sessionId/alert_text</code>	Sends keystrokes to a JavaScript <code>prompt()</code> dialog.
POST	<code>/session/:sessionId/accept_alert</code>	Accepts the currently displayed alert dialog.
POST	<code>/session/:sessionId/dismiss_alert</code>	Dismisses the currently displayed alert dialog.
POST	<code>/session/:sessionId/moveto</code>	Move the mouse by an offset of a specified element.
POST	<code>/session/:sessionId/click</code>	Click any mouse button (at the coordinates set by the last move command).
POST	<code>/session/:sessionId/buttondown</code>	Click and hold the left mouse button (at the coordinates set by the last moveto command).
POST	<code>/session/:sessionId/buttonup</code>	Releases the mouse button previously held (where the mouse was currently at).
POST	<code>/session/:sessionId/doubleclick</code>	Double-clicks at the current mouse coordinates (set by moveto).
POST	<code>/session/:sessionId/touch/click</code>	Single tap on the touch enabled device.
POST	<code>/session/:sessionId/touch/down</code>	Finger down on the screen.
POST	<code>/session/:sessionId/touch/up</code>	Finger up on the screen.
POST	<code>/session/:sessionId/touch/move</code>	Finger move on the screen.
POST	<code>/session/:sessionId/touch/scroll</code>	Scroll on the touch screen using finger based motion events.

HTTP

Method	Path	Summary
POST	<u>session/:sessionId/touch/scroll</u>	Scroll on the touch screen using finger based motion events.
POST	<u>session/:sessionId/touch/doubleclick</u>	Double tap on the touch screen using finger motion events.
POST	<u>session/:sessionId/touch/longclick</u>	Long press on the touch screen using finger motion events.
POST	<u>session/:sessionId/touch/flick</u>	Flick on the touch screen using finger motion events.
POST	<u>session/:sessionId/touch/flick</u>	Flick on the touch screen using finger motion events.
GET	<u>/session/:sessionId/location</u>	Get the current geo location.
POST	<u>/session/:sessionId/location</u>	Set the current geo location.
GET	<u>/session/:sessionId/local_storage</u>	Get all keys of the storage.
POST	<u>/session/:sessionId/local_storage</u>	Set the storage item for the given key.
DELETE	<u>/session/:sessionId/local_storage</u>	Clear the storage.
GET	<u>/session/:sessionId/local_storage/key/:key</u>	Get the storage item for the given key.
DELETE	<u>/session/:sessionId/local_storage/key/:key</u>	Remove the storage item for the given key.
GET	<u>/session/:sessionId/local_storage/size</u>	Get the number of items in the storage.
GET	<u>/session/:sessionId/session_storage</u>	Get all keys of the storage.
POST	<u>/session/:sessionId/session_storage</u>	Set the storage item for the given key.
DELETE	<u>/session/:sessionId/session_storage</u>	Clear the storage.
GET	<u>/session/:sessionId/session_storage/key/:key</u>	Get the storage item for the given key.
DELETE	<u>/session/:sessionId/session_storage/key/:key</u>	Remove the storage item for the given key.
GET	<u>/session/:sessionId/session_storage/size</u>	Get the number of items in the storage.
POST	<u>/session/:sessionId/log</u>	Get the log for a given log type.
GET	<u>/session/:sessionId/log/types</u>	Get available log types.
GET	<u>/session/:sessionId/application_cache/status</u>	Get the status of the html5 application cache.

Command Detail

/status

GET /status

Query the server's current status. The server should respond with a general "HTTP 200 OK" response if it is alive and accepting commands. The response body should be a JSON object describing the state of the server. All server implementations should return two basic objects describing the server's current platform and when the server was built. All fields are optional; if omitted, the client should assume the value is unknown. Furthermore, server implementations may include additional fields not listed here.

Key Type Description

build	object
build.version	string A generic release label (i.e. "2.0rc3")
build.revision	string The revision of the local source control client from which the server was built
build.time	string A timestamp from when the server was built.
os	object
os.arch	string The current system architecture.
os.name	string The name of the operating system the server is currently running on: "windows", "linux", etc.
os.version	string The operating system version.

Returns:

{object} An object describing the general status of the server.

/session

POST /session

Create a new session. The server should attempt to create a session that most closely matches the desired and required capabilities. Required capabilities have higher priority than desired capabilities and must be set for the session to be created.

JSON Parameters:

desiredCapabilities - {object} An object describing the session's [desired capabilities](#).

requiredCapabilities - {object} An object describing the session's [required capabilities](#) (Optional).

Returns:

{object} An object describing the session's [capabilities](#).

Potential Errors:

SessionNotCreatedException - If a required capability could not be set.

/sessions

GET /sessions

Returns a list of the currently active sessions. Each session will be returned as a list of JSON objects with the following keys:

Key Type Description

id	string The session ID.
capabilities	object An object describing the session's <u>capabilities</u> .

Returns:

{Array.<Object>} A list of the currently active sessions.

/session/:sessionId

GET /session/:sessionId

Retrieve the capabilities of the specified session.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{object}` An object describing the session's [capabilities](#).

DELETE /session/:sessionId

Delete the session.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

/session/:sessionId/timeouts

POST /session/:sessionId/timeouts

Configure the amount of time that a particular type of operation can execute for before they are aborted and a `|Timeout|` error is returned to the client.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`type` - `{string}` The type of operation to set the timeout for. Valid values are: "script" for script timeouts, "implicit" for modifying the implicit wait timeout and "page load" for setting a page load timeout.

`ms` - `{number}` The amount of time, in milliseconds, that time-limited commands are permitted to run.

/session/:sessionId/timeouts/async_script

POST /session/:sessionId/timeouts/async_script

Set the amount of time, in milliseconds, that asynchronous scripts executed by

`/session/:sessionId/execute_async` are permitted to run before they are aborted and a `|Timeout|` error is returned to the client.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`ms` - `{number}` The amount of time, in milliseconds, that time-limited commands are permitted to run.

/session/:sessionId/timeouts/implicit_wait

POST /session/:sessionId/timeouts/implicit_wait

Set the amount of time the driver should wait when searching for elements. When searching for a single element, the driver should poll the page until an element is found or the timeout expires, whichever occurs first. When searching for multiple elements, the driver should poll the page until at least one element is found or the timeout expires, at which point it should return an empty list.

If this command is never sent, the driver should default to an implicit wait of 0ms.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`ms` - `{number}` The amount of time to wait, in milliseconds. This value has a lower bound of 0.

/session/:sessionId/window_handle

GET /session/:sessionId/window_handle

Retrieve the current window handle.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{string}` The current window handle.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/window_handles

GET /session/:sessionId/window_handles

Retrieve the list of all window handles available to the session.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{Array.<string>}` A list of window handles.

/session/:sessionId/url

GET /session/:sessionId/url

Retrieve the URL of the current page.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{string}` The current URL.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

POST /session/:sessionId/url

Navigate to a new URL.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`url` - `{string}` The URL to navigate to.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/forward

POST /session/:sessionId/forward

Navigate forwards in the browser history, if possible.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/back

POST /session/:sessionId/back

Navigate backwards in the browser history, if possible.

URL Parameters:

:sessionId - ID of the session to route the command to.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

/session/:sessionId/refresh

POST /session/:sessionId/refresh

Refresh the current page.

URL Parameters:

:sessionId - ID of the session to route the command to.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

/session/:sessionId/execute

POST /session/:sessionId/execute

Inject a snippet of JavaScript into the page for execution in the context of the currently selected frame. The executed script is assumed to be synchronous and the result of evaluating the script is returned to the client.

The `script` argument defines the script to execute in the form of a function body. The value returned by that function will be returned to the client. The function will be invoked with the provided `args` array and the values may be accessed via the `arguments` object in the order specified.

Arguments may be any JSON-primitive, array, or JSON object. JSON objects that define a [WebElement reference](#) will be converted to the corresponding DOM element. Likewise, any WebElements in the script result will be returned to the client as [WebElement JSON objects](#).

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

`script` - {string} The script to execute.

`args` - {Array.<*>} The script arguments.

Returns:

{*} The script result.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If one of the script arguments is a WebElement that is not attached to the page's DOM.

JavaScriptError - If the script throws an Error.

/session/:sessionId/execute_async

POST /session/:sessionId/execute_async

Inject a snippet of JavaScript into the page for execution in the context of the currently selected frame. The executed script is assumed to be asynchronous and must signal that is done by invoking the provided callback, which is always provided as the final argument to the function. The value to this callback will be returned to the client.

Asynchronous script commands may not span page loads. If an `unload` event is fired while waiting for a script result, an error should be returned to the client.

The `script` argument defines the script to execute in the form of a function body. The function will be invoked with the provided `args` array and the values may be accessed via the `arguments` object in the

order specified. The final argument will always be a callback function that must be invoked to signal that the script has finished.

Arguments may be any JSON-primitive, array, or JSON object. JSON objects that define a [WebElement reference](#) will be converted to the corresponding DOM element. Likewise, any WebElements in the script result will be returned to the client as [WebElement JSON objects](#).

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`script` - {string} The script to execute.

`args` - {Array.<*>} The script arguments.

Returns:

{*} The script result.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

`StaleElementReference` - If one of the script arguments is a WebElement that is not attached to the page's DOM.

`Timeout` - If the script callback is not invoked before the timeout expires. Timeouts are controlled by the `/session/:sessionId/timeout/async_script` command.

`JavaScriptError` - If the script throws an Error or if an `unload` event is fired while waiting for the script to finish.

/session/:sessionId/screenshot

GET /session/:sessionId/screenshot

Take a screenshot of the current page.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

{string} The screenshot as a base64 encoded PNG.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/ime/available_engines

GET /session/:sessionId/ime/available_engines

List all available engines on the machine. To use an engine, it has to be present in this list.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

{Array.<string>} A list of available engines

Potential Errors:

`ImeNotAvailableException` - If the host does not support IME

/session/:sessionId/ime/active_engine

GET /session/:sessionId/ime/active_engine

Get the name of the active IME engine. The name string is platform specific.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

{string} The name of the active IME engine.

Potential Errors:

ImeNotAvailableException - If the host does not support IME

/session/:sessionId/ime/activated

GET /session/:sessionId/ime/activated

Indicates whether IME input is active at the moment (not if it's available).

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{boolean} true if IME input is available and currently active, false otherwise

Potential Errors:

ImeNotAvailableException - If the host does not support IME

/session/:sessionId/ime/deactivate

POST /session/:sessionId/ime/deactivate

De-activates the currently-active IME engine.

URL Parameters:

:sessionId - ID of the session to route the command to.

Potential Errors:

ImeNotAvailableException - If the host does not support IME

/session/:sessionId/ime/activate

POST /session/:sessionId/ime/activate

Make an engines that is available (appears on the list returned by getAvailableEngines) active. After this call, the engine will be added to the list of engines loaded in the IME daemon and the input sent using sendKeys will be converted by the active engine.

Note that this is a platform-independent method of activating IME (the platform-specific way being using keyboard shortcuts)

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

engine - {string} Name of the engine to activate.

Potential Errors:

ImeActivationFailedException - If the engine is not available or if the activation fails for other reasons.

ImeNotAvailableException - If the host does not support IME

/session/:sessionId/frame

POST /session/:sessionId/frame

Change focus to another frame on the page. If the frame id is null, the server should switch to the page's default content.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

id - {string|number|null|WebElement JSON Object} Identifier for the frame to change focus to.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

NoSuchFrame - If the frame specified by `id` cannot be found.

/session/:sessionId/frame/parent

POST /session/:sessionId/frame/parent

Change focus to the parent context. If the current context is the top level browsing context, the context remains unchanged.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

/session/:sessionId/window

POST /session/:sessionId/window

Change focus to another window. The window to change focus to may be specified by its server assigned window handle, or by the value of its `name` attribute.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`name` - `{string}` The window to change focus to.

Potential Errors:

NoSuchWindow - If the window specified by `name` cannot be found.

DELETE /session/:sessionId/window

Close the current window.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Potential Errors:

NoSuchWindow - If the currently selected window is already closed

/session/:sessionId/window/:windowHandle/size

POST /session/:sessionId/window/:windowHandle/size

Change the size of the specified window. If the `:windowHandle` URL parameter is "current", the currently active window will be resized.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`width` - `{number}` The new window width.

`height` - `{number}` The new window height.

GET /session/:sessionId/window/:windowHandle/size

Get the size of the specified window. If the `:windowHandle` URL parameter is "current", the size of the currently active window will be returned.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{width: number, height: number}` The size of the window.

Potential Errors:

NoSuchWindow - If the specified window cannot be found.

/session/:sessionId/window/:windowHandle/position

POST /session/:sessionId/window/:windowHandle/position

Change the position of the specified window. If the :windowHandle URL parameter is "current", the currently active window will be moved.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

x - {number} The X coordinate to position the window at, relative to the upper left corner of the screen.

y - {number} The Y coordinate to position the window at, relative to the upper left corner of the screen.

Potential Errors:

NoSuchWindow - If the specified window cannot be found.

GET /session/:sessionId/window/:windowHandle/position

Get the position of the specified window. If the :windowHandle URL parameter is "current", the position of the currently active window will be returned.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{x: number, y: number} The X and Y coordinates for the window, relative to the upper left corner of the screen.

Potential Errors:

NoSuchWindow - If the specified window cannot be found.

/session/:sessionId/window/:windowHandle/maximize

POST /session/:sessionId/window/:windowHandle/maximize

Maximize the specified window if not already maximized. If the :windowHandle URL parameter is "current", the currently active window will be maximized.

URL Parameters:

:sessionId - ID of the session to route the command to.

Potential Errors:

NoSuchWindow - If the specified window cannot be found.

/session/:sessionId/cookie

GET /session/:sessionId/cookie

Retrieve all cookies visible to the current page.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{Array.<object>} A list of [cookies](#).

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

POST /session/:sessionId/cookie

Set a cookie. If the [cookie](#) path is not specified, it should be set to "/". Likewise, if the domain is omitted, it should default to the current page's domain.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

cookie - {object} A [JSON object](#) defining the cookie to add.

DELETE /session/:sessionId/cookie

Delete all cookies visible to the current page.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Potential Errors:

`InvalidCookieDomain` - If the cookie's `domain` is not visible from the current page.

`NoSuchWindow` - If the currently selected window has been closed.

`UnableToSetCookie` - If attempting to set a cookie on a page that does not support cookies (e.g. pages with mime-type `text/plain`).

/session/:sessionId/cookie/:name

DELETE /session/:sessionId/cookie/:name

Delete the cookie with the given name. This command should be a no-op if there is no such cookie visible to the current page.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:name` - The name of the cookie to delete.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/source

GET /session/:sessionId/source

Get the current page source.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{string}` The current page source.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/title

GET /session/:sessionId/title

Get the current page title.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{string}` The current page title.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/element

POST /session/:sessionId/element

Search for an element on the page, starting from the document root. The located element will be returned as a WebElement JSON object. The table below lists the locator strategies that each server should support. Each locator must return the first matching element located in the DOM.

Strategy	Description
----------	-------------

Strategy	Description
class name	Returns an element whose class name contains the search value; compound class names are not permitted.
css selector	Returns an element matching a CSS selector.
id	Returns an element whose ID attribute matches the search value.
name	Returns an element whose NAME attribute matches the search value.
link text	Returns an anchor element whose visible text matches the search value.
partial link text	Returns an anchor element whose visible text partially matches the search value.
tag name	Returns an element whose tag name matches the search value.
xpath	Returns an element matching an XPath expression.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`using - {string}` The locator strategy to use.

`value - {string}` The The search target.

Returns:

`{ELEMENT:string}` A WebElement JSON object for the located element.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

`NoSuchElement` - If the element cannot be found.

`XPathLookupError` - If using XPath and the input expression is invalid.

/session/:sessionId/elements

POST /session/:sessionId/elements

Search for multiple elements on the page, starting from the document root. The located elements will be returned as a WebElement JSON objects. The table below lists the locator strategies that each server should support. Elements should be returned in the order located in the DOM.

Strategy	Description
class name	Returns all elements whose class name contains the search value; compound class names are not permitted.
css selector	Returns all elements matching a CSS selector.
id	Returns all elements whose ID attribute matches the search value.
name	Returns all elements whose NAME attribute matches the search value.
link text	Returns all anchor elements whose visible text matches the search value.
partial link text	Returns all anchor elements whose visible text partially matches the search value.
tag name	Returns all elements whose tag name matches the search value.
xpath	Returns all elements matching an XPath expression.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`using - {string}` The locator strategy to use.

`value - {string}` The The search target.

Returns:

`{Array.<{ELEMENT:string}>}` A list of WebElement JSON objects for the located elements.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

`XPathLookupError` - If using XPath and the input expression is invalid.

/session/:sessionId/element/active

POST /session/:sessionId/element/active

Get the element on the page that currently has focus. The element will be returned as a WebElement JSON object.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{ELEMENT:string} A WebElement JSON object for the active element.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

/session/:sessionId/element/:id

GET /session/:sessionId/element/:id

Describe the identified element.

Note: This command is reserved for future use; its return type is currently undefined.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

/session/:sessionId/element/:id/element

POST /session/:sessionId/element/:id/element

Search for an element on the page, starting from the identified element. The located element will be returned as a WebElement JSON object. The table below lists the locator strategies that each server should support. Each locator must return the first matching element located in the DOM.

Strategy Description

class	Returns an element whose class name contains the search value; compound class names are not permitted.
css selector	Returns an element matching a CSS selector.
id	Returns an element whose ID attribute matches the search value.
name	Returns an element whose NAME attribute matches the search value.
link text	Returns an anchor element whose visible text matches the search value.
partial link text	Returns an anchor element whose visible text partially matches the search value.
tag name	Returns an element whose tag name matches the search value.
xpath	Returns an element matching an XPath expression. The provided XPath expression must be applied to the server "as is"; if the expression is not relative to the element root, the server should not modify it. Consequently, an XPath query may return elements not contained in the root element's subtree.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

JSON Parameters:

using - {string} The locator strategy to use.

value - {string} The The search target.

