The decision of using generic JavaScript that can drive the WAUT on any browser should comply with a security policy named **Same-Origin Policy**. Every available browser in the market imposes this policy on the websites that are loaded on it.

To know about this policy, we should take a closer look at how a browser executes JavaScript loaded from a website. For every website that is loaded on it, the browser creates a separate sandbox for the website's JavaScript, which restricts the JavaScript to be executed only on it's respective website domain. This way, a JavaScript that belongs to one website doesn't execute on another website that is currently loaded on that browser. This security vulnerability, named **Cross-site scripting**, is the browser's responsibility to restrict. So, coming back to Selenium RC, its generic JavaScript is not allowed, by the browser, to execute on a website (WAUT) that is coming from a different domain.

So, how did Selenium RC handle this? To overcome this security restriction, Selenium RC acts as an HTTP Proxy Server. When the test script asks to launch a browser, Selenium RC server launches the browser and injects its JavaScript (Selenium Core) into the browser. All the subsequent requests for the WAUT go through Selenium RC (acting as an HTTP Proxy Server) to the actual web server hosting WAUT. Thus making the browser think that the web application is being served from the Selenium RC's server domain than the actual web server's domain and allowing Selenium Core to execute and drive the web application.

Typically, it works in the following way:

1. A tester or a developer, through his/her test script, can command Selenium RC server to perform certain actions on the WAUT on a certain browser. The way the user can command Selenium RC to perform something is by using the client libraries provided by Selenium RC. These libraries are provided in different languages, such as Java, Ruby, Python, Perl, PHP, and .NET. These commands, which are passed from the test scripts to Selenium RC, are named **Selenese** commands. In a test script, you will have a set of Selenese commands to test a scenario on the WAUT.

2. Once the Selenium RC server receives the command from the test script, it will launch the test script preferred browser, and while launching, it injects the Selenium Core into the browser.

3 Upon loading on the browser, Selenium Core executes all the Selenese commands from the test script, coming through Selenium RC, against the WAUT. The browser doesn't restrict it, because it treats Selenium Core and WAUT as a part of the same domain

4 Now comes the HTTP Proxy part of the Selenium RC server. All the requests and responses of the browser for WAUT go to the actual web server via Selenium RC server, because the browser thinks Selenium RC is serving WAUT.

To overcome some of the limitations of Selenium 1, which we are going to discuss shortly, WebDriver has come into existence for the following reasons:

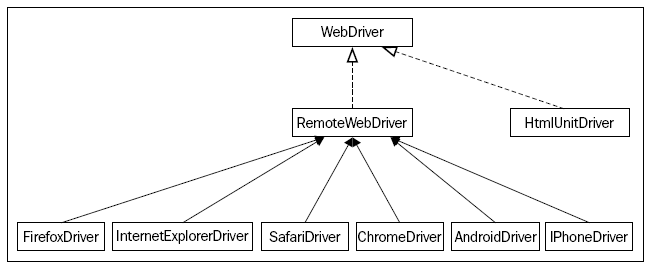
1. To give a better control on the browser by implementing browser-specific implementations.
2. To give a better programming experience to the developer by adhering more closely to the object-oriented programming fundamentals.

**Handling the browser**

As we saw earlier, Selenium RC drives the browser from within the browser by sitting in it as JavaScript (Selenium Core). All the events that are to be executed on the WAUT go through Core. This kind of approach will come with some limitations, such as:

* Core being limited within the JavaScript sandbox of the browser, as it needs to comply with the Same-Origin policy.
* Because this JavaScript library is generic and not specific to any particular browser, the developers of test scripts sometimes end up with a situation where their test scripts execute very well on some browsers but not on some other.

To overcome this limitation, WebDriver, on the other hand, handles the browser from outside the browser. It has an implementation for each browser, and the developer who wants to execute his/her tests on a particular browser should use that particular implementation of WebDriver. This gives the test scripts a better handle on the browser because these WebDriver implementations speak to the browsers natively, thus increasing the robustness of the test scripts.



SearchContext(I) > WebDriver (I) > RemoteWebDriver(C) > FirefoxDriver(C)

Firefox Driver extends Remote WebDriver

RemoteWebDriver implements :-

**implements** WebDriver, JavascriptExecutor,

FindsById, FindsByClassName, FindsByLinkText, FindsByName,

FindsByCssSelector, FindsByTagName, FindsByXPath,

HasInputDevices, HasCapabilities, TakesScreenshot

**The By.className() method**

Before we discuss about the className() method, we have to talk a little about style and CSS. Every HTML element on a web page, generally, is styled by the web page developer or designer. It is not mandatory that each element should be styled, but it is generally followed to make it appealing to the end user.

So, in order to apply styles to an element, they can be declared directly in the element tag or placed in a separate file called the CSS file and can be referenced in the element using the className() method. For instance, a style attribute for a button can be declared in a CSS file as follows:

.buttonStyle{

width: 50px;

height: 50px;

border-radius: 50%;

margin: 0% 2%;

}

Now, this style can be applied on the button element in a web page as follows:

<button name="sampleBtnName" id="sampleBtnId" class="buttonStyle">I'm Button</button>

So, buttonStyle is used as value for the class attribute of the button element, and it inherits all the styles declared in the CSS file.

XPath is a short name for the XML path. The HTML for our web page is also one form of the XML document.

The Shift key to type the text in uppercase in the Google Search Box:

searchBox.sendKeys(Keys.**chord**(Keys.SHIFT,"packt publishing"));

Simulate pressing many keys at once in a "chord".

**Void clear**() - This can be achieved using the Keys.BACK\_SPACE enum, but WebDriver has given us an explicit method to clear the text easily.

The **submit**() method :-

The submit action can be taken on a form or on an element, which is inside a form. This is used to submit a form of a web page to the server hosting the web application. a NoSuchElementException is thrown when this method is executed on a WebElement that is not present within a form.

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

WebElement searchBox = driver.findElement(By.name("q"));

searchBox.sendKeys(Keys.chord(Keys.SHIFT,"packt publishing"));

searchBox.submit();

**getLocation()**

This position is calculated relative to the top-left corner of the web page of which the (x, y) coordinates are assumed as (0, 0)

The API syntax of the getLocation() method is as follows:

Point getLocation()

**The getSize() method**

The API syntax of the getSize() method is as follows:

Dimension getSize() int i= element.getSize().height

**The moveByOffset action**

When the page is loaded, generally the initial position of a mouse would be (0, 0), unless there is an explicit focus declared by the page.

WebElement three = driver.findElement(By.name("three"));

System.out.println("X coordinate: "+three.getLocation().getX()+" Y coordinate: "+three.getLocation().getY());

Actions builder = new Actions(driver);

**builder.moveByOffset**(three.getLocation().getX()+1, three. getLocation().getY()+1);

x axis. A positive value is used to move the cursor to the right, and a negative value is used to move the cursor to the left.

The moveByOffset() method may not work in Mac OSX and may raise a JavaScript error when used independently like the previous code.

**relase(WebElement ele):-**

builder.clickAndHold(three)

.release(two)

.perform();

Using this, you can actually release the currently held WebElement in the middle of another WebElement. In this way, we don't have to calculate the offset of the target WebElement from the held WebElement.

**The sendKeys() method**

This is used to type in alphanumeric and special character keys into WebElements such as textbox, textarea, and so on. This is different from the WebElement. sendKeys(CharSequence keysToSend) method, as this method expects the WebElements to have the focus before being called.

Chapter 3

**Capabilities** is an interface in the WebDriver library whose direct implementation is the DesiredCapabilities class

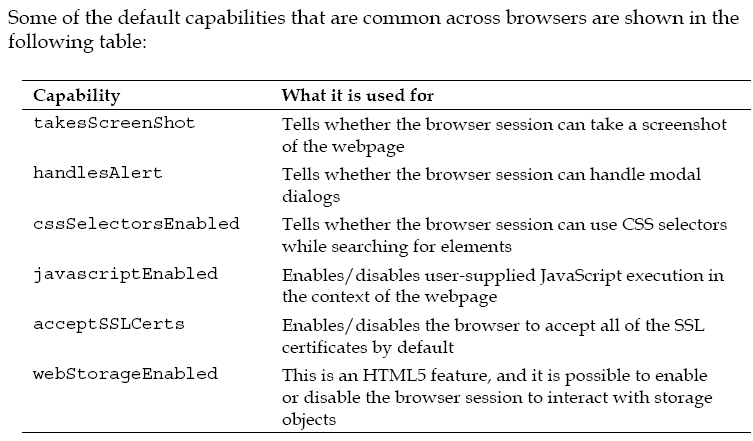
Map capabilitiesMap = new HashMap();

capabilitiesMap.put("takesScreenShot", true);