Returns:

{ELEMENT:string} A WebElement JSON object for the located element.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

NoSuchElement - If the element cannot be found.

XPathLookupError - If using XPath and the input expression is invalid.

/session/:sessionId/element/:id/elements

POST /session/:sessionId/element/:id/elements

Search for multiple elements on the page, starting from the identified element. The located elements will be returned as a WebElement JSON objects. The table below lists the locator strategies that each server should support. Elements should be returned in the order located in the DOM.

Strategy Description

class	Returns all elements whose class name contains the search value; compound class names are not permitted.
css selector	Returns all elements matching a CSS selector.
id	Returns all elements whose ID attribute matches the search value.
name	Returns all elements whose NAME attribute matches the search value.
link text	Returns all anchor elements whose visible text matches the search value.
partial link text	Returns all anchor elements whose visible text partially matches the search value.
tag name	Returns all elements whose tag name matches the search value.
xpath	Returns all elements matching an XPath expression. The provided XPath expression must be applied to the server "as is"; if the expression is not relative to the element root, the server should not modify it. Consequently, an XPath query may return elements not contained in the root element's subtree.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

JSON Parameters:

using - {string} The locator strategy to use.

value - {string} The The search target.

Returns:

{Array.<{ELEMENT:string}>} A list of WebElement JSON objects for the located elements.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

XPathLookupError - If using XPath and the input expression is invalid.

/session/:sessionId/element/:id/click

POST /session/:sessionId/element/:id/click

Click on an element.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

[StaleElementReference](#) - If the element referenced by `:id` is no longer attached to the page's DOM.

[ElementNotVisible](#) - If the referenced element is not visible on the page (either is hidden by CSS, has 0-width, or has 0-height)

/session/:sessionId/element/:id/submit

POST /session/:sessionId/element/:id/submit

Submit a [FORM](#) element. The submit command may also be applied to any element that is a descendant of a [FORM](#) element.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:id` - ID of the element to route the command to.

Potential Errors:

[NoSuchWindow](#) - If the currently selected window has been closed.

[StaleElementReference](#) - If the element referenced by `:id` is no longer attached to the page's DOM.

/session/:sessionId/element/:id/text

GET /session/:sessionId/element/:id/text

Returns the visible text for the element.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:id` - ID of the element to route the command to.

Potential Errors:

[NoSuchWindow](#) - If the currently selected window has been closed.

[StaleElementReference](#) - If the element referenced by `:id` is no longer attached to the page's DOM.

/session/:sessionId/element/:id/value

POST /session/:sessionId/element/:id/value

Send a sequence of key strokes to an element.

Any UTF-8 character may be specified, however, if the server does not support native key events, it should simulate key strokes for a standard US keyboard layout. The Unicode [Private Use Area](#) code points, 0xE000-0xF8FF, are used to represent pressable, non-text keys (see table below).

Key	Code	Key	Code	Key	Code	Key	Code	Key	Code
NULL	U+E000	Space	U+E00D	Numpad 0	U+E01A	Multiply	U+E024	F1	U+E031
Cancel	U+E001	Pageup	U+E00E	Numpad 1	U+E01B	Add	U+E025	F2	U+E032
Help	U+E002	Pagedown	U+E00F	Numpad 2	U+E01C	Separator	U+E026	F3	U+E033
Back space	U+E003	End	U+E010	Numpad 3	U+E01D	Subtract	U+E027	F4	U+E034
Tab	U+E004	Home	U+E011	Numpad 4	U+E01E	Decimal	U+E028	F5	U+E035
Clear	U+E005	Left arrow	U+E012	Numpad 5	U+E01F	Divide	U+E029	F6	U+E036
Return ¹	U+E006	Right arrow	U+E014	Numpad 6	U+E020			F7	U+E037
Enter ¹	U+E007	Down arrow	U+E015	Numpad 7	U+E021			F8	U+E038
Shift	U+E008	Insert	U+E016	Numpad 8	U+E022			F9	U+E039
Control	U+E009	Delete	U+E017	Numpad 9	U+E023			F10	U+E03A
Alt	U+E00A	Semicolon	U+E018					F11	U+E03B
Pause	U+E00B	Equals	U+E019					F12	U+E03C
Escape	U+E00C							Command/Meta	U+E03D

¹The return key
is *not the same*
as the [enter key](#).

The server must process the key sequence as follows:

- Each key that appears on the keyboard without requiring modifiers are sent as a keydown followed by a key up.
- If the server does not support native events and must simulate key strokes with JavaScript, it must generate keydown, keypress, and keyup events, in that order. The keypress event should only be fired when the corresponding key is for a printable character.
- If a key requires a modifier key (e.g. "!" on a standard US keyboard), the sequence is: *modifier* down, *key* down, *key* up, *modifier* up, where *key* is the ideal unmodified key value (using the previous example, a "1").
- Modifier keys (Ctrl, Shift, Alt, and Command/Meta) are assumed to be "sticky"; each modifier should be held down (e.g. only a keydown event) until either the modifier is encountered again in the sequence, or the NULL (U+E000) key is encountered.
- Each key sequence is terminated with an implicit NULL key. Subsequently, all depressed modifier keys must be released (with corresponding keyup events) at the end of the sequence.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

JSON Parameters:

value - {Array.<string>} The sequence of keys to type. An array must be provided. The server should flatten the array items to a single string to be typed.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

ElementNotVisible - If the referenced element is not visible on the page (either is hidden by CSS, has 0-width, or has 0-height)

/session/:sessionId/keys

POST /session/:sessionId/keys

Send a sequence of key strokes to the active element. This command is similar to the [send keys](#) command in every aspect except the implicit termination: The modifiers are **not** released at the end of

the call. Rather, the state of the modifier keys is kept between calls, so mouse interactions can be performed while modifier keys are depressed.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`value - {Array.<string>}` The keys sequence to be sent. The sequence is defined in the [send keys](#) command.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/element/:id/name

GET /session/:sessionId/element/:id/name

Query for an element's tag name.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:id` - ID of the element to route the command to.

Returns:

`{string}` The element's tag name, as a lowercase string.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

`StaleElementReference` - If the element referenced by `:id` is no longer attached to the page's DOM.

/session/:sessionId/element/:id/clear

POST /session/:sessionId/element/:id/clear

Clear a `TEXTAREA` or `text INPUT` element's value.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:id` - ID of the element to route the command to.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

`StaleElementReference` - If the element referenced by `:id` is no longer attached to the page's DOM.

`ElementNotVisible` - If the referenced element is not visible on the page (either is hidden by CSS, has 0-width, or has 0-height)

`InvalidElementState` - If the referenced element is disabled.

/session/:sessionId/element/:id/selected

GET /session/:sessionId/element/:id/selected

Determine if an `OPTION` element, or an `INPUT` element of type `checkbox` or `radio` is currently selected.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:id` - ID of the element to route the command to.

Returns:

`{boolean}` Whether the element is selected.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

`StaleElementReference` - If the element referenced by `:id` is no longer attached to the page's DOM.

/session/:sessionId/element/:id/enabled

GET /session/:sessionId/element/:id/enabled

Determine if an element is currently enabled.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Returns:

{boolean} Whether the element is enabled.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

/session/:sessionId/element/:id/attribute/:name

GET /session/:sessionId/element/:id/attribute/:name

Get the value of an element's attribute.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Returns:

{string|null} The value of the attribute, or null if it is not set on the element.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

/session/:sessionId/element/:id>equals/:other

GET /session/:sessionId/element/:id>equals/:other

Test if two element IDs refer to the same DOM element.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

:other - ID of the element to compare against.

Returns:

{boolean} Whether the two IDs refer to the same element.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If either the element referred to by :id or :other is no longer attached to the page's DOM.

/session/:sessionId/element/:id/displayed

GET /session/:sessionId/element/:id/displayed

Determine if an element is currently displayed.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Returns:

{boolean} Whether the element is displayed.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

/session/:sessionId/element/:id/location

GET /session/:sessionId/element/:id/location

Determine an element's location on the page. The point (0, 0) refers to the upper-left corner of the page. The element's coordinates are returned as a JSON object with x and y properties.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Returns:

{x:number, y:number} The X and Y coordinates for the element on the page.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

/session/:sessionId/element/:id/location_in_view

GET /session/:sessionId/element/:id/location_in_view

Determine an element's location on the screen once it has been scrolled into view.

Note: This is considered an internal command and should **only** be used to determine an element's location for correctly generating native events.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Returns:

{x:number, y:number} The X and Y coordinates for the element.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

/session/:sessionId/element/:id/size

GET /session/:sessionId/element/:id/size

Determine an element's size in pixels. The size will be returned as a JSON object with width and height properties.

URL Parameters:

:sessionId - ID of the session to route the command to.

:id - ID of the element to route the command to.

Returns:

{width:number, height:number} The width and height of the element, in pixels.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

StaleElementReference - If the element referenced by :id is no longer attached to the page's DOM.

/session/:sessionId/element/:id/css/:propertyName

GET /session/:sessionId/element/:id/css/:propertyName

Query the value of an element's computed CSS property. The CSS property to query should be specified using the CSS property name, **not** the JavaScript property name (e.g. `background-color` instead of `backgroundColor`).

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:id` - ID of the element to route the command to.

Returns:

`{string}` The value of the specified CSS property.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

`StaleElementReference` - If the element referenced by `:id` is no longer attached to the page's DOM.

/session/:sessionId/orientation

GET /session/:sessionId/orientation

Get the current browser orientation. The server should return a valid orientation value as defined in

[ScreenOrientation](#): `{LANDSCAPE | PORTRAIT}`.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{string}` The current browser orientation corresponding to a value defined in [ScreenOrientation](#):
`{LANDSCAPE | PORTRAIT}`.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

POST /session/:sessionId/orientation

Set the browser orientation. The orientation should be specified as defined in [ScreenOrientation](#):
`{LANDSCAPE | PORTRAIT}`.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`orientation` - `{string}` The new browser orientation as defined in [ScreenOrientation](#):
`{LANDSCAPE | PORTRAIT}`.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/alert_text

GET /session/:sessionId/alert_text

Gets the text of the currently displayed JavaScript `alert()`, `confirm()`, or `prompt()` dialog.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{string}` The text of the currently displayed alert.

Potential Errors:

`NoAlertPresent` - If there is no alert displayed.

POST /session/:sessionId/alert_text

Sends keystrokes to a JavaScript `prompt()` dialog.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

text - {string} Keystrokes to send to the `prompt()` dialog.

Potential Errors:

NoAlertPresent - If there is no alert displayed.

/session/:sessionId/accept_alert

POST /session/:sessionId/accept_alert

Accepts the currently displayed alert dialog. Usually, this is equivalent to clicking on the 'OK' button in the dialog.

URL Parameters:

:sessionId - ID of the session to route the command to.

Potential Errors:

NoAlertPresent - If there is no alert displayed.

/session/:sessionId/dismiss_alert

POST /session/:sessionId/dismiss_alert

Dismisses the currently displayed alert dialog. For `confirm()` and `prompt()` dialogs, this is equivalent to clicking the 'Cancel' button. For `alert()` dialogs, this is equivalent to clicking the 'OK' button.

URL Parameters:

:sessionId - ID of the session to route the command to.

Potential Errors:

NoAlertPresent - If there is no alert displayed.

/session/:sessionId/moveto

POST /session/:sessionId/moveto

Move the mouse by an offset of the specified element. If no element is specified, the move is relative to the current mouse cursor. If an element is provided but no offset, the mouse will be moved to the center of the element. If the element is not visible, it will be scrolled into view.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

element - {string} Opaque ID assigned to the element to move to, as described in the WebElement JSON Object. If not specified or is null, the offset is relative to current position of the mouse.

xoffset - {number} X offset to move to, relative to the top-left corner of the element. If not specified, the mouse will move to the middle of the element.

yoffset - {number} Y offset to move to, relative to the top-left corner of the element. If not specified, the mouse will move to the middle of the element.

/session/:sessionId/click

POST /session/:sessionId/click

Click any mouse button (at the coordinates set by the last moveto command). Note that calling this command after calling buttondown and before calling button up (or any out-of-order interactions sequence) will yield undefined behaviour).

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

`button - {number}` Which button, enum: `{LEFT = 0, MIDDLE = 1 , RIGHT = 2}`. Defaults to the left mouse button if not specified.

/session/:sessionId/buttondown

POST /session/:sessionId/buttondown

Click and hold the left mouse button (at the coordinates set by the last moveto command). Note that the next mouse-related command that should follow is buttonup . Any other mouse command (such as click or another call to buttondown) will yield undefined behaviour.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`button - {number}` Which button, enum: `{LEFT = 0, MIDDLE = 1 , RIGHT = 2}`. Defaults to the left mouse button if not specified.

/session/:sessionId/buttonup

POST /session/:sessionId/buttonup

Releases the mouse button previously held (where the mouse is currently at). Must be called once for every buttondown command issued. See the note in click and buttondown about implications of out-of-order commands.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`button - {number}` Which button, enum: `{LEFT = 0, MIDDLE = 1 , RIGHT = 2}`. Defaults to the left mouse button if not specified.

/session/:sessionId/doubleclick

POST /session/:sessionId/doubleclick

Double-clicks at the current mouse coordinates (set by moveto).

URL Parameters:

`:sessionId` - ID of the session to route the command to.

/session/:sessionId/touch/click

POST /session/:sessionId/touch/click

Single tap on the touch enabled device.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`element - {string}` ID of the element to single tap on.

/session/:sessionId/touch/down

POST /session/:sessionId/touch/down

Finger down on the screen.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`x - {number}` X coordinate on the screen.

y - {number} Y coordinate on the screen.

/session/:sessionId/touch/up

POST /session/:sessionId/touch/up

Finger up on the screen.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

x - {number} X coordinate on the screen.

y - {number} Y coordinate on the screen.

session/:sessionId/touch/move

POST session/:sessionId/touch/move

Finger move on the screen.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

x - {number} X coordinate on the screen.

y - {number} Y coordinate on the screen.

session/:sessionId/touch/scroll

POST session/:sessionId/touch/scroll

Scroll on the touch screen using finger based motion events. Use this command to start scrolling at a particular screen location.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

element - {string} ID of the element where the scroll starts.

xoffset - {number} The x offset in pixels to scroll by.

yoffset - {number} The y offset in pixels to scroll by.

session/:sessionId/touch/scroll

POST session/:sessionId/touch/scroll

Scroll on the touch screen using finger based motion events. Use this command if you don't care where the scroll starts on the screen.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

xoffset - {number} The x offset in pixels to scrollby.

yoffset - {number} The y offset in pixels to scrollby.

session/:sessionId/touch/doubleclick

POST session/:sessionId/touch/doubleclick

Double tap on the touch screen using finger motion events.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`element` - `{string}` ID of the element to double tap on.

session/:sessionId/touch/longclick

POST session/:sessionId/touch/longclick

Long press on the touch screen using finger motion events.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`element` - `{string}` ID of the element to long press on.

session/:sessionId/touch/flick

POST session/:sessionId/touch/flick

Flick on the touch screen using finger motion events. This flick command starts at a particular screen location.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`element` - `{string}` ID of the element where the flick starts.

`xoffset` - `{number}` The x offset in pixels to flick by.

`yoffset` - `{number}` The y offset in pixels to flick by.

`speed` - `{number}` The speed in pixels per seconds.

session/:sessionId/touch/flick

POST session/:sessionId/touch/flick

Flick on the touch screen using finger motion events. Use this flick command if you don't care where the flick starts on the screen.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`xspeed` - `{number}` The x speed in pixels per second.

`yspeed` - `{number}` The y speed in pixels per second.

/session/:sessionId/location

GET /session/:sessionId/location

Get the current geo location.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{latitude: number, longitude: number, altitude: number}` The current geo location.

POST /session/:sessionId/location

Set the current geo location.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`location - {latitude: number, longitude: number, altitude: number}` The new location.

/session/:sessionId/local_storage

GET /session/:sessionId/local_storage

Get all keys of the storage.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Returns:

`{Array.<string>}` The list of keys.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

POST /session/:sessionId/local_storage

Set the storage item for the given key.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

JSON Parameters:

`key - {string}` The key to set.

`value - {string}` The value to set.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

DELETE /session/:sessionId/local_storage

Clear the storage.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/local_storage/key/:key

GET /session/:sessionId/local_storage/key/:key

Get the storage item for the given key.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:key` - The key to get.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

DELETE /session/:sessionId/local_storage/key/:key

Remove the storage item for the given key.

URL Parameters:

`:sessionId` - ID of the session to route the command to.

`:key` - The key to remove.

Potential Errors:

`NoSuchWindow` - If the currently selected window has been closed.

/session/:sessionId/local_storage/size

GET /session/:sessionId/local_storage/size

Get the number of items in the storage.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{number} The number of items in the storage.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

/session/:sessionId/session_storage

GET /session/:sessionId/session_storage

Get all keys of the storage.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{Array.<string>} The list of keys.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

POST /session/:sessionId/session_storage

Set the storage item for the given key.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

key - {string} The key to set.

value - {string} The value to set.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

DELETE /session/:sessionId/session_storage

Clear the storage.

URL Parameters:

:sessionId - ID of the session to route the command to.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

/session/:sessionId/session_storage/key/:key

GET /session/:sessionId/session_storage/key/:key

Get the storage item for the given key.

URL Parameters:

:sessionId - ID of the session to route the command to.

:key - The key to get.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

DELETE /session/:sessionId/session_storage/key/:key

Remove the storage item for the given key.

URL Parameters:

:sessionId - ID of the session to route the command to.

:key - The key to remove.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

/session/:sessionId/session_storage/size

GET /session/:sessionId/session_storage/size

Get the number of items in the storage.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{number} The number of items in the storage.