DesiredCapabilities capabilities = new DesiredCapabilities(capabilitiesMap);

WebDriver driver = new FirefoxDriver(capabilities);

In the preceding code, we set all of the capabilities that we desire in a map and created an instance of DesiredCapabilities using that map. Now, we have created an instance of FirefoxDriver with these capabilities. This will now launch a Firefox browser that will have support for taking screenshots of the webpage. If you see the definition of the DesiredCapabilities class, the constructor of the class is overloaded in many different ways. Passing a map is one of them. You can use the default constructor and create an instance of the DesiredCapabilities class, and then set the capabilities using the setCapability() method.



**Taking screenshots**

Taking a screenshot of a webpage is a very useful capability of WebDriver. This is very handy when your test case fails, and you want to see the state of the application when the test case failed. The TakesScreenShot interface in the WebDriver library is implemented by all of the different variants of WebDriver, such as Firefox Driver, Internet Explorer Driver, Chrome Driver, and so on.

The TakesScreenShot capability is enabled in all of the browsers by default. Because this is a read-only capability, a user doesn't have much say on toggling it

OutputType is another interface of the WebDriver lib. We can ask WebDriver to give your screenshot in three different formats; they are: BASE64, BYTES (raw data), and FILE. If you choose the FILE format, it writes the data into a .png file, which will be deleted once the JVM is killed. So, you should always copy that file into a safe location so that it can be used for later reference.

if you are using Firefox Driver, getScreenshotAs() takes the screenshot of the entire page, but Chrome Driver returns only the visible portion of the current frame.

**File scrFile = ((TakesScreenShot)driver). getScreenshotAs(OutputType.FILE);**

System.out.println(scrFile.getAbsolutePath());

we have used the getScreenshotAs() method to take the screenshot of the webpage and save it to a file format. The getAbsolutePath() method returns the path of the saved image, which you can open and examine

**Switching among windows**

Every time you open a web page using WebDriver in a browser window, WebDriver assigns a window handle to that. WebDriver uses this identifier to identify the window.

**Navigate**

Back() – may be used in banking apps, back should logout the page

first driver.navigate() returns the WebDriver.Navigation interface on which the to() method is used to navigate to a web URL.

The input parameter for this method is the url string that has to be loaded in the browser. This method will load the page in the browser by using the HTTP GET operation, and it will block everything else until the page is completely loaded. This method is the same as the driver.get(String url) method.

**Handling cookies**

Let's say you are automating the Facebook webpage. There could be many scenarios you want to automate, such as writing on your wall, writing on your friend's wall, reading other walls, adding friends, deleting friends, and so on. For all these actions, one common thing is to have to log in to Facebook in each of the test cases. So, logging in to Facebook in every test case of yours will increase the overall test execution time significantly. To reduce the execution time of your test cases, you can actually skip signing in for every test case. This can be done by signing in for one time and writing all the cookies of that domain into a file. From the next login onwards, you can actually load the cookies from the file and add to the driver.

To fetch all of the cookies that are loaded for a webpage, WebDriver provides the following method:

driver.manage().getCookies() a set of Cookie

driver.manage().addCookie(ck); Cookie ck

File f = new File("browser.data");

f.delete();

f.createNewFile();

FileWriter fos = new FileWriter(f);

BufferedWriter bos = new BufferedWriter(fos);

for(Cookie ck : **driver.manage().getCookies()**) {

bos.write((ck.getName()+";"+ck.getValue()+";"+ck. getDomain()+";"+ck.getPath()+";"+ck.getExpiry()+";"+ck. isSecure()));

bos.newLine();

}

bos.flush();

bos.close();

fos.close();

Reading and setting

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

File f = new File("browser.data");