Potential Errors:

NoSuchWindow - If the currently selected window has been closed.

/session/:sessionId/log

POST /session/:sessionId/log

Get the log for a given log type. Log buffer is reset after each request.

URL Parameters:

:sessionId - ID of the session to route the command to.

JSON Parameters:

type - {string} The [log type](#). This must be provided.

Returns:

{Array.<object>} The list of [log entries](#).

/session/:sessionId/log/types

GET /session/:sessionId/log/types

Get available log types.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{Array.<string>} The list of available [log types](#).

/session/:sessionId/application_cache/status

GET /session/:sessionId/application_cache/status

Get the status of the html5 application cache.

URL Parameters:

:sessionId - ID of the session to route the command to.

Returns:

{number} Status code for application cache: {UNCACHED = 0, IDLE = 1, CHECKING = 2, DOWNLOADING = 3, UPDATE_READY = 4, OBSOLETE = 5}

7.6 - Legacy Selenium Desired Capabilities

These capabilities worked with the legacy JSON Wire Protocol

This documentation previously located [on the wiki](#)

See [JSON Wire Protocol](#) for common capabilities.

Remote Driver Specific

Key	Type	Description
webdriver.remote.sessionid	string	WebDriver session ID for the session. Readonly and only returned if the server implements a server-side webdriver-backed selenium.
webdriver.remote.quietExceptions	boolean	Disable automatic screenshot capture on exceptions. This is False by default.

Grid Specific

Key	Type	Description
path	string	Path to route request to, or maybe listen on.
seleniumProtocol	string	Which protocol to use. Accepted values: WebDriver, Selenium.
maxInstances	integer	Maximum number of instances to allow to connect to grid
environment	string	Possible duplicate of browserName? See RegistrationRequest

Selenium RC Specific

Key	Type	Description
proxy_pac	boolean	Legacy proxy mechanism. Do not use.
commandLineFlags	string	Flags to pass to browser command line.
executablePath	string	Path to browser executable.
timeoutInSeconds	long integer	Timeout to wait for the browser to launch, in seconds.
onlyProxySeleniumTraffic	boolean	Whether to only proxy selenium traffic. See browserlaunchers.Proxies
avoidProxy	boolean	??? See browserlaunchers.Proxies
proxyEverything	boolean	??? See browserlaunchers.Proxies
proxyRequired	boolean	??? See browserlaunchers.Proxies

Key	Type	Description
browserSideLog	boolean	??? See AbstractBrowserLauncher.
optionsSet	boolean	??? See BrowserOptions.
singleWindow	boolean	Whether to enable single window mode.
dontInjectRegex	javascript RegExp	Regular expression that proxy injection mode can use to know when to bypass injection. Ignored if not in proxy injection mode.
userJSInjection	boolean	??? Whether to inject user JS. Ignored if not in proxy injection mode.
userExtensions	string	Path to a JavaScript file that will be loaded into selenium.

Selenese-Backed-WebDriver specific

Key	Type	Description
selenium.server.url	string	URL of Selenium server to use, to back this WebDriver

Firefox specific

Key	Type	Description
captureNetworkTraffic	boolean	Whether to capture network traffic.
addCustomRequestHeaders	boolean	Whether to add custom request headers.
trustAllSSLCertificates	boolean	Whether to trust all SSL certificates.
changeMaxConnections	boolean	??? See FirefoxChromeLauncher.
firefoxProfileTemplate	string	??? See FirefoxChromeLauncher.
profile	string	??? See FirefoxChromeLauncher

FirefoxProfile settings

Preferences accepted by the FirefoxProfile with special meaning, in the WebDriver API:

Key	Type	Description
webdriver_accept_untrusted_certs	boolean	Whether to trust all SSL certificates. TODO: Maybe in some way different to the acceptSslCerts or trustAllSSLCertificates capabilities.
webdriver_assume_untrusted_issuer	boolean	Whether to trust all SSL certificate issuers. TODO: Maybe in some way different to the acceptSslCerts or trustAllSSLCertificates capabilities.

Key	Type	Description
webdriver.log.driver	string	Level at which to log FirefoxDriver logging statements to a temporary file, so that they can be retrieved by a getLogs command. Available options; DEBUG, INFO, WARNING, ERROR, OFF. Defaults to OFF.
webdriver.log.file	string	Path to file to which to copy firefoxdriver logging output. Defaults to no file (like /dev/null).
webdriver.load.strategy	string	Experimental API. Defines different strategies for how long to wait until a page is loaded. Values: unstable, conservative. Defaults to conservative.
webdriver_firefox_port	integer	Port to listen on for WebDriver commands. Defaults to 7055.

IE specific

Key	Type	Description
killProcessesByName	boolean	Whether to try to kill processes by name, instead (or addition) to killing processes we happen to have handles to.
honorSystemProxy	boolean	Whether to honor the system proxy.
ensureCleanSession	boolean	Whether to make sure the session has no cookies or temporary internet files on Windows. I believe this is passed to the IEDriver as well, but ignored by it.

Safari specific

Key	Type	Description
honorSystemProxy	boolean	Whether to honour the sysem proxy.
ensureCleanSession	boolean	Whether to make sure the session has no cookies, cache entries. And that any registry and proxy settings are restored after the session.

7.7 - Legacy developer documentation

Information of interest to developers of Selenium

7.7.1 - Crazy Fun Build Tool

The original Selenium Build Tool that grew from nothing to be extremely unwieldy, making it both crazy and “fun” to work with.

This documentation previously located [on the wiki](#)

WebDriver is a large project: if we tried to push everything into a single monolithic build file it eventually becomes unmanageable. We know this. We’ve tried it. So we broke the single Rakefile into a series of `build.desc` files. Each of these describe a part of the build.

Let’s take a look at a `build.desc` file. This is part of the [main test build.desc](#):

```
java_test(name = "single",
  srcs = [
    "SingleTestSuite.java",
  ],
  deps = [
    ":tests",
    "//java/server/src/org/openqa/selenium/server",
    "//java/client/test/org/openqa/selenium/v1:selenium-backed-webdriver-test",
    "//java/client/test/org/openqa/selenium/firefox:test",
  ])
])
```

Targets

This highlights most of the key concepts. Firstly, it declares **target**, in this case there is a single `java_test` target. Each target has a `name` attribute.

Target Names

The combination of the location of the “`build.desc`” file and the name are used to derive the rake tasks that are generated. All task names are prefixed with “`//`” followed by the path to the directory containing the “`build.desc`” file relative to the `Rakefile`, followed by a “`:`” and then the name of the target within the “`build.desc`”. An example makes this far clearer :)

The rake task generated by this example is `//java/client/test/org/openqa/selenium:single`

Short Target Names

As a shortcut, if a target is named after the directory containing the “`build.desc`” file, you can omit the part of the rake task name after the colon. In our example:

`//java/server/src/org/openqa/selenium/server` is the same as
`//java/server/src/org/openqa/selenium/server:server`.

Implicit Targets

Some build rules supply implicit targets, and provide related extensions to a normal build target. Examples include generating archives of source code, or running tests. These are declared by appending a further colon and the name of the implicit target to the full name of a build rule. In our example, you could run the tests using “`//java/client/test/org/openqa/selenium:single:run`”

Each of the rules described below have a list of implicit targets that are associated with them.

Outputs

Each target specified in a “build.desc” file produces one and only one output. This is important. Keep it in mind. Generally, all output files are placed in the “build” directory, relative to the rake task name. In our example, the output of “//java/org/openqa/selenium/server” would be found in “build/java/org/openqa/selenium/server.jar”. Build rules should output the names and locations of any files that they generate.

Dependencies

Take a look at the “deps” section of the “single” target above. The “:tests” is a reference to a target in the current “build.desc” file, in this case, it’s a “java_library” target immediately above. You’ll also see that there’s a reference to several full paths. For example

“//java/server/src/org/openqa/selenium/server” This refers to another target defined in a crazy fun build.desc file.

Browsers

The py_test and js_test rules have special handling for running the same tests in multiple browsers. Relevant browser-specific meta-data is held in rake-tasks/browsers.rb. The general way to use this is to append _ browsername to the target name; without the _ browsername suffix, the tests will be run for all browsers.

As an example, if we had a js_test rule //foo/bar, we would run its tests in firefox by running the target //foo/bar_ff:run or we would run in all available browsers by running the target //foo/bar:run

Build Targets

You can list all the build targets using the -T option. e.g.

```
./go -T
```

Being a brief description of the available targets that you can use.

Common Attributes

The following attributes are required for all build targets:

Attribute		
Name	Type	Meaning
name	string	Used to derive the rake target and (often) the name of the generated binary

The following attributes are commonly used:

Attribute Name	Type	Meaning
srcs	array	The raw source to be build for this target
deps	array	Prerequisites of this target

java_library

- **Output:** JAR file named after the “name” attribute if the “srcs” attribute is set.
- **Implicit Targets:** run (if “main” attribute specifies), project, project-srcs, uber, zip
- **Required Attributes:** “name” and at least one of “srcs” or “deps”.

Attribute		
Name	Type	Meaning

Attribute

Name	Type	Meaning
deps	array	As above
srcs	array	As above
resources	array	Any resources that should be copied into the jar file.
main	string	The full classname of the main class of the jar (used for creating executable jars)

java_test

- **Output:** JAR file named after the “name” attribute if the “srcs” attribute is set.
- **Implicit Targets:** run, project, project-srcs, uber, zip
- **Required Attributes:** “name” and at least one of “srcs” or “deps”.

Attribute Name **Type** **Meaning**

deps	array	As above.
srcs	array	As above.
resources	array	Any resources that should be copied into the jar file.
main	string	The alternative class to use for running these tests.
args	string	The argument line to pass to the main class
sysproperties	array	An array of maps containing System properties that should be set

js_deps

- **Output:** Marker file to indicate task is up to date.
- **Implicit Targets:** None
- **Required Attributes:** “name” and “srcs”

Attribute Name **Type** **Meaning**

name	string	As above
srcs	array	As above
deps	array	As above

js_binary

- **Output:** A monolithic JS file containing all dependencies and sources compiled using the closure compiler without optimizations.
- **Implicit Targets:** None
- **Required Attributes:** At least one of srcs or deps.

Attribute Name **Type** **Meaning**

name	string	As above

Attribute Name	Type	Meaning
srcs	array	As above
deps	array	As above

js_fragment

- **Output:** Source of an anonymous function representing the exported function, compiled by the closure compiler with all optimizations turned on.
- **Implicit Targets:** None
- **Required Attributes:** name, module, function, deps

Attribute Name	Type	Meaning
name	string	As above
module	string	The name of the module containing the function
function	string	The full name of the function to export
deps	array	As above

js_fragment_header

- **Output:** A C header file with all js_fragment dependencies declared as constants.
- **Implicit Targets:** None
- **Required Attributes:** name, deps

Attribute Name	Type	Meaning
name	string	As above
srcs	array	As above
deps	array	As above

js_test

- **Output:**
- **Implicit Targets:** _BROWSER:run, run
- **Required Attributes:** None.

Attribute		
Name	Type	Meaning
deps	array	As above.
srcs	array	As above.
path	string	The path at which to expect the test files to be hosted on the test server.
browsers	array	List of browsers, from rake_tasks/browsers.rb, to run the tests in. Will only attempt to run tests in those browsers which are available on the system. If absent, defaults to all browsers on the system.

Assuming browsers = ['ff', 'chrome'], for target //foo, the implicit targets: //foo_ff:run and //foo_chrome:run will be generated, which run the tests in each of those browsers, and the implicit target //foo:run will be generated, which runs the tests in both ff and chrome.

py_test

- **Output:** Creates the directory structure required to run the listed python tests.
- **Implicit Targets:** _BROWSER:run, run
- **Required Attributes:** name.

Attribute Name	Type	Meaning
deps	array	Other py_test rule(s), whose tests should also be run.
common_tests	array	Test file(s) to be run in all browsers. These tests will be passed through a template, with browser-specific substitutions, so that they are laid out properly for each browser in the python output file tree.
BROWSER_specific_tests	array	Test file(s) to be run only in browser BROWSER.
resources	array	Resources which should be copied to the python directory structure.
browsers	array	List of browsers, from rake_tasks/browsers.rb, to run the tests in. Will only attempt to run tests in those browsers which are available on the system. If absent, defaults to all browsers on the system.

Note: Every py_test invocation is performed in a new virtualenv.

rake_task

- **Output:** A crazy fun build rule that can be referred to “blow the escape” hatch and use ordinary rake targets.
- **Implicit Targets:** None
- **Required Attributes:** name, task_name, out.

Attribute Name	Type	Meaning
name	string	As above
task_name	string	The ordinary rake target to call
out	string	The file that is generated, relative to the Rakefile

gcc_library

- **Output:** Shared library file named after the “name” attribute if the “srcs” attribute is set.
- **Implicit Targets:** None.
- **Required Attributes:** “name” and “srcs”.

Attribute Name	Type	Meaning
srcs	array	As above
arch	string	“amd64” for 64-bit builds, “i386” for 32-bit builds.

Attribute Name	Type	Meaning
args	string	Arguments to the compiler (-I flags, for example).
link_args	string	Arguments to the linker (-l flags, for example)

Note: When building a new library for the first time, the build will succeed but copying to pre-built will fail with a similar message:

```
cp build/cpp/amd64/libimetesthandler64.so
go aborted!
can't convert nil into String
```

Solution: Copy the just-built library to the appropriate prebuilt folder (cpp/prebuilt/arch/).

7.7.2 - Buck Build Tool

Buck is a build tool from Facebook that we were working with to replace Crazy fun. We have since replaced it with [Bazel](#).

This documentation previously located [on the wiki](#)
You can read the documentation for the legacy [Crazy Fun Build tool](#).

Building Selenium with Buck

The easiest thing to do is to just run “./go”. The build process will download the right version of Buck for you so long as there’s no `.nobuckcheck` file in the root of the project. The download ends up in `buck-out/crazy-fun/HASH/buck.pex` where `HASH` is the value of the current buck version (given in the `.buckversion` file in the root of the project).

If you’d like to build and run our fork of Buck, then:

```
git clone https://github.com/SeleniumHQ/buck.git
cd buck && ant
export PATH=`pwd`/bin:$PATH
cd ~/src/selenium
buck build chrome firefox htmlunit remote leg-rc
buck test --all
```

Updating the `buck.pex`

Should you need to update the version of Buck that is downloaded:

- Checkout the source to Buck and build the PEX: `buck build --show-output buck`
- Figure out the git hash of the version you’ve just built. Normally that’ll be the HEAD of master. Put that full hash into the `.buckversion` of the main selenium project.
- Put the md5 hash of the PEX into the `.buckhash` file in the main selenium project.
- Create a new release of SeleniumHQ’s Buck fork on GitHub. The name is `buck-release-$VERSION`, where `$VERSION` is whatever’s in `.buckversion` in the main selenium project.
- Upload the PEX to the release, and make the release public.
- Commit the changes to the main selenium project and push them.

7.7.3 - Adding new drivers to Selenium 2 code

Instructions for how to create tests for new drivers for Selenium 2.

This documentation previously located [on the wiki](#) \

Introduction

WebDriver has a comprehensive suite of tests that describe the expected behavior of a new implementation. We'll assume that you're implementing the driver in Java for the sake of simplicity, but you can take a look at any of the existing implementations for how we handle more complex builds or other languages once you've read this.

Writing a New WebDriver Implementation

Create New Top-Level Directories

Create a new top-level folder, parallel to "common" and "firefox", named after your browser. In this, create a "src/java" and a "test/java" directory. It should be obvious what goes where.

Set Up a Test Suite

Copy one of the existing test suites to your test tree, and modify it for your new browser. This will probably cause you to modify the "Ignore.java" class, which is to be expected, and to add a holding class for your implementation in the source tree. You **must** include the "common" directory in order to pick up all the tests. For now, as long as nothing causes a fatal crash, leave the tests as they are.

Once you've added the test suite, add a "build.desc" CrazyFunBuild file in the top level of your project. Model it after the one in the "htmlunit" directory. You should then be able to run your tests from the command line using the "go" script.

At this point, we expect total and catastrophic failure when tests are being run.

Start Implementing

If your browser runs out of process, it is *strongly encouraged* to make use of the JsonWireProtocol. This will make the client-side (the APIs that users use) relatively cheap to implement, and means that you get Java, C#, Ruby and Python support for significantly less effort since you can extend the remote client.

Implementation Tips

Where to Start

As mentioned, has a suite of tests. The suggested order to make these pass is roughly:

1. ElementFindingTest — needed because element location is key
2. PageLoadingTest
3. ChildrenFindingTest — more finding elements
4. FormHandlingTest
5. FrameSwitchingTest
6. ExecutingJavascriptTest
7. JavascriptEnabledDriverTest

At this point, you'll have a reasonably complete working driver. After that, it's probably best to get the user interactions correct:

1. CorrectEventFiringTest
2. TypingTest

Before spelunking into the cutting-edge stuff:

1. AlertsTest

It's not necessary to get every test working in a class before moving on. I tend to go as far down a class as I can, and then switch to the next class on the list when the going gets tough. This allows you to maintain reasonable velocity and still cover the basics.

Running a Single Test

It's far from ideal, but the method we use is to modify the SingleTestSuite class in the common project, and then modify the module it's run from via the IDE's UI (that is, just go into the launch configuration (in IDEA) and modify the module used: don't move the file!) This class should be self-explanatory.

Ignoring Tests

At some point you'll want to stop running tests on an ad-hoc basis and make use of a continuous build product to ensure that you're not introducing regressions. At this point, the process is to run the tests from the command line. This will generate a list of failing tests. Go through each of these tests and add or modify the "@Ignore" associated with the test. Re-run the tests. It may take a few iterations, but your build will eventually go green. Nice.