FileReader fr = new FileReader(f2);

BufferedReader br = new BufferedReader(fr);

String line;

while((line=br.readLine())!=null){

StringTokenizer str = new StringTokenizer(line,";");

while(str.hasMoreTokens()){

String name = str.nextToken();

String value = str.nextToken();

String domain = str.nextToken();

String path = str.nextToken();

Date expiry = null;

String dt;

if(!(dt=str.nextToken()).equals("null")){

expiry = new Date(dt);

}

boolean isSecure = new Boolean(str.nextToken()). booleanValue();

Cookie ck = new Cookie(name,value,domain,path,expi ry,isSecure);

driver.manage().addCookie(ck);

}

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

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

Ideally, this line should be visible after we have set the cookies to the driver. But the reason it is at the top is because the WebDriver doesn't allow you to set the cookies directly into this session, because it treats those cookies as if they are from a different domain. Try removing the previous line of code and execute it, and you will see the error. So, initially you will try to visit the Facebook page to set the domain value of the driver to Facebook and load all of the cookies. When you execute this code, initially you will see the login page of Facebook, and you will be automatically taken to the home page when the same code at the end is invoked again after the cookies are loaded.

Chapter 5 :-

**FirefoxDriver**

The FirefoxDriver works as an extension to the Firefox browser. It uses the **XPCOM** (**Cross Platform Component Object Model**) framework of Mozilla to execute the commands sent by the language bindings. Language bindings communicate with the extension, that is, FirefoxDriver, by connecting over a socket and sending commands. This socket is bound to a port, which is called the locking port; typically, it would be 7055. The reason it is called the locking port is because it is used as a mutex so that it allows only one instance of Firefox to listen to a Firefox Driver on that port.

After this socket is established, the client language binding (in our case, the Java binding) sends the commands to the Firefox extension in a serialized JSON format. The JSON format contains the following components:

1. **Context**: This is the current window or frame
2. **CommandName**: For example, DragAndDrop, SendKeys
3. **Parameters**: This can be empty, or sometimes the text will need to be typed
4. **ElementId**: This is the ID of the element on which the action has to be performed

This serialized JSON is sent over the socket or wire established earlier to the Firefox Extension or FirefoxDriver. This is the reason Selenium-2 or WebDriver is said to be working on JSON-Wire protocol.

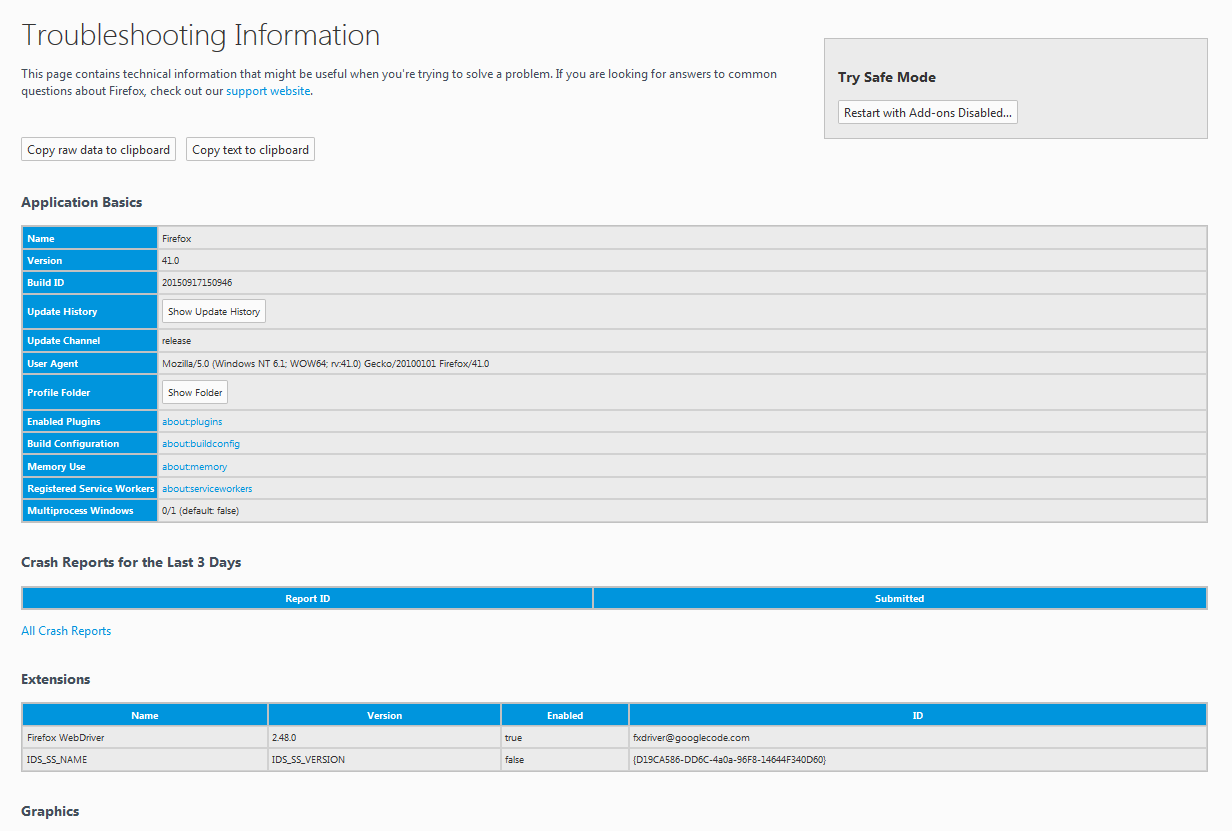
Once the commands reach from the client language bindings to the FirefoxDriver, it deserializes the JSON, and the commands are interpreted and looked up in the Firefox Driver prototype, which are the JavaScript functions for each command. After execution, the response is sent back via the socket to the client. This response is again a JSON that contains methodName (this is same as the commandName in the request), Context, isError (indicating if an error has occurred, so that the client can thrown an exception), and ResponseText (the output of the command executed).