The build makes use of ant behind the scenes and stores logs in "build/build_log.xml" and the test logs in "build/test_logs"

7.7.4 - Selenium's Continuous Integration Implementation

We used to have a Jenkins CI tool that executed unit tests and ran integration tests on Sauce Labs. We moved all of the tests to Travis, and now execute everything with Github Actions.

This documentation previously located [on the wiki](#)

General architecture

We have a number of Google Compute Engine virtual machines running Ubuntu, currently hosted at `{0..29}.ci.seleniumhq.org` - they have publicly addressable DNS set up to point [ab](#).
`{0..29}.ci.seleniumhq.org` pointing at them as well, so that cookie tests can do subdomain lookups.

One of these machines, `ci.seleniumhq.org`, is running jenkins. If you want a login on jenkins, get in touch with juangj. The Build All Java job polls SCM for changes, and does the following:

- Does a clean build of the ‘release’ target, any tests which are going to be run, and any artifacts (e.g. the IEDriverServer executable) which will be required to run those tests
- Tars up the entire built working directory and publishes it to [http://ci.seleniumhq.org/selenium-trunk-r\\${REVISION}.tgz](http://ci.seleniumhq.org/selenium-trunk-r${REVISION}.tgz) - this is used later by test runs
- Publishes the selenium-server-standalone jar to [http://ci.seleniumhq.org/selenium-server-standalone-r\\${REVISION}.tgz](http://ci.seleniumhq.org/selenium-server-standalone-r${REVISION}.tgz) - this is copied down directly by [SauceLabs](#) when running tests.
- Zips up the IEDriverServer and publishes it to [http://ci.seleniumhq.org/IEDriverServer-Win32-r\\${REVISION}.zip](http://ci.seleniumhq.org/IEDriverServer-Win32-r${REVISION}.zip) - this is copied down directly by [SauceLabs](#) to run IE tests This machine is backed by a 1TB persistent disk, which can hold many build artifacts, but they should be cleared out occasionally (particularly when moving disk between zones).

When this build is successful, it triggers downstream builds for each OS/browser/test combination we care about. It also triggers a downstream clean build to ensure our maven poms are still in order (“Maven build”).

Apart from “Maven build” which runs on the same build node as the compile (a beefy, 8-CPU machine with 32GB RAM), all downstream builds run on separate build nodes.

The downstream builds are configured using environment variables, as per the [SauceDriver](#) class. The downstream builds download the selenium-trunk tar from the build master, and then run tests (which should already have been compiled by the Build All Java rule). Two of these downstream builds are special; “HtmlUnit Java Tests” and “Small Tests” just run headless locally. The others use [SauceLabs](#).

A note about networking: The build nodes are set up on an internal network 10.1.0/24, so network communication between them is incredibly fast and free.

When a non-headless browser test is running, the test-file servlet hosts the test files on ports determined by an environment variable (231\${EXECUTOR_NUMBER} and 241\${EXECUTOR_NUMBER} - EXECUTOR_NUMBER is currently always equal to 0). The hostname used by tests is set by an environment variable (`ab.${NODE_NAME}.ci.seleniumhq.org` where NODE_NAME in {0..29}). A browser is requested from [SauceLabs](#) using our credentials (stored in jenkins-wide environment variables, set on the System Configuration page). Jenkins is currently set to run three test-classes at a time in parallel, per test run, again on the System Configuration page.

The tests are run, and the results get notified to IRC.

Thanks to [SauceLabs](#) and [Google](#) for donating the infrastructure to run all of these tests.

FAQ

I want to run my tests on Sauce like Jenkins does (my tests are failing on CI, but work fine on my machine!)

See the [SauceDriver](#) page

I want to add a new browser (Firefox has released a new version!)

Jenkins doesn't have a great concept of templates. I (dawagner) have some selenium scripts which automate the UI of Jenkins, to create new jobs using canned settings. If you want to do it manually, here are roughly the steps to take:

- Find the most similar config(s) you want to copy. If it's a new Firefox release, find the latest firefox (which should have roughly 6 builds associated with it: Javascript + Java {Windows,Linux} **{Native,Synthesized})
- For each of those builds, create a New Job (menu on the left hand side of the home page, when logged in)
- Name the job in the style of the others. Select "Copy existing job", and enter the job you're copying.
- Scroll through the job it's pre-populated. Replace the version numbers, browser name, and any other details that need replacing. For firefox updates, there are currently three places you should be replacing the number (the "browser_version" field, and two in the Build Execute Shell)
- Save
- Go to the Build All Java task, configure it, add your new build to the "Projects to build" field where there are many others listed.**

If it's a firefox update, you probably also want to delete an existing build.

7.7.5 - Google Summer of Code 2011

Selenium encouraged users to take advantage of this program.

This documentation previously located [on the wiki](#)

What is Google Summer of Code?

Since 2005, Google has administered [Google Summer of Code Program](#) to encourage student participation in open source development. The program has several goals:

- Inspire young developers to begin participating in open source development
- Provide students in Computer Science and related fields the opportunity to do work related to their academic pursuits during the summer
- Give students more exposure to real-world software development scenarios (e.g., distributed development, software licensing questions, mailing-list etiquette, etc.)
- Get more open source code created and released for the benefit of all
- Help open source projects identify and bring in new developers and committers

Google will pay successful student contributors a \$5000 USD stipend, enabling them to focus on their coding projects for three months. The deadline for application is **April 8, 2011**.

When participating in the Selenium - Google Summer of Code program, students will learn that testing, and building automated testing tools, can be both fun and an integral part of delivering high quality software. The collaborative effort with Selenium contributors can provide you with a new toolset to develop and document a set of components used by thousands of people. You will gain valuable professional experience towards your career development and ultimately help drive higher quality web applications everywhere.

Please [Email](#) questions to GSoC coordinator Adam Goucher

Student Eligibility

- 18 years of age or older by April 26, 2010
- Currently enrolled in an accredited institution such as colleges, university, master programs, PhD programs and undergraduate programs
- Residents and nationals of countries other than Iran, Syria, Cuba, Sudan, North Korea and Myanmar (Burma) with whom we are prohibited by U.S. law from engaging in commerce
- Strong skills in some or more of the following: Web Application Development, JavaScript, Python, Flash, iPhone / Android
- Self-directed, resourceful, responsible, communicative
- Ability to work full-time from May 24 – August 20, 2010 (students residing in SF Bay Area may have an opportunity to work on-site from time to time with some of our mentors)
- More info on Student Eligibility can be found [here](#)

If you meet the above requirements, we'd love to have you apply to Selenium for this year's Google Summer of Code.

Next steps and deadlines

1. Read [Expectations](#) to understand what is expected of you.
2. Read [Applications](#) to find out what to put on your application.
3. Take a look at the [Project Ideas](#). If any interest you, feel free to contact the proposer for details. You can also discuss your own project ideas with the people mentioned or talk about them on our [developer mailing list](#) or on IRC #selenium on freenode
4. [Submit your application directly to Google](#) before April 8, 2011. You can modify your application with your mentors' feedback after your initial submission, the final version of your application is

due April 23, 2011.

5. Selenium GSoC team will finish reviewing applications and match students with mentors by April 23, 2011.
6. Google announces accepted students on April 26, 2011.

Please [Email](#) questions to GSoC coordinator Adam Goucher

Project Ideas

These are project ideas proposed by mentors. Please send a post to the [developer mailing list](#) if you are interested in it or [email](#) GSoC coordinator Adam Goucher.

A Scriptable Proxy

Mentor Patrick Lightbody(?)

Difficulty

<unknown>

Description Selenium is a browser control framework, but sometimes you want to do things to/with the traffic generates. The 'right' way to do this is to put a proxy in the middle and use its API to do get / fiddle with the traffic information. This project extends the BrowserMob proxy to add the APIs that users of Selenium would need.

Tags Se-RC, Se2

Image Based Locators

Mentor

<unknown>

Difficulty

<unknown>

Description Sikuli gets a lot of play for its ability to interact with items on the page based on Images. This project would add Image Based Locators to the list of available ones.

Tags Se-RC, Se2, Se-IDE

Selenese Runner

Mentor Adam Goucher

Difficulty

<unknown>

Description It is possible to run Selenese scripts outside of Se-IDE with the -htmlSuite option on the server. There are a number of downsides to this, like the need to start/stop the server constantly. This project will create a standalone 'runner' for Selenese scripts to interact with the server – and remove the related code from the server.

Tags Se-RC, Se, Se-IDE

Perspective mentors

It's not too late to apply to be a mentor, if you are interested, please add your project idea here and discuss logistics with [Adam Goucher](#)

Expectations

Summary

This page covers, in detail, the expectations for Google Summer of Code students in regards to communication. This is useful for Selenium projects which haven't codified their expectations—they can point to this document and use it as is.

The Google Summer of Code coding period is very short. On top of that, many students haven't done a lot of real-world development/engineering work previously; one of the primary purposes of the program is to introduce students to F/OSS and real-world development scenarios. On top of that, most mentors and students are in different locations—so face-to-face time is difficult. Because of this, it's vitally important to the success of the GSoC project for all expectations to be specified before students begin coding on May 26th. This should be the first step in a long series of frequent communication between student and mentor(s).

This document walks through various expectations for students and mentors, as well as addressing various ways to communicate effectively.

40 hour work week

Students are expected to work at least 40 hours a week on their GSoC project. This is essentially a full-time job.

The benefits for the GSoC project are huge:

- the chance to become part of a project community over the long term—this can lead to involvement with other projects, social network, good friends, valuable resources, ...
- the chance to work with real developers on a real project
- the end result of the student's project can be used for resume material that is available for all future employers to see

The final point is an important one for a beginning developer. Employers greatly appreciate having a referenceable body of work when looking at potential employees. Your code says more about your abilities than any amount of algorithms on a whiteboard can.

And of course, the program will provide you with 5000 USD in income and a really cool t-shirt.

Some GSoC students have become prominent technology bloggers, committers to open source projects, speakers at conferences, mentors for other students, and more...

Self-motivation and steady schedule

The student is expected to be self-motivated. The mentor may push the student to excel, but if the student is not self-motivated to work, then the student probably won't get much out of participating. The student should schedule time to work on the project each day and keep to a regular schedule. It's not acceptable to fiddle around for days on end and then pull an all-nighter just before deadlines. It will show in your code.

Regular Weekly Meeting and Frequent Communication with mentor

Regular weekly meeting with your mentor is a must. The planned meeting should cover:

- what you're currently working on
- how far you've gotten
- how you're implementing it
- what you plan on working on next
- what issues have come up
- what you did to get around them
- what's blocking you if you're stuck
- code review, when applicable

The mentor is one of the most valuable resources for GSoC projects. The mentor is both a solid developer and a solid engineer. The mentor likely has worked on the project for long enough to know the history of decisions, how things are architected, the other people involved, the process for doing things, and all other cultural lore that will help the student be most successful.

Before the GSoC project starts, the mentor and student should iron out answers to the following questions:

1. When is the regular, scheduled communication scheduled? Weekly? Every two days? Mondays, Wednesdays, Fridays?
2. What is the best medium to use for regular, scheduled communication? VOIP? Telephone call? Face-to-face?
3. What is the best medium to use for non-scheduled communication? Email? Instant messenger?

DO:

- be considerate of your time and your mentor's time and plan for your regular meeting
 - Consider emailing the answers to the above agenda ahead of time so you can spend your time productively on coming up with solutions, code reviews, and planning for the next milestones.
- talk to your mentors and developers on the mailing list / IRC frequently, outside of your planned meeting
 - your mentor is not the only person that can help you out and keep you stay on track, Selenium has a nice community and you will learn a lot from the other people as well.
- let your mentor know what your schedule is
 - Are you going on vacation, moving, writing papers for class? If your mentor doesn't know where you'll be or to expect a lag in your productivity, your mentor can't help you course correct or plan accordingly.

AVOID:

- going for more than a week without communicating with your mentor
 - The project timeline doesn't allow for unplanned gaps in communication.

Version control

Students should be using version control for their project.

DO:

- commit-early/commit-often
 - This allows issues to be caught quickly and prevents the dreaded one-massive-commit-before-deadline.
- use quality commit messages

Bad examples: Fixed a bug. Tweaks.

Good examples: Fixed a memory leak where the thingamajig wasn't getting freed after the parent doohicky was freed. Fixed bug #902 (on Google Code) by changing the comparison used for duplicate removal. Implemented Joe's good idea about rendering in a separate buffer and then swapping the buffer in after rendering is complete. Improved HTML by simplifying tables.

- refer to specific bug numbers, links, and issues as much as possible
- The history in version control is frequently the best timeline log of what happened, why, and who did it.

AVOID:

- checking in multiple non-related changes in one big commit
 - If something is bad about one of the changes and someone needs to roll it back, it's more difficult to do so.
- checking in changes that haven't been tested

Communication with project

Most F/OSS projects have mailing lists for project members and the community and/or have IRC channels to communicate. These communication channels allow the student to keep in touch with the other project members and are an incredibly valuable resource. Other members of the project may be better versed in various parts of the project, they may provide a fallback if the mentor isn't available, and they may be a good sounding board for figuring out the specific behavior for features. You are assigned a mentor, but the whole community is there to help you learn. Make use of all the resources at your disposal.

Shyness is a common problem for students who are new to open source development. At the beginning of the project, the student is encouraged to send a “Hello! I’m ... and I’m working on a GSoC project on ... and here’s a link to the proposal.” email to project mailing lists and encouraged to log in and say “hi” on IRC. Break the ice early—it makes the rest of the project easier. If you don’t know where you announce yourself, ask your mentor.

Project mailing lists

Mailing lists are a great way to work out feature specifications and expected behavior.

Often mailing lists are archived and the archives are a rich source of information regarding prior discussions, decisions, and technical errata.

DO:

- search through the archives for answers before asking on the list
- be courteous at all times
- be specific
 - Cite data, references, and use links wherever possible when discussing technical things.
- be patient
 - Don’t expect an answer within minutes or hours; people often read their mailing-list messages once per day.

AVOID:

- being rude
 - Since most mailing lists are archived or recorded, it’s likely anything you say will be available for everyone to see forever; exercise good manners in all aspects of life.
- saying things with all capital letters and excessive punctuation
 - This is perceived as shouting
- getting into heated arguments
 - If someone insults you, it’s best to ignore it.

IRC

Most F/OSS projects have an IRC channel and some have more than one. People from the project and its community “hang out” on these channels and talk about various things. Some projects have regularly scheduled meetings to cover the status of the project, how development is going, status of major blocking bugs, map out future plans, ...

If the project has an IRC channel, it’s a good idea to hang out there. This allows the student to interact with the community and also a forum for working out problems and ideas in real time.

DO:

- **hang out on the project IRC channels when you’re working on the project**
- **take time to interact with people who are on the IRC channel** This builds community and it’s easier to get help from people who are familiar with you than people who aren’t.

AVOID:

- **saying things with all capital letters and excessive punctuation** This is perceived as shouting.
- **poor grammar** It makes it harder for other people to understand what you’re trying to say.
- **being rude**

We're all real people with real feelings and if you're rude it's likely people will interact with you and help you less; also it's not uncommon for IRC history to be recorded and archived for all to see forever.

See:

- <http://www.linuxchix.org/irc-basics.html>
- http://en.wikipedia.org/wiki/Internet_Relay_Chat

Design documents

It's a good idea for the student to maintain design documents during the course of the GSoC project. These design documents should cover:

1. the project plan, with additional detail to flesh out the original program application
2. deviations from the project plan and how and why the original design plan changed
3. any issues that could not be worked out or overcome
4. possible future directions
5. any resources used or relevant specifications

The student and mentor should work out what design documents should be maintained during the course of the GSoC.

One thing to note is that the student shouldn't spend all his/her time doing design documents. It's important to keep track of the design, but it's also important to get some code done. The mentor should be able to help the student strike a balance between these two goals.

Blogging

Students should get in the habit of blogging about his/her work at least once every two weeks. Historically, students who do learn much faster, are more productive, and develop a stronger tie to the community. Some have gone on to become contributors, others have given talks / presentations at conferences. How would you like to see your career grow?

Application

Evaluation Criteria

We recognized that very few students have exposures to Selenium during their studies and will therefore evaluate you based on your:

- ability to think, learn, and reason
- talents (what have you accomplished thus far, programming and otherwise)
- attitude, communications, ability to work well with the community and your mentor
- availability and commitment to succeeding in GSoC and potential for continuous involvement with community
- in a nutshell, what makes you a good person to lead that project initiative :-)

As long as you get your application in before April 9, you will have until April 18 to fine-tune your proposal with our mentors.

Preparing Your Proposal

Here are some questions to get you started. You don't have to follow it and your application will still be considered, but it's a good place to start.

Feel free to include anything else you feel is important. One liner answers are not likely going to be considered. In the meanwhile, do feel free to introduce yourself to the community and discuss your project proposal by writing to our developer mailing list.

General Questions:

1. Give us a short introduction of yourself.

2. Email address and phone number we can reach you at.
3. What are you studying? What year of study will you be in September 2010?
4. How much time can you devote to Summer of code? What else are you doing this summer?

Your Experience:

1. How did you get started with programming? How long have you been doing it? Why do you love it? Any personal projects you can show us? Have you participated in coding contests / taught / mentor other students?
2. What are your programming interests? Are you a C guy - do you like to get down and dirty with the linux kernel? Do you know more about Java than your peers? Or are you more of a python/ruby person? What about JavaScript? You know, what's your style?
3. Have you worked for a software company as a programmer before?
4. Have you worked on an open source project before? Which ones? Describe your participation
5. Do you have a blog? A resume?
6. What makes you a good person for Google Summer of Code? What do you want to get out of it?

Project Questions:

1. What idea did you choose?
2. Elaborate on the idea and describe what you would like to accomplish during the summer. This question is especially important if you have your own idea instead of picking one from our list, as we want to have a good understanding of what you're proposing so we can help you take the idea forward.
3. Give us a brief time-line of the project for the things you'd like to accomplish. It's OK to include thinking time ("investigation") in your work schedule. Work should include:
 - investigation
 - programming
 - documentation
 - dissemination
4. How do you plan to test your code? What version control and build systems do you plan to use?
5. If your project is very successful, do you wish to contribute to it further once Google Summer of Code is complete?

Sample Proposal Outline

A good proposal will have the following component:

- **Name and Contact Information.** Include email, phone(s), IM, Skype, etc.
- **Title.** One liner on the goal to your project.
- **Synopsis.** Short summary, what would your project do?
- **Benefits to Community.** Why would Google and Selenium be proud to sponsor this work? How would open source or society as a whole benefit?
- **Deliverables.** We want to know that you have a plan and that at the end of the summer, something gets delivered. :-) Give a brief, clear work breakdown structure with milestones and deadlines. Make sure to label deliverables as optional or required. You may want to start by producing some kind of whitepaper, or planning the project in traditional Software Engineering style. It's OK to include thinking time ("investigation") in your work schedule. Work should include:
 - investigation
 - programming
 - documentation
 - dissemination
- **Description.** A small list of project details. Your mentors can give you some guidance on this, but start by letting us know you are thinking :-)
 - rough architecture
 - links to parallel projects that you may get insights from
 - what version control and build system do you plan to use
 - how do you plan to test
 - best practices to get your code accepted, etc.

- **Bio.** Who are you? What makes you the best person to work on this project?
 - Summarize your education, work, and open source experience.
 - List your skills and give evidence of your qualifications. Convince us that you can do the work.
 - Any published papers, successful open source projects, etc? Please tell us!
 - Please list any non-Summer-of-Code plans you have for the Summer, especially employment and class-taking. Be specific about schedules and time commitments.

7.7.6 - Developer Tips

Details on how to execute Selenium Test Suite with Crazy Fun.

This documentation previously located [on the wiki](#)

Running an Individual Test

When developing WebDriver, it is common to want to run a single test rather than the entire test suite for a particular driver.

You can run all the tests in a given test class this way:

```
./go test_firefox onlyRun=CombinedInputActionsTest
```

You can also run a single test directly from the command line by typing:

```
./go test_firefox method=foo
```

Not Halting On Errors Or Failures

The test suite will halt on errors and failures by default. You can disable this behaviour by setting the `haltonerror` or `haltonfailure` environmental variables to `0`.

Reviewing the Logs For the Tests

When you run the tests, the test results don't appear on the screen. They are written to the `./build/test_logs` folder. A pair of files are written. Their names are relatively consistent and include the details of the tests which were run. The pair comprise a txt file and an xml file. The xml file contains more information about the runtime environment such as the path, Ant version, etc. These files are overwritten the next time the same test target is executed so you may want to archive results if they're important to you.

Using Rake

Rake is very similar to using other build tools such as "make" or "ant". You can specify a "target" to run by adding it as a parameter, and you can add more than one target at a time. Note that since WebDriver does not rely on ruby being installed and uses JRuby, rake should **not** be involved directly - use the `go` script instead. For example, in order to clean the build and then build and run the HtmlUnitDriver tests:

```
./go clean test_htmlunit
```

The default target that's used will compile the code and run all the tests. More interesting targets are:

Target	Description
clean	Delete the contents of the build directory, removing all compiled artifacts
test	Compile the dependencies of and run all the tests for the HtmlUnitDriver, FirefoxDriver, and InternetExplorerDriver as well as the support library's tests
firefox	Compile the FirefoxDriver

Target	Description
htmlunit	Compile the HtmlUnitDriver
ie	Compile the InternetExplorerDriver. This won't compile the C++ on a non-Windows system, but will always compile the Java, no matter which OS you happen to be using
support	Guess what this does :)
test_htmlunit	Compile the dependencies and then run the tests for the HtmlUnitDriver. The same "test_x" pattern can be followed for all the compilation targets in this table.

Running a remote Debugger with Java tests

You can run the tests in debug mode and wait for a remote java listener (which one would setup in eclipse or intellij).

```
./go debug=true suspend=true test_firefox
```

Debugging the Firefox Driver

Getting output from the Firefox process itself

This is usually useful to debug issues with Firefox starting up. The Java system property `webdriver.firefox.logfile` will instruct the FirefoxDriver to redirect the output to a file:

```
java -Dwebdriver.firefox.logfile=/dev/stdout -cp selenium-2.jar <sometest>
```

Outputting to the Error Console

A common technique used for debugging of the Firefox driver extension is debug statements. The two following methods can be used from almost any Javascript code inside the extension:

- `Logger.dumpn()` - Logs a string into console (and converts arguments to strings). For example:
`Logger.dumpn("Found element: " + node).`
- `Logger.dump()` - Gets a single argument, an object, and dumps its entire contents: implemented interfaces, data fields, methods, etc.

Getting output from the error console to a file

To see output generated using the `Logger` utility, one has to open up Firefox's error console - difficult or simply impossible on remote machines. Fortunately, there's a way to get the contents of the output dumped to a file:

```
FirefoxProfile p = new FirefoxProfile();
p.setPreference("webdriver.log.file", "/tmp/firefox_console");
WebDriver driver = new FirefoxDriver(p);
...
```

The `webdriver.log.file` preference will instruct the `Logger` to dump all contents of the console to the specified file. `webdriver.log.file`

Getting even more output to the command line

When suspecting additional logging from Firefox could be beneficial, one can crank debugging level all the way up:

```
export NSPR_LOG_MODULES=all:3
```

Setting this environment variable will cause Firefox to log additional messages to the console. Use this environment variable together with `webdriver.firefox.logfile` to get a hold of Firefox's output to the console.

Debugging the Internet Explorer Driver

In order to get detailed information from IEDriverServer.exe you can run tests with the option `devMode=true`, this option will set logging level to DEBUG and redirect log output to the file `iedriver.log`

```
./go test_ie devMode=true
```

Adding a test

Most of WebDriver's test cases live under `java/client/test/org/openqa/selenium`. For example, to demonstrate an issue with clicking on elements, a test case should be added to `ClickTest`. The test cases already have a driver instance - no need to create one. The test use pages that are served by an in-process server, served from `common/src/web`. Their URLs are provided by the `Pages` class, so when adding a page and add it to the `Pages` class as well.

Manually interacting with `RemoteWebDriverServer`

We can use a web browser or tools such as telnet to interact with a `RemoteWebDriverServer` e.g. to debug the JSON protocol. Here's a simple example of checking the status of a server installed on the local machine

In a web browser

```
http://localhost:8080/wd/hub/status/
```

In telnet

```
telnet localhost 8080  
GET /wd/hub/status/ HTTP/1.0
```

On Macs and Unix in general try `curl`

```
curl http://localhost:8080/wd/hub/status
```

And on linux `wget`

```
wget http://localhost:8080/wd/hub/status
```

In all these cases the `RemoteWebDriverServer` should respond with

```
{status:0}
```

7.7.7 - Snapshot of Roadmaps for Selenium Releases

The list of plans and things to accomplish before a release

Preparation for Selenium 2

Date unknown This documentation previously located [on the wiki](#)

The following issues need to be resolved before the final release:

Issue	Summary	HtmlUnitDriver Progress	FirefoxDriver Progress	InternetExplorerDriver Progress	ChromeDriver Progress
27	Handle alerts in Javascript-enabled browsers	n/a	Started	Started	Not Started
32	User guide	Started			
34	Support HTTP Basic and Digest Authentication	Not Started			
35	Selenium emulation	Done for Java and C#			
36	Support for drag and drop behaviour	n/a	Done	Done	Started
none	Example tests	Not Started			

A final release will be made once these are implemented in Firefox, IE and at least one webkit-based browser.

The Future

The following are also planned:

- **JsonWireProtocol** — The formalisation of the current RemoteWebDriver wire protocol in [JSON](#).

Preparation for Selenium 3

As of Mar 16, 2015 This documentation previously located [on the wiki](#)

User Visible Changes

- Migrate all drivers to use the status strings rather than status codes in responses
- Update client bindings to also cope with that
- Write a new runner for the html-suite tests
- Segment the build to remove RC

Clean up

- Using WebDriver after quit() should be an IllegalStateException
- Actions to have a single end point
- Capabilities to be the same as the spec
- Multiple calls to WebDriver.quit() should still be safe.
- Clean up WebDriver constructors, pulling heavy initialization logic into a Builder class
- Migrate to Netty or webbit server
- Delete unnecessary cruft
- Land a cleaner end point for the rc emulation

Preparation for Selenium 4

This documentation previously located [on the wiki](#) As of April 12, 2017

- Finish the [W3C WebDriver Spec](#)
- Implement the local end requirements for the spec in selenium
- Implement protocol conversion in the standalone server
- Ship 4.0
-

8 - About this documentation

These docs, like the code itself, are maintained 100% by volunteers within the Selenium community. Many have been using it since its inception, but many more have only been using it for a short while, and have given their time to help improve the onboarding experience for new users.

If there is an issue with the documentation, we want to know! The best way to communicate an issue is to visit <https://github.com/seleniumhq/seleniumhq.github.io/issues> and search to see whether or not the issue has been filed already. If not, feel free to open one!

Many members of the community are present at the #selenium Libera chat at [Libera.chat](#). Feel free to drop in and ask questions and if you get help which you think could be of use within these documents, be sure to add your contribution! We can update these documents, but it is much easier for everyone when we get contributions from outside the normal committers.

8.1 - Copyright and attributions

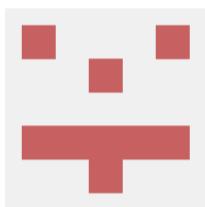
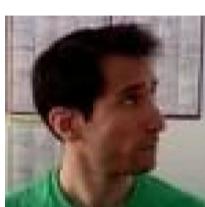
Copyright, contributions and all attributions for the different projects under the Selenium umbrella.

The Documentation of Selenium

Every effort has been made to make this documentation as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as-is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book. No patent liability is assumed with respect to the use of the information contained herein.

Attributions

Selenium Main Repository

 @shs96c 5090 commits	 @barancev 3352 commits
 @jimevans 2458 commits	 @jleyba 1464 commits
 @jarib 1299 commits	 @AutomatedTester 1197 commits
 @dfabulich 1175 commits	 @illicitonion 1162 commits
 @titusfortner 759 commits	 @diemol 742 commits
 @lukeis 599 commits	 @eranmes 473 commits
 @p0deje 434 commits	 @mdub 326 commits
 @andreastt 289 commits	 @krosenvold 225 commits



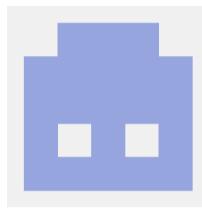
[@pujagani](#) 219 commits



[@hugs](#) 205 commits



[@davehunt](#) 200 commits



[@hbchai](#) 191 commits



[@lmtierney](#) 179 commits



[@freynaud](#) 138 commits



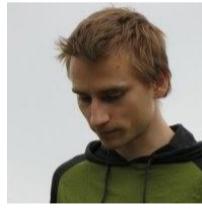
[@samitbadle](#) 137 commits



[@nirvdrum](#) 133 commits



[@harsha509](#) 124 commits



[@sevaseva](#) 115 commits



[@gigix](#) 109 commits



[@juangj](#) 108 commits



[@aslakhellesoy](#) 94 commits



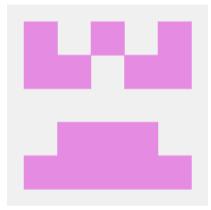
[@alex-savchuk](#) 90 commits



[@andrashatvani](#) 66 commits



[@ajayk](#) 63 commits



[@twalpole](#) 49 commits



[@asashour](#) 48 commits



[@mikemelia](#) 46 commits



[@tebeka](#) 44 commits



[@symonk](#) 43 commits



[@raju249](#) 42 commits



[@santiycr](#) 41 commits



[@luke-hill](#) 40 commits



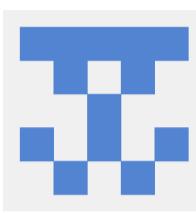
[@mach6](#) 36 commits



[@ddavison](#) 32 commits



[@joshbruning](#) 30 commits



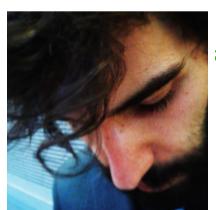
[@valfirst](#) 30 commits



[@mikebroberts](#) 28 commits



[@JohnChen0](#) 26 commits



[@alb-i986](#) 22 commits



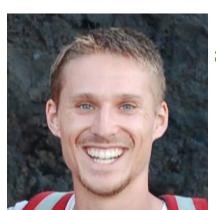
[@bret](#) 20 commits



[@cgoldberg](#) 20 commits



[@corevo](#) 20 commits



[@Stuk](#) 19 commits



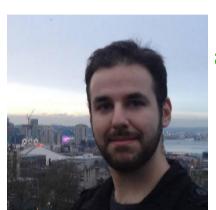
[@dependabot\[bot\]](#) 18 commits



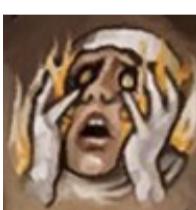
[@r bri](#) 16 commits



[@krmahadevan](#) 14 commits



[@bwalderman](#) 14 commits



[@bayandin](#) 12 commits



[@carlosgcampos](#) 12 commits



[@jayakumarc](#) 12 commits



[@bonigarcia](#) 11 commits



[@43081j](#) 11 commits



[@detro](#) 10 commits



[@josephg](#) 10 commits



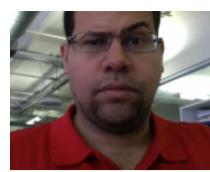
9 commits



9 commits



[@red squirrel](#)



[@isaulv](#)