**Understanding the Firefox profile:-**

According to Mozilla, the following are the different attributes that can be stored in the profiles:

1. Bookmarks and browsing history
2. Passwords
3. Site-specific preferences
4. Search engines
5. A personal dictionary
6. Autocomplete history
7. Download history
8. Cookies
9. DOM Storage
10. Security certificate settings
11. Security device settings
12. Download actions
13. Plugin MIME types
14. Stored sessions
15. Toolbar customizations
16. User styles

WebDriver, whenever we create an instance of FirefoxDriver, a temporary profile is created and used by the WebDriver. To see the profile that is currently being used by a Firefox instance, you have to navigate to **Help** | **Troubleshooting Information**.



Show Folder



Adding Furebug plugin to FF profile

*The following is the API syntax for the method:*

*public void addExtension(java.io.File extensionToInstall) throws java.io.IOException*

profile.addExtension(new File("C:\\firebug-1.12.0-fx.xpi"));

If WebDriver doesn't find the file in the specified location, it will raise an IOException.

**Storing and retrieving a profile – toJson, fromJson**

profile.addExtension(new File("C:\\firebug-1.12.0-fx.xpi"));

String json = profile.toJson();

FirefoxDriver driver = new FirefoxDriver(FirefoxProfile.fromJson(json));

In the preceding code, we have exported the profile as a JSON string. In your test case, you can write that JSON information to a file and store it. Later, you can read the JSON file

**Dealing with Firefox preferences**

According to Mozilla, a Firefox Preference is any value or defined behavior that can be set by a user. These values are saved to the preference files. If you open the profile directory by navigating to **Help** | **Troubleshooting Information** and clicking on the **Show Folder** button, you will see two preference files; they are prefs.js and user.js. All the user preferences are written to the prefs.js file by the Firefox application during the launch. A user can override those values to those of his/her choice, and they are stored in the user.js file. The value in user.js for a preference takes precedence over all the other values set for that particular preference. So, your FirefoxDriver overwrites all the default preferences of Firefox in the user.js file for you. When you add a new preference, FirefoxDriver writes that to the user.js preference file, and the Firefox browser behaves accordingly.