[@hoefling](#) 9 commits



[@RustyNail](#) 9 commits



[@asolntsev](#) 8 commits



[@touredave](#) 8 commits



[@potapovDim](#) 8 commits



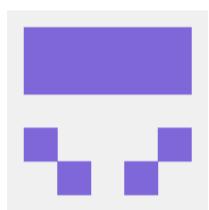
[@glibas](#) 7 commits



[@SinghHrmn](#) 7 commits



[@llaskin](#) 7 commits



[@DrMarcll](#) 7 commits



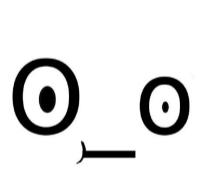
[@ammerrell](#) 7 commits



[@TamsilAmani](#) 7 commits



[@User253489](#) 7 commits



[@dratler](#) 7 commits



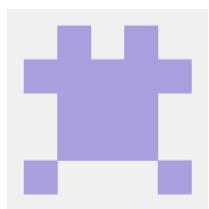
[@dima-groupon](#) 6 commits



[@nikolas](#) 6 commits



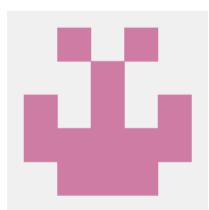
[@shucon](#) 6 commits



[@gpt14](#) 5 commits



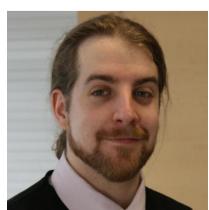
[@Herst](#) 5 commits



[@jimvm](#) 5 commits



[@johanLorenzo](#) 5 commits



[@nschonni](#) 5 commits



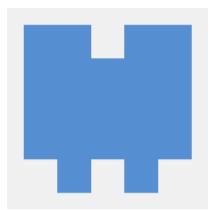
[@seanpoulter](#) 5 commits



[@oddui](#) 5 commits



[@adiohana](#) 5 commits



[@iampopovich](#) 4 commits



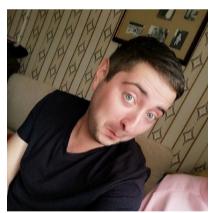
[@Zitrax](#) 4 commits



[@dbo](#) 4 commits



[@Dharin-shah](#) 4 commits



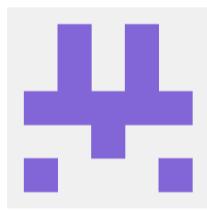
[@xaircore](#) 4 commits



[@bongosway](#) 4 commits



[@sangaline](#) 4 commits



[@guangyuexu](#) 4 commits



[@lauromoura](#) 4 commits



[@Ardesco](#) 4 commits



[@klepikov](#) 4 commits



[@tobii](#) 4 commits

Selenium IDE



[@corevo](#) 2445 commits



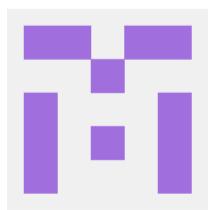
[@tourdedave](#) 607 commits



[@baimao8437](#) 88 commits



[@toddtarsi](#) 59 commits



[@coindude](#) 54 commits



[@Jongkeun](#) 51 commits



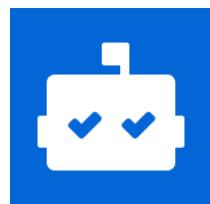
[@petermouse](#) 36 commits



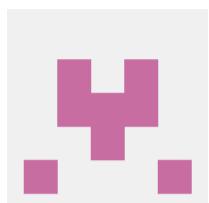
[@LinYunWen](#) 36 commits



[@zavelevsky](#) 34 commits



[@dependabot\[bot\]](#) 24 commits



[@xdennix](#) 15 commits



[@AutomatedTester](#) 15 commits



[@raju249](#) 12 commits



[@diemol](#) 6 commits



[@manoj9788](#) 3 commits



[@shs96c](#) 3 commits



[@zewa666](#) 3 commits



[@Angus3280](#) 2 commits



[@lukeis](#) 2 commits



[@Meir017](#) 2 commits



[@toshiya](#) 2 commits



[@marknoble](#) 2 commits



[@amitzur](#) 1 commits



[@aplorenzen](#) 1 commits



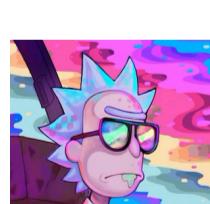
[@atkulp](#) 1 commits



[@dvd900](#) 1 commits



[@p2635](#) 1 commits



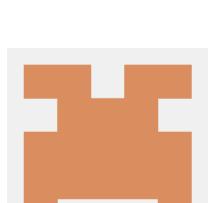
[@sotayamashita](#) 1 commits



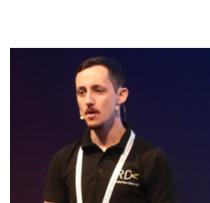
[@samitbadle](#) 1 commits



[@harsha509](#) 1 commits



[@swes1117](#) 1 commits



[@vivrichards600](#) 1 commits



[@bolasblack](#) 1 commits



[@peter-kehl](#) 1 commits

Docker Selenium



[@diemol](#) 478 commits



[@ddavison](#) 134 commits



[@selenium-ci](#) 84 commits



[@mtscout6](#) 53 commits



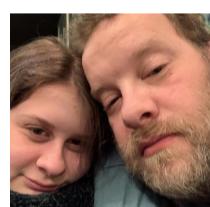
[@kayabendroth](#) 50 commits



[@elgalu](#) 24 commits



[@dependabot\[bot\]](#) 14 commits



[@WillAbides](#) 8 commits



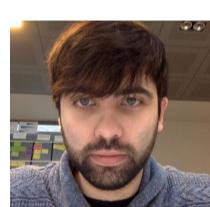
[@marten-cz](#) 5 commits



[@MacCracken](#) 5 commits



[@jsa34](#) 5 commits



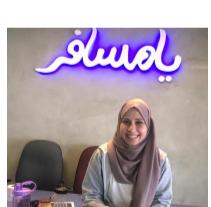
[@garagepoort](#) 4 commits



[@metajiji](#) 4 commits



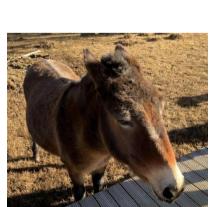
[@manoj9788](#) 4 commits



[@ZainabSalameh](#) 4 commits



[@vasikarla](#) 4 commits



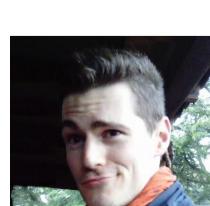
[@chenrui333](#) 4 commits



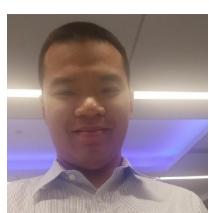
[@niQo](#) 4 commits



[@testphreak](#) 4 commits



[@Remi-p](#) 3 commits



[@tnguyen14](#) 3 commits



[@alexgibson](#) 3 commits



[@jeff-jk](#) 3 commits



[@pabloFuente](#) 3 commits



[@adriangonciarz](#) 2 commits



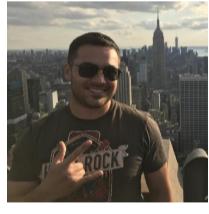
[@ilpianista](#) 2 commits



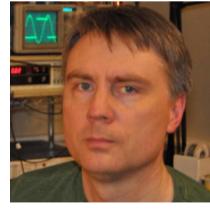
[@chuckg](#) 2 commits



[@davehunt](#) 2 commits



[@den-is](#) 2 commits



[@therealdjryan](#) 2 commits



[@AutomationD](#) 2 commits



[@ehbello](#) 2 commits



[@glibas](#) 2 commits



[@joaoluzjoaquim](#) 2 commits



[@luisfcorreia](#) 2 commits



[@mathieu-pousse](#) 2 commits



[@naveensrinivasan](#) 2 commits



[@phensley](#) 2 commits



[@ryneeverett](#) 2 commits



[@wheleph](#) 2 commits



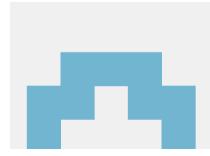
[@evertones](#) 2 commits



[@kmala](#) 2 commits



[@schmunk42](#) 2 commits



[@a-k-g](#) 1 commits



[@alexkogon](#) 1 commits



[@lorantalas](#) 1 commits



[@v1-wizard](#) 1 commits



[@sahajamit](#) 1 commits



[@deviantintegral](#) 1 commits



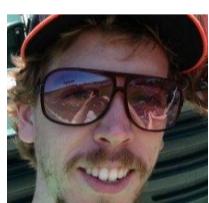
[@Grisu118](#) 1 commits



[@gensc004](#) 1 commits



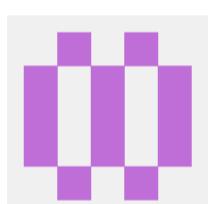
[@budtmo](#) 1 commits



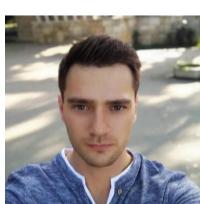
[@charford](#) 1 commits



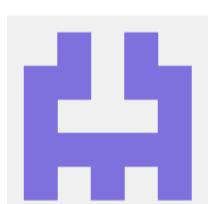
[@cyrille-leclerc](#) 1 commits



[@Dan-DeAraujo](#) 1 commits



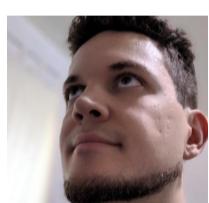
[@germetist](#) 1 commits



[@theddub](#) 1 commits



[@deiwin](#) 1 commits



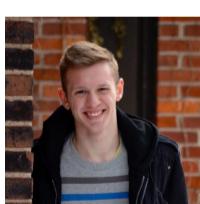
[@enolan](#) 1 commits



[@glogiotatidis](#) 1 commits



[@hbouhadji](#) 1 commits



[@hnryjms](#) 1 commits



[@doublemarket](#) 1 commits



[@hazmeister](#) 1 commits



[@jamesmortensen](#) 1 commits



[@jamesottaway](#) 1 commits



[@jarspi](#) 1 commits



[@JasonBerenbach](#) 1 commits



[@BeyondEvil](#) 1 commits



[@ja8zyjits](#) 1 commits



[@jwhitlock](#) 1 commits



[@jonaseicher](#) 1 commits



[@CaffeinatedCM](#) 1 commits



[@karel1980](#) 1 commits



[@lcnja](#) 1 commits



[@lmtierney](#) 1 commits



[@lukeis](#) 1 commits



[@manusa](#) 1 commits



[@m15o](#) 1 commits



[@matzegebbe](#) 1 commits



[@michallepicki](#) 1 commits



[@mikewrighton](#) 1 commits



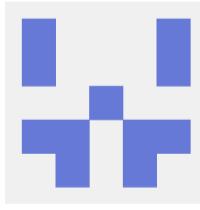
[@meeroslaph](#) 1 commits



[@nipafx](#) 1 commits



[@double16](#) 1 commits



[@PDUBEYOO](#) 1 commits



[@reinholdfuereder](#) 1 commits



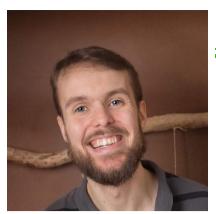
[@remcorakers](#) 1 commits



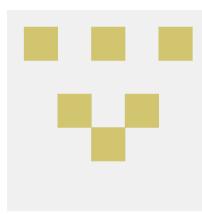
[@rjatkins](#) 1 commits



[@tuxiqae](#) 1 commits



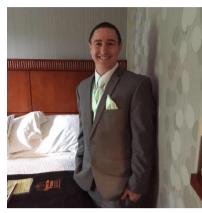
[@cablespaghetti](#) 1 commits



[@scottturley](#) 1 commits



[@sethuster](#) 1 commits



[@smccarthy](#) 1 commits



[@smccarthy-godaddy](#) 1 commits



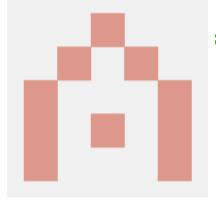
[@simonardejr](#) 1 commits



[@srguglielmo](#) 1 commits



[@stigkj](#) 1 commits



[@subanithak](#) 1 commits



[@tadashi0713](#) 1 commits

Selenium Website & Docs



[@diemol](#) 527 commits



[@harsha509](#) 475 commits



[@selenium-ci](#) 323 commits



[@alaahong](#) 92 commits



[@titusfortner](#) 54 commits



[@kzhirata](#) 41 commits



[@boris779](#) 32 commits



[@pujagani](#) 30 commits



[@alenros](#) 28 commits



[@AlexAndradeSan](#) 25 commits



[@manoj9788](#) 20 commits



[@jmartinezpog](#) 18 commits



18 commits



17 commits



[@AutomatedTester](#)



[@ivanrodjr](#)



[@sindhudiddi](#) 12 commits



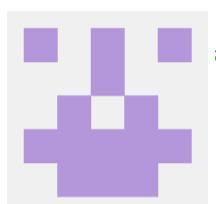
[@shs96c](#) 11 commits



[@hiroksarker](#) 10 commits



[@bonigarcia](#) 9 commits



[@nwintop](#) 8 commits



[@pallavigitwork](#) 7 commits



[@tetration](#) 7 commits



[@connerT](#) 6 commits



[@Greavox](#) 6 commits



[@github-actions\[bot\]](#) 6 commits



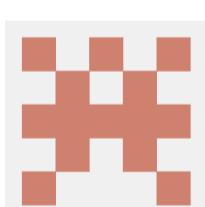
[@liushilive](#) 6 commits



[@nainappa](#) 5 commits



[@hyanx](#) 5 commits



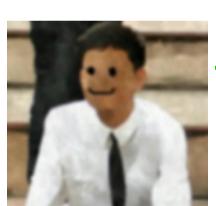
[@lvninety](#) 5 commits



[@Catalin-Negrut](#) 5 commits



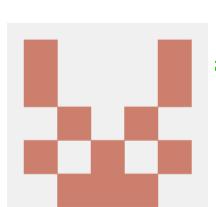
[@davieduardo94](#) 5 commits



[@abhi2810](#) 4 commits



[@barancev](#) 4 commits



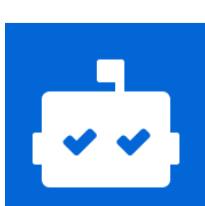
[@kjayachandra2000](#) 4 commits



[@peter-kinzelman](#) 4 commits



[@ant9112](#) 4 commits



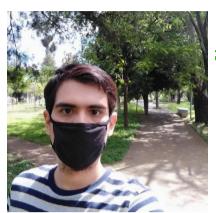
[@dependabot\[bot\]](#) 4 commits



[@TritzA](#) 3 commits



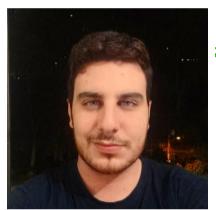
[@Arc-Jung](#) 3 commits



[@ArCiGo](#) 3 commits



[@gnatcatcher-bg](#) 3 commits



[@gustavoefeiche](#) 3 commits



[@Madh93](#) 3 commits



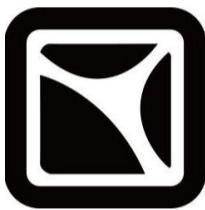
[@rajeevbarde](#) 3 commits



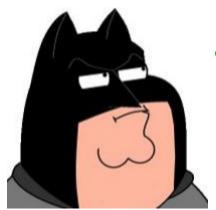
[@raju249](#) 3 commits



[@takeya0x86](#) 3 commits



[@TestOpsCloudchen](#) 3 commits



[@cambiph](#) 3 commits



[@twinstae](#) 3 commits



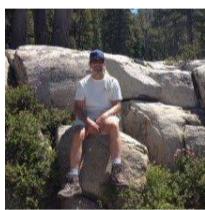
[@Tolerblanc](#) 3 commits



[@luisfcorreia](#) 3 commits



[@beatfactor](#) 2 commits



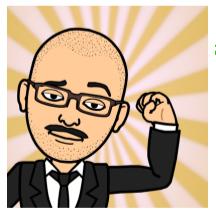
[@wcmcgee](#) 2 commits



[@Bredda](#) 2 commits



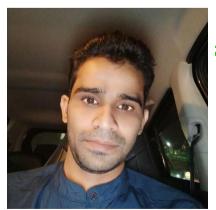
[@cjayswal](#) 2 commits



[@bongosway](#) 2 commits



[@jags14385](#) 2 commits



[@kapilyadav1204](#) 2 commits



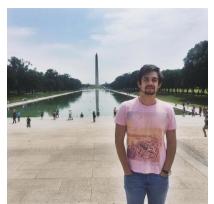
[@mookkiah](#) 2 commits



[@palotas](#) 2 commits



[@miekoF](#) 2 commits



[@natanportilho](#) 2 commits



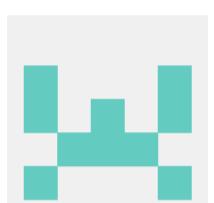
[@rahuljhakant](#) 2 commits



[@TamsilAmani](#) 2 commits



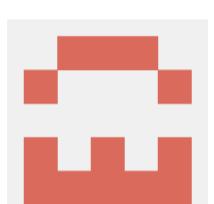
[@urig](#) 2 commits



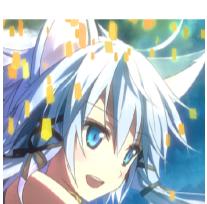
[@ilhanoztozlu](#) 2 commits



[@coodjokergl](#) 2 commits



[@sangheon4353](#) 2 commits



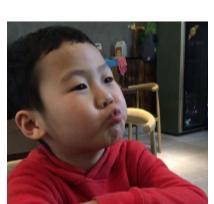
[@imba-tjd](#) 2 commits



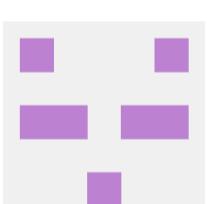
[@0420syj](#) 2 commits



[@TheTestLynx](#) 2 commits



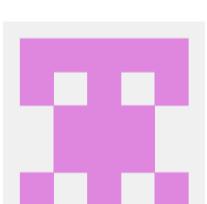
[@k19810703](#) 2 commits



[@ajsdecode](#) 1 commits



[@abhishek-malani](#) 1 commits



[@adithyab94](#) 1 commits



[@alexako](#) 1 commits



[@vinogradoff](#) 1 commits



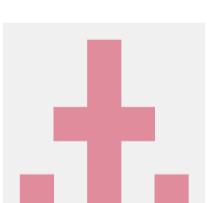
[@amiluslu](#) 1 commits



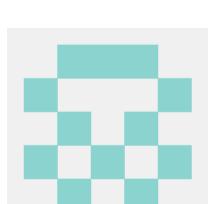
[@amirmojiry](#) 1 commits



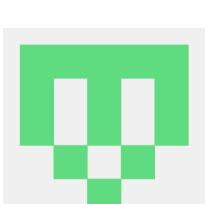
[@sahajamit](#) 1 commits



[@fiveych](#) 1 commits



[@Archibaald-dev](#) 1 commits



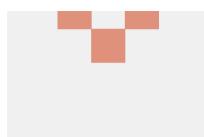
[@bafIOA](#) 1 commits



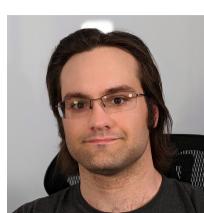
[@bharathmuddada](#) 1 commits



[@BlazerYoo](#) 1 commits



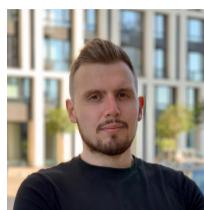
[@stakato2005](#) 1 commits



[@SalmonMode](#) 1 commits



[@ddavison](#) 1 commits



[@dbudim](#) 1 commits



[@dylanlive](#) 1 commits



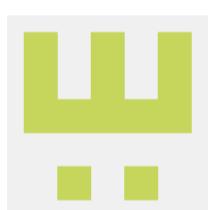
[@Eekain](#) 1 commits



[@fastrde](#) 1 commits



[@yufanme](#) 1 commits



[@unam3](#) 1 commits



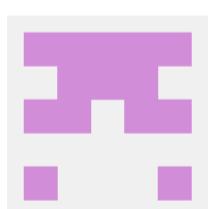
[@f-g-s](#) 1 commits



[@gabrielcristhie](#) 1 commits



[@gifflet](#) 1 commits



[@Haifischbecken](#) 1 commits



[@SinghHrmn](#) 1 commits



[@Hex052](#) 1 commits



[@wafuwafu13](#) 1 commits

Previous Selenium Website



[@lukejs](#) 417 commits



[@tourededave](#) 89 commits



[@shs96c](#) 85 commits



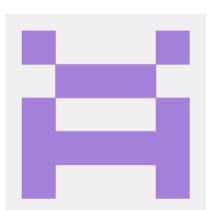
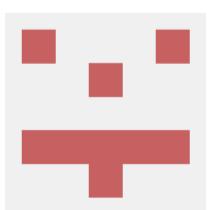
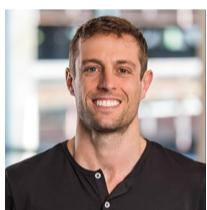
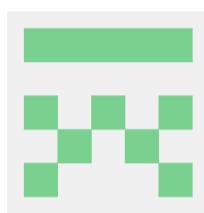
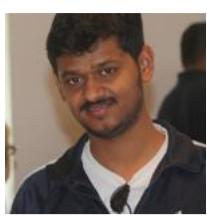
[@pgrandje](#) 79 commits



[@barancev](#) 63 commits



[@lightbody](#) 59 commits

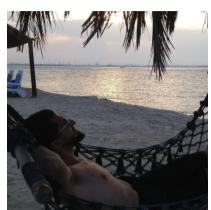




[@corevo](#) 5 commits



[@diemol](#) 3 commits



[@asashour](#) 2 commits



[@oleksandr-lobunets](#) 2 commits



[@alex-savchuk](#) 2 commits



[@javabrett](#) 2 commits



[@darrincherry](#) 2 commits



[@eranmes](#) 2 commits



[@hazmeister](#) 2 commits



[@julianharty](#) 2 commits



[@mikemelia](#) 2 commits



[@paul-hammant](#) 2 commits



[@labkey-tchad](#) 2 commits



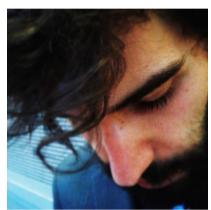
[@abhijain2618](#) 2 commits



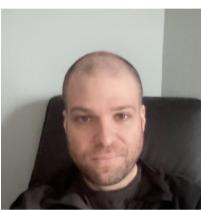
[@agabrys](#) 1 commits



[@azawawi](#) 1 commits



[@alb-i986](#) 1 commits



[@hollingsworthd](#) 1 commits



[@dylans](#) 1 commits



[@EmidioStani](#) 1 commits



[@FagnerMartinsBrack](#) 1 commits



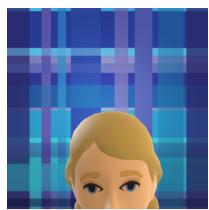
[@Xaeroxe](#) 1 commits



[@JamesZoft](#) 1 commits



[@jleyba](#) 1 commits



[@JasnoWa](#) 1 commits



[@JustAGuyTryingToCodeSomething](#) 1 commits



[@kdamball](#) 1 commits



[@laurinkeithdavis](#) 1 commits



[@klamping](#) 1 commits



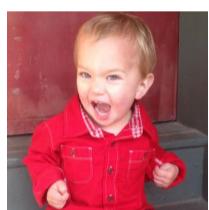
[@krmahadevan](#) 1 commits



[@krosenvold](#) 1 commits



[@mmerrell](#) 1 commits



[@grawk](#) 1 commits



[@mcavigelli](#) 1 commits



[@michaelwowro](#) 1 commits



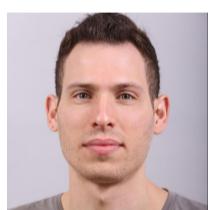
[@muralidharand](#) 1 commits



[@meeroslaph](#) 1 commits



[@NickAb](#) 1 commits



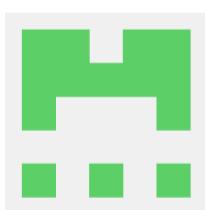
[@ohadschn](#) 1 commits



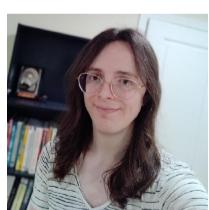
[@oifland](#) 1 commits



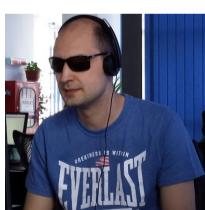
[@rbri](#) 1 commits



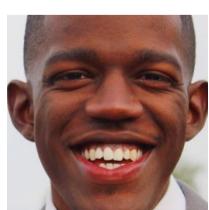
[@roydekleijn](#) 1 commits



[@QuinnWilton](#) 1 commits



[@smatei](#) 1 commits



[@harrissAvalon](#) 1 commits



[@SteveDesmond-ca](#) 1 commits



[@Vimal-N](#) 1 commits



[@yasinguzel](#) 1 commits



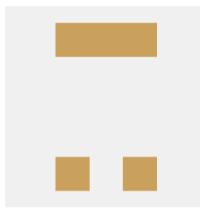
[@tobecrazy](#) 1 commits



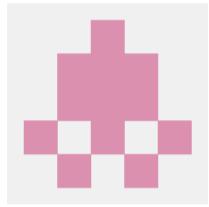
[@Zearin](#) 1 commits



[@beckendorff](#) 1 commits



[@daveOrleans](#) 1 commits



[@androiddriver](#) 1 commits



[@mauk81](#) 1 commits



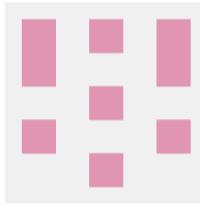
[@pharry22](#) 1 commits



[@prab2112](#) 1 commits



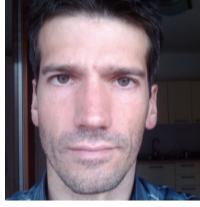
[@refactoror](#) 1 commits



[@rogerdc](#) 1 commits



[@tibord](#) 1 commits



[@ygmarchi](#) 1 commits

Previous Documentation Rewrite Project



[@diemol](#) 54 commits



[@hazmeister](#) 30 commits



[@santiycr](#) 27 commits



[@AlexAndradeSan](#) 25 commits



[@lukeis](#) 21 commits



[@harsha509](#) 17 commits



[@ddavison](#) 16 commits



[@davehunt](#) 12 commits



[@manoj9788](#) 12 commits



[@orieken](#) 12 commits



[@djangofan](#) 12 commits



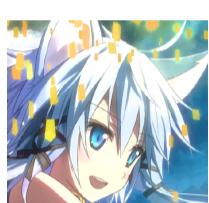
[@liushilive](#) 8 commits



[@User253489](#) 7 commits



[@jimholmes](#) 6 commits



[@imba-tjd](#) 6 commits



[@mmerrell](#) 6 commits



[@shs96c](#) 6 commits



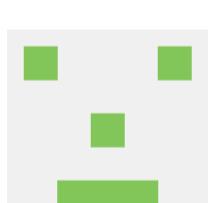
[@picimako](#) 5 commits



[@vijay44](#) 5 commits



[@cambiph](#) 5 commits



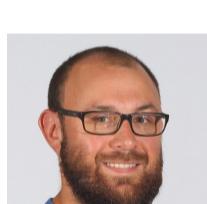
[@nvonop](#) 4 commits



[@rivlinp](#) 4 commits



[@sheg](#) 4 commits



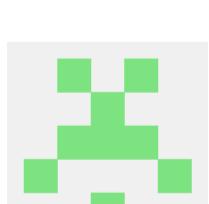
[@bizob2828](#) 4 commits



[@detro](#) 3 commits



[@Ardesco](#) 3 commits



[@TheTestLynx](#) 3 commits



[@boris779](#) 2 commits



[@Bredda](#) 2 commits



[@juperala](#) 2 commits



[@lmtierney](#) 2 commits



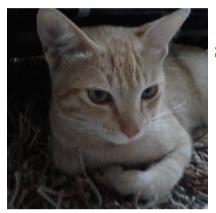
[@systemboogie](#) 2 commits



[@palotas](#) 2 commits



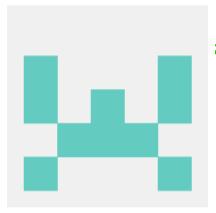
[@miekof](#) 2 commits



[@sri85](#) 2 commits



[@hoanluu](#) 2 commits



[@ilhanoztozlu](#) 2 commits



[@paul-barton](#) 2 commits



[@adithyab94](#) 1 commits



[@alenros](#) 1 commits



[@p0deje](#) 1 commits



[@AJ-72](#) 1 commits



[@abotalov](#) 1 commits



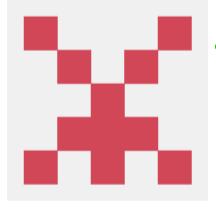
[@bhardin](#) 1 commits



[@chamiz](#) 1 commits



[@dennybiasioli](#) 1 commits



[@donhuvy](#) 1 commits



[@bongosway](#) 1 commits



[@nicegraham](#) 1 commits



[@austenjt](#) 1 commits



[@kmcgon](#) 1 commits



[@MartinDelille](#) 1 commits



[@michael-coleman](#) 1 commits



[@misiekofski](#) 1 commits

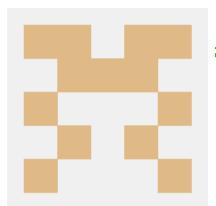




[@MilanMasek](#) 1 commits



[@rakib-amin](#) 1 commits



[@NRezek](#) 1 commits

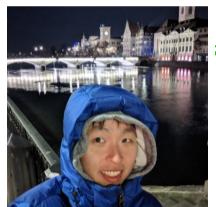
[@nikai3d](#) 1 commits



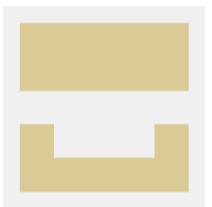
[@OndraM](#) 1 commits



[@sourabhkt](#) 1 commits



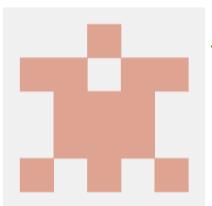
[@whhone](#) 1 commits



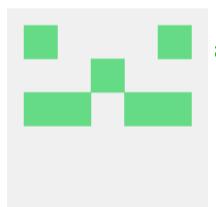
[@yarix](#) 1 commits



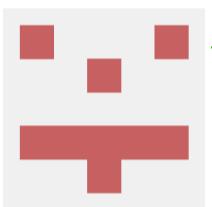
[@ZbigniewZabost](#) 1 commits



[@agmen](#) 1 commits



[@hking-shutterfly](#) 1 commits



[@jimevans](#) 1 commits



[@948462448](#) 1 commits



[@marilyn](#) 1 commits



[@riccione](#) 1 commits



[@tungla](#) 1 commits



[@zeljkofilipin](#) 1 commits

Third-Party software used by Selenium documentation project:

Software	Version	License
Hugo	v0.83.1	Apache 2.0
Docsy	—	Apache 2.0

License

All code and documentation originating from the Selenium project is licensed under the Apache 2.0 license, with the [Software Freedom Conservancy](#) as the copyright holder.

The license is included here for convenience, but you can also find it on the [Apache Foundation's websites](#):

Apache License
Version 2.0, January 2004
<http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted"

means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents

of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

8.2 - Contributing to the Selenium site & documentation

Information on improving documentation and code examples for Selenium

Selenium is a big software project, its site and documentation are key to understanding how things work and learning effective ways to exploit its potential.

This project contains both Selenium's site and documentation. This is an ongoing effort (not targeted at any specific release) to provide updated information on how to use Selenium effectively, how to get involved and how to contribute to Selenium.

Contributions toward the site and docs follow the process described in the below section about contributions.

The Selenium project welcomes contributions from everyone. There are a number of ways you can help:

Report an issue

When reporting a new issues or commenting on existing issues please make sure discussions are related to concrete technical issues with the Selenium software, its site and/or documentation.

All of the Selenium components change quite fast over time, so this might cause the documentation to be out of date. If you find this to be the case, as mentioned, don't hesitate to create an issue for that. It also might be possible that you know how to bring up to date the documentation, so please send us a pull request with the related changes.

If you are not sure about what you have found is an issue or not, please ask through the communication channels described at <https://selenium.dev/support>.

Contributions

The Selenium project welcomes new contributors. Individuals making significant and valuable contributions over time are made *Committers* and given commit-access to the project.

This guide will guide you through the contribution process.

Step 1: Fork

Fork the project [on Github](#) and check out your copy locally.

```
% git clone git@github.com:seleniumhq/seleniumhq.github.io.git  
% cd seleniumhq.github.io
```

Dependencies: Hugo

We use [Hugo](#) and the [Docsy theme](#) to build and render the site. You will need the “extended” Sass/SCSS version of the Hugo binary to work on this site. We recommend to use Hugo 0.94 or higher.

Please follow the [Install Hugo](#) instructions from Docsy.

Step 2: Branch

Create a feature branch and start hacking:

```
% git checkout -b my-feature-branch
```

We practice HEAD-based development, which means all changes are applied directly on top of `dev`.

Step 3: Make changes

The repository contains the site and docs. Before jumping into making changes, please initialize the submodules and install the needed dependencies (see commands below). To make changes to the site, work on the `website_and_docs` directory. To see a live preview of your changes, run `hugo server` on the site's root directory.

```
% git submodule update --init --recursive  
% cd website_and_docs  
% hugo server
```

See [Style Guide](#) for more information on our conventions for contribution

Step 4: Commit

First make sure git knows your name and email address:

```
% git config --global user.name 'Santa Claus'  
% git config --global user.email 'santa@example.com'
```

Writing good commit messages is important. A commit message should describe what changed, why, and reference issues fixed (if any). Follow these guidelines when writing one:

1. The first line should be around 50 characters or less and contain a short description of the change.
2. Keep the second line blank.
3. Wrap all other lines at 72 columns.
4. Include `Fixes #N`, where `N` is the issue number the commit fixes, if any.

A good commit message can look like this:

```
explain commit normatively in one line

Body of commit message is a few lines of text, explaining things
in more detail, possibly giving some background about the issue
being fixed, etc.

The body of the commit message can be several paragraphs, and
please do proper word-wrap and keep columns shorter than about
72 characters or so. That way `git log` will show things
nicely even when it is indented.

Fixes #141
```

The first line must be meaningful as it's what people see when they run `git shortlog` or `git log --oneline`.

Step 5: Rebase

Use `git rebase` (not `git merge`) to sync your work from time to time.

```
% git fetch upstream  
% git rebase upstream/trunk
```

Step 6: Test

Always remember to [run the local server](#), with this you can be sure that your changes have not broken anything.

Step 7: Push

```
% git push origin my-feature-branch
```

Go to <https://github.com/yourusername/seleniumhq.github.io.git> and press the *Pull Request* and fill out the form. **Please indicate that you've signed the CLA** (see Step 7).

Pull requests are usually reviewed within a few days. If there are comments to address, apply your changes in new commits (preferably [fixups](#)) and push to the same branch.

Step 8: Integration

When code review is complete, a committer will take your PR and integrate it on the repository's trunk branch. Because we like to keep a linear history on the trunk branch, we will normally squash and rebase your branch history.

Communication

All details on how to communicate with the project contributors and the community overall can be found at <https://selenium.dev/support>

8.3 - Style guide for Selenium documentation

Conventions for contributions to the Selenium documentation and code examples

Read our [contributing documentation](#) for complete instructions on how to add content to this documentation.

Alerts

Alerts have been added to direct potential contributors to where specific help is needed.

When code examples are needed, this code has been added to the site:

```
{{< alert-code />}}
```

Which gets displayed like this:

Coding Help ×

Note: This section could use some updated code examples

Check our [contribution guidelines](#) and [code example formats](#) if you'd like to help.

To specify what code is needed, you can pass information inside the tag:

```
<{{ alert-code }}>  
specifically code that does this one thing.  
<{{ /alert-code }}>
```

Which looks like this:

Coding Help ×

Note: This section could use some updated code examples

specifically code that does this one thing.

Check our [contribution guidelines](#) and [code example formats](#) if you'd like to help.

Similarly, for additional content you can use:

```
<{{ alert-content }}>
```

or

```
<{{ alert-content }}>  
Additional information about what specific content is needed  
<{{ /alert-content }}>
```

Which gets displayed like this:

contribution guidelines if you'd like to help.'"/>

Note: This section needs additional and/or updated content

Additional information about what specific content is needed

Check our [contribution guidelines](#) if you'd like to help.

Capitalization of titles

Our documentation uses Title Capitalization for `linkTitle` which should be short and Sentence capitalization for `title` which can be longer and more descriptive. For example, a `linkTitle` of *Special Heading* might have a `title` of *The importance of a special heading in documentation*

Line length

When editing the documentation's source, which is written in plain HTML, limit your line lengths to around 100 characters.

Some of us take this one step further and use what is called [semantic linefeeds](#), which is a technique whereby the HTML source lines, which are not read by the public, are split at 'natural breaks' in the prose. In other words, sentences are split at natural breaks between clauses. Instead of fussing with the lines of each paragraph so that they all end near the right margin, linefeeds can be added anywhere that there is a break between ideas.

This can make diffs very easy to read when collaborating through git, but it is not something we enforce contributors to use.

Translations

Selenium now has official translators for each of the supported languages.

- If you add a code example to the `important_documentation.en.md` file, also add it to `important_documentation.ja.md` , `important_documentation.pt-br.md` , `important_documentation.zh-cn.md` .
- If you make text changes in the English version, just make a Pull Request. The new process is for issues to be created and tagged as needs translation based on changes made in a given PR.

Code examples

All references to code should be language independent, and the code itself should be placed inside code tabs.

Default Code Tabs

The Docsy code tabs look like this:

[Java](#) [Python](#) [CSharp](#) [Ruby](#) [JavaScript](#) [Kotlin](#)

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

To generate the above tabs, this is what you need to write. Note that the tabpane includes `langEqualsHeader=true`. This auto-formats the code in each tab to match the header name.

```
{{< tabpane langEqualsHeader=true >}}
{{< tab header="Java" >}}
  WebDriver driver = new ChromeDriver();
{{< /tab >}}
{{< tab header="Python" >}}
  driver = webdriver.Chrome()
{{< /tab >}}
{{< tab header="CSharp" >}}
  var driver = new ChromeDriver();
{{< /tab >}}
{{< tab header="Ruby" >}}
  driver = Selenium::WebDriver.for :chrome
{{< /tab >}}
{{< tab header="JavaScript" >}}
  let driver = await new Builder().forBrowser('chrome').build();
{{< /tab >}}
{{< tab header="Kotlin" >}}
  val driver = ChromeDriver()
{{< /tab >}}
{{< /tabpane >}}
```

Link to GitHub

All code examples should be present and linked to in our example [repos](#).

With the `gh-codeblock` shortcode, it is possible to render code hosted in a GitHub repository. In this case, `langEqualsHeader=true` is only needed if the `tabpane` mixes more than one `tab` with code hosted on GitHub and code examples added directly in the markdown file (which should not happen often).

However, `disableCodeBlock=true` is needed at the `tab` level when using the `gh-codeblock` shortcode. This is an example of the `gh-codeblock` shortcode usage:

```
{{< tabpane >}}
{{< tab header="Link" disableCodeBlock=true >}}
  {{< gh-codeblock path="examples/java/src/test/java/dev/selenium/getting_started/FirstScr
{{< /tab >}}
{{< tab header="No Link" >}}
  This content should not get linked to GitHub
{{< /tab >}}
{{< /tabpane >}}
```

Which looks like this:

[Link](#) [No Link](#)

```
@Test
public void eightComponents() {
  driver.get("https://duckduckgo.com/");

  String title = driver.getTitle();
  assertTrue(title.contains("DuckDuckGo"));

  driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));

  WebElement searchBar = driver.findElement(By.name("q"));
  WebElement searchButton = driver.findElement(By.id("search_button_homepa
```

```
searchBox.sendKeys("Selenium");
searchButton.click();

searchBox = driver.findElement(By.name("q"));
String value = searchBox.getAttribute("value");
assertEquals("Selenium", value);
}
```

 [Check code on GitHub](#)

Disabling Code Block

If you want your example to include both text and code, you need to disable the default behavior that puts everything inside a code block

Maybe you want something like this:

[Java](#)

1. Start the driver

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

2. Navigate to a page

```
driver.get("https://selenium.dev");
```

3. Quit the driver

```
driver.quit();
```

For this you need to use `disableCodeBlock=true` instead of `langEqualsHeader=true`

You need to be explicit about which parts are code and which are not, do not indent plain text or it will still be treated like a codeblock:

```
{{< tabpane disableCodeBlock=true >}}
{{< tab header="Java" >}}
1. Start the driver
```java
WebDriver driver = new ChromeDriver();
```
2. Navigate to a page
```java
driver.get("https://selenium.dev");
```
3. Quit the driver
```java
driver.quit();
```
{{< /tab >}}
< ... >
{{< /tabpane >}}
```

Link to GitHub

All code examples should be present and linked to in our example [repos](#).

With the `gh-codeblock` shortcode, it is possible to render code hosted in a GitHub repository. This is an example of the `gh-codeblock` shortcode usage:

```
 {{< gh-codeblock path="examples/java/src/test/java/dev/selenium/getting_started/FirstScriptTest.java" >}}
```

Which looks like this:

```
@Test
public void eightComponents() {
    driver.get("https://duckduckgo.com/");

    String title = driver.getTitle();
    assertTrue(title.contains("DuckDuckGo"));

    driver.manage().timeouts().implicitlyWait(Duration.ofMillis(500));

    WebElement searchBox = driver.findElement(By.name("q"));
    WebElement searchButton = driver.findElement(By.id("search_button_homepage"));

    searchBox.sendKeys("Selenium");
    searchButton.click();

    searchBox = driver.findElement(By.name("q"));
    String value = searchBox.getAttribute("value");
    assertEquals("Selenium", value);
}
```

 [Check code on GitHub](#)

Consistent Heights

Typically, a user will pick their language and not switch tabs again. If they do, however, the entire page will shift because the height of the block is based on the amount of code in the tab. This can get especially jarring at the bottom of a page with a lot of tabs.

You can add a “height” value to the `tabpane` that corresponds to the lines of code or some equivalent for text:

```
 {{< tabpane disableCodeBlock=true height="5" >}}
```

Code Comments

Minimize code comments because they are difficult to translate. Further, try to avoid over-explaining each line of code unless it is directly related to the page you are writing.

8.4 - Musings about how things came to be

Details mostly of interest to Selenium devs about how and why certain parts of the project were created

This documentation previously located [on the wiki](#)

Introduction

This is a work in progress. Feel free to add things you know or remember.

How did the Automation Atoms come about?

On 2012-04-04, jimevans asked on the #selenium IRC channel:

“What I wanted to ask you about was the history of the automation atoms. I seem to remember them springing fully formed, as if from the head of Zeus, and I’m sure that wasn’t the case. Can you refresh my memory as to how the concept happened?”

simonstewart then proceeded to tell us a nice little story:

Sure. Are we sitting comfortably? Then I’ll begin. (Brit joke, there)

Imagine wavy lines as the screen dissolves and we’re transported back to when selenium and webdriver were different projects. Before the projects merged, there was an awful lot of congruent code in webdriver. Congruent, but not shared. The Firefox driver was in JS. The IE driver was mostly C++. The Chrome driver was mostly JS, but different JS from the Firefox driver. And HtmlUnit was unique.

We then added Selenium Core to the mix. Yet more JS that did basically the same thing.

Within Google, I was becoming the TL of the browser automation team. And was corralling a framework of our own into the mix. Which was written in JS, and had once been based on Core before it span off on its own path.

So: multiple codebases, lots of JS doing more or less the same thing. And loads of bugs. Weird mismatches of behaviour in edge-cases.

shudder

So I had a bit of a think. (Dangerous, I know) The idea was to extract the “best of breed” code from all three frameworks (Core, WebDriver and the Google tool). Break them down into code that could be shared. “The smallest, indivisible unit of browser automation” .

Or “atoms” for short.

These could be used as the basis the *everything*. Consistent behaviour between browsers. and apis. The other important point was that the JS code in webdriver and core was grown organically. Which is a polite way of saying “I’d rather never edit it again”. Which is a polite way of saying that it was of dubious quality . In places.

So: high quality was important. And I wanted the code broken up into modules. Because editing a 10k LOC file isn’t a bright idea.

Within Google we had a library called Closure. Which not only allowed modularization, but “denormalization” of modules into a single file via compilation. And I knew it was being open sourced. So we started building the library in the google codebase. (Where we had access to the unreleased library, code review tools and our amazing testing infrastructure). Using Closure Library.

“dom.js” was probably the first file I wrote. (We can check). Greg Dennis and Jason Leyba joined in the fun. And the atoms have been growing ever since.

Technically, we should be calling anything outside of “javascript/atoms” molecules. But then we can’t say that we have atomic drivers. and use imagery from the 50s to describe them.

sigh

jimevans replied: “molecular drivers?”

And simonstewart finished with:

Indeed :) The idea is that the atoms are the lowest level. And we compose the atoms to conform to the WebDriver or RC apis in “javascript/{selenium,webdriver}-atoms” respectively. And then suck those in as necessary.

A Story of Crazy-Fun

Simon Stewart :

So, let’s go back to the very beginning of the project

When it was me, on my own

(the webdriver project, that is, not selenium itself)

I knew that I wanted to cover multiple different languages, and so wanted a build tool that could work with all of them

That is, that didn't have a built in preference for one that made working with other languages painful

ant is java biased. As is maven.

nant and msbuild are .net biased

rake, otoh, supports nothing very well

But, and this is key, any valid rake script is also a valid ruby program

It's possible to extend rake to build *anything*

So: rake it was

The initial rake file was pretty small and manageable

But as the project grew, so did the Rakefile

Until there was only person who could deal with it (me), and even then it was pretty shaky

So, rather than have a project that couldn't be built, I extracted some helper methods to do some of the heavy lifting

Which made the Rakefile comprehensible again

But they project kept getting bigger

And the Rakefile got harder and harder to grok

At the time, I was working at Google, who have a wonderful build system

Google's system is declarative and works across multiple different languages consistently

And, most important, it breaks up the build from a single file into little fragments

I asked the OSS chaps at Google if it was okay to open source the build grammar, and they gave it the green light

So we layered that build grammar into the selenium codebase

With one minor change (we handle dictionary args)

But that grammar sits on top of rake

still, after all this time

And there's a problem

And that's that rake is single threaded

So our builds are constrained to run serially

We could use "multitask" types to improve things, but when I've tried that things got very messy, very fast

So, our next hurdle is that crazyfun.rb is slow: we need to go faster

Which implies a rewrite of crazyfun

I'm most comfortable in java

So, I've spiked a new version in java that handles the java and js compilation

It's significantly faster

But, and this is also important, it's a spike

The code was designed to be disposable.

Now that things have been proved out, I'd really like to do a clean implementation

But I'm torn

Do I "finish" the new, very fast crazyfun java enough to replace the ruby version?

A story of driver executables

jimevans

noob_einsteinsfo: alright, story time, then. are we sitting comfortably? then we'll begin.

noob_einsteinsfo: back when i first started working on the project (circa 2010), the drivers for all of the browsers were built and maintained by the project.

at the time, that meant IE, firefox, and chrome.

all of those drivers were packaged as part of the selenium standalone server, and were also packaged in with the various language bindings.

this was a conscious decision, so that if one were running locally, there would be no need for the java runtime on the machine just to automate a given browser.

there were two factors that led to the development of browser drivers as separate executables.

as a quick aside, remember that the webdriver philosophy is to automate the browser using the "best-fit" mechanism for that particular browser.

for IE, that means using the COM interfaces; for firefox at the time, that meant using a browser extension; for chrome, it also meant a browser extension.

so that meant that the IE driver was developed as a DLL in C++ that was loaded by the language bindings, and communicated with via whatever native-code mechanism was provided by the language (JNI for java, P/Invoke for .NET, ctypes for python, etc.).

it also meant that the firefox driver was developed as a browser extension that was packaged inside the various language bindings, and extracted, and used in a profile in firefox.

as i said, the IE driver was implemented as a DLL, loaded and communicated with using different mechanisms for different language bindings.

the problem is that each of those language-specific mechanisms had different load/unload semantics.

ruby, for example, would never call the windows FreeLibrary API after loading the DLL into memory, making multiple instances really challenging.

process semantics, however, as in, starting, stopping, and managing the lifetime of a process on the OS, whatever the OS, are remarkably similar across all languages.

so when the IE driver rewrite was completed in 2010, the development team (me) decided to make it a separate executable, so that the load/unload semantics could be consistent no matter what language bindings one was using.

concurrently with this, the chromium team made the decision to follow opera's lead and provide a driver implementation for chrome.

an implementation that they would develop, enhance, and maintain going forward, relieving the selenium project of the burden of maintaining a chrome driver.

XgizmoX

and that driver is part of the browser?

jimevans

XgizmoX: not really, but i believe there may be some smarts built into chrome itself that knows when it's being automated via chromedriver. one of the googlers would be a better person to ask about that.

anyway, knowing the different in shared library (.dll/.so/.dynlib) loading semantics, the chromium team (with my encouragement) decided to release their chromedriver implementation as a separate executable.

fast-forward a couple of years, and you begin to see the effort to make webdriver a w3c standard. a working group with the w3c created a specification (still in progress, but getting close to finished with the first version), which codified the behavior of webdriver, and how a browser should react to its methods. furthermore, it standardized the protocol used to communicate between language bindings and a driver for a particular browser.

i can't emphasize how important and groundbreaking this was.

because the w3c and the webdriver working group within it are made up of representatives from the browser vendors themselves, it ensures that the solution will be supported directly by the browser vendors.

mozilla created their webdriver implementation (geckodriver) for firefox.

the most efficient mechanism for distribution of that browser driver, while maintaining the proper semantics for the language bindings, was to ship as a separate executable.

note, this is a gross oversimplification of the geckodriver architecture; the actual executable acts as a relatively thin shim, translating from the wire protocol of the spec to their internal marionette protocol

but the point still stands.

anyway, the landscape is currently evolving regarding browser-vendor-provided driver implementation. microsoft has one for edge, apple has one for safari (10 and above), the chromium team (largely staffed by googlers) has one for chrome, and now mozilla has one for firefox. given the limited utility of the legacy firefox driver going forward, breaking it out into a separate executable would be wasted effort.

this is particularly so, since all of the communication bits that are normally handled by the executable (listening for and responding to http requests from the language bindings) are handled entirely by the browser extension. \

there's literally no need for the legacy firefox driver to be a separate executable.

moreover, making it independent of a language runtime would be a significant portion of work (because a .NET shop might reasonably balk at being required to install, say, the java runtime just to automate firefox)

so historically speaking, noob-einstiefsfo, that's the general reason for why separate executables have become the norm, and why that paradigm wasn't extended to include the legacy firefox driver.

does that make sense?

okay.

now.

about geckodriver.

the tale of geckodriver is intimately bound with the status of the aforementioned w3c webdriver spec.

level 1 of the specification is mostly done, though it took a number of years of effort to get there. it took a large effort from some very smart people (AutomatedTester among them) to mold the initial documentation of what the webdriver open source software (OSS) project did into proper specification language that could be interpreted and turned into actionable stuff by a browser vendor or other implementor.

when beginning the geckodriver (nee marionette) project, mozilla decided to base their implementation on the spec, and only the spec, not following the OSS implementation.

this created something of a chicken-and-egg problem, in that while the spec language wasn't completed, it couldn't be implemented.

it's only been in the last six months or so that the language concerning the advanced user interactions api (the Actions class in java and .NET) has been made robust enough to actually implement.

accordingly, that's the single biggest missing chunk of functionality in geckodriver at present. it wasn't implementable via the spec, so it hasn't been implemented.

i do know that it's a very high priority for AutomatedTester and his team to get that implementation done and available.

as for why geckodriver is mandatory, and the default implementation for automating firefox in 3.x, that also comes down to some decisions made by mozilla.

TheSchaf

so i guess there is no other choice than to use the old FF as long as required features are missing
WhereIsMySpoon

TheSchaf: if you need those features, yes

or use another browser

TheSchaf

well, moveTo and sendKeys should be pretty basic :p

jimevans

TheSchaf: element.sendKeys works just fine. it's Actions.sendKeys that would be broken.
in firefox version fortysomething (i misremember the exact version), there was a feature added that blocked browser extensions that hadn't been signed by the mozilla security team.
remember that the legacy firefox driver was built as a browser extension? well, with that feature of the browser enabled, the legacy driver couldn't be loaded by the browser.

now, for several versions of firefox, it was possible to disable this feature of the browser, and allow unsigned extensions to continue to be loaded.

and selenium did this, by virtue of the settings used in the anonymous profile the bindings created when launching firefox.

until firefox 48, at which point, it was no longer possible to disable loading of unsigned extensions.
at that point, geckodriver was the only way forward for that.

now, two more slight points, then i'll be done with story time.

first, by nature of what the legacy driver extension does, it's not possible to get it to pass the

certification process of the mozilla security team.
we asked, were denied, and were told it wouldn't happen ever, full stop.
and that's perfectly reasonable, since what that extension does is a security hole big enough to
drive a whole fleet of lorries through.
second, it turns out there may, in fact, be a way to privately sign the legacy extension so that it can
be loaded and used privately by versions of firefox 48 and higher.
that's still a less-than-ideal approach, because there's no way that our merry band of open source
developers can know how to automate firefox better than the development teams at mozilla, who
create the browser in the first place.
i totally get the frustration that geckodriver doesn't have the full feature parity of the legacy
implementation, especially when it feels like one is being forced to move to it.
raging at the selenium project about that decision is directing one's ire in entirely the wrong
direction.
however, before going off and saying horrible things about mozilla's decisions, do know that
mozilla has several people who are constantly engaged in the project, a few of them right here in
this very channel (AutomatedTester, davehunt, to name two).
i'm sure i've glossed over or mischaracterized some of the historical details of these things, and i'm
happy to be corrected. i'm old, after all, and the memory isn't what it used to be.
but that, my friends, is the (not so very) short history of why we have separate executables for
drivers, and why geckodriver is the way forward, and why a move to it was necessary when the
move was made even though some functionality was lacking.

jimevans feels like he's become an unofficial historian of the webdriver project

transcript: <https://botbot.me/freenode/selenium/2016-12-21?msg=78265715&page=6>

An informal naming of our releases (by channel topic in IRC)

- Selenium 2 beta 3 'the next generation browser release' now available - <http://bit.ly/i9bkC2>
- Selenium 2 RC1 'the grid release' now available - <http://bit.ly/jgZxW8>
- Selenium 2 RC2 the 'works better release' now available - <http://bit.ly/mJjX1z>
- Selenium RC3 - "the next one is the 'big' one" release - <http://bit.ly/kpiACx>
- Selenium 2.0 Final unleashed upon the unsuspecting masses
- Selenium 2.1.0 now available (yes, even for maven users now)
- Selenium 2.2.0 now available (in nuget .. and yes, even maven)
- Selenium 2.3.0 available now. A new tradition!
- Selenium 2.4.0 is out – stuff changed, but there is no blog post yet
- Selenium 2.5.0. mmmm. bacon.
- Selenium 2.6.0 is now available. Switch and save 15% or more on car insurance
- Ruby bindings for Selenium 2.7.0 first out of the gate (on twitter at any rate). Jari is a machine...
- Selenium 2.8.0 is out now – day old bacon is still bacon
- sadly we are missing IRC logs...
- Selenium 2.22: The month long weekly release is finally here!
- Selenium 2.23: "Now with awesome!" Wait. What? Now?!
- Selenium 2.24: Now with more, erm, stuff?
- Selenium 2.25: Tracking nicely

- 2.26 is out!
- Selenium 2.27 has been released with fixes for Firefox 17. Get it while it's hot!
- (there was no 2.28 topic update) code.google.com/p/selenium mirrored on github.com/seleniumHQ/selenium - we're on git now!
- 2.29.0 is out now! First git release with FF18 support!
- BOOM! 2.31 is released with native event support for Firefox 19 even.
- “correlation does not imply causation” 2.32.0 released with Firefox 20 support.
- the US government is open again! Let's celebrate with 2.36 newly released, with FF24 support
- 2.40 is wow much automate so fixes such awe
- 2.41 - the last ie6 “supported” release
-
- 2.45.0 - released w/ FF36 support
- 2.46.0 - released w/ FF38 support
- 2.47.0 - released w/ Edge support
- 2.48.0 - released w/ Marionette support in all languages
- 2.49.0 Released - w/ FF 43 support
- 2.50.0 Released - “It's all bloody edge cases!” - D.W-H
- 2.51.0 Released - “It's all bloody edge cases!” - D.W-H
- 2.52.0 Released - Now you can disable “all bloody edge cases!”
- 2.53.0 The FINAL RC RELEASE
- 3.0 The Christmas Release! FF48 now requires GeckoDriver
- 3.6 The “Not Released On A Friday” Release

Selenium Level Sponsors





Support the Selenium Project

Want to support the Selenium project? Learn more or view the full list of sponsors.

[LEARN MORE ▶](#)