FirefoxProfile profile = new FirefoxProfile();

FirefoxDriver driver = new FirefoxDriver(profile);

In the preceding code, we are not setting any preferences, but it will still launch the Firefox browser. Now, open the user.js file in the profile directory. The following are the list of all the preferences that FirefoxDriver sets for you by default:

user\_pref("extensions.update.notifyUser", false);

user\_pref("security.warn\_entering\_secure.show\_once", false);

user\_pref("devtools.errorconsole.enabled", true);

user\_pref("extensions.update.enabled", false);

user\_pref("browser.dom.window.dump.enabled", true);

user\_pref("offline-apps.allow\_by\_default", true);

user\_pref("dom.disable\_open\_during\_load", false);

user\_pref("extensions.blocklist.enabled", false);

user\_pref("browser.startup.page", 0);

user\_pref("toolkit.telemetry.rejected", true);

user\_pref("prompts.tab\_modal.enabled", false);

user\_pref("app.update.enabled", false);

user\_pref("app.update.auto", false);

user\_pref("toolkit.networkmanager.disable", true);

user\_pref("browser.startup.homepage", "about:blank");

user\_pref("network.manage-offline-status", false);

user\_pref("browser.search.update", false);

user\_pref("toolkit.telemetry.enabled", false);

user\_pref("browser.link.open\_newwindow", 2);

user\_pref("browser.EULA.override", true);

user\_pref("extensions.autoDisableScopes", 10);

user\_pref("browser.EULA.3.accepted", true);

user\_pref("security.warn\_entering\_weak", false);

user\_pref("toolkit.telemetry.prompted", 2);

user\_pref("browser.safebrowsing.enabled", false);

user\_pref("security.warn\_entering\_secure", false);

user\_pref("security.warn\_leaving\_secure.show\_once", false);

user\_pref("webdriver\_accept\_untrusted\_certs", true);

user\_pref("browser.download.manager.showWhenStarting", false);

user\_pref("dom.max\_script\_run\_time", 30);

user\_pref("javascript.options.showInConsole", true);

user\_pref("network.http.max-connections-per-server", 10);

user\_pref("network.http.phishy-userpass-length", 255);

user\_pref("extensions.logging.enabled", true);

user\_pref("security.warn\_leaving\_secure", false);

user\_pref("browser.offline", false);

user\_pref("browser.link.open\_external", 2);

user\_pref("signon.rememberSignons", false);

user\_pref("webdriver\_enable\_native\_events", true);

user\_pref("browser.tabs.warnOnClose", false);

user\_pref("security.fileuri.origin\_policy", 3);

user\_pref("security.fileuri.strict\_origin\_policy", false);

user\_pref("webdriver\_assume\_untrusted\_issuer", true);

user\_pref("startup.homepage\_welcome\_url", "");

user\_pref("browser.shell.checkDefaultBrowser", false);

user\_pref("browser.safebrowsing.malware.enabled", false);

user\_pref("security.warn\_submit\_insecure", false);

user\_pref("webdriver\_firefox\_port", 7055);

user\_pref("dom.report\_all\_js\_exceptions", true);

user\_pref("security.warn\_viewing\_mixed", false);

user\_pref("browser.sessionstore.resume\_from\_crash", false);

user\_pref("browser.tabs.warnOnOpen", false);

user\_pref("security.warn\_viewing\_mixed.show\_once", false);

user\_pref("security.warn\_entering\_weak.show\_once", false);

This Firefox Driver treats them as Frozen Preferences and doesn't allow the test script developer to change them. However, there are a few preferences in the preceding list that FirefoxDriver allows you to change, which we will see shortly.

**Setting preferences**

Now we will learn how to set our own preferences. As an example, we will see how to change the user agent of your browser. Many web applications these days are have the main/normal site as well as the mobile site / m. site. The application will validate the user agent of the incoming request and accordingly decide whether to act as a server for a normal site or mobile site. So, in order to test your mobile site from your laptop or desktop browser, you just have to change your user agent. Let us see a code example where we can change the user agent preference of our Firefox browser using FirefoxDriver, and send a request to the Google Search page

FirefoxProfile profile = new FirefoxProfile();

profile.**setPreference**("general.useragent.override", "Mozilla/5.0 (iPhone; U; CPU iPhone OS 4\_0 like Mac OS X; en-us) AppleWebKit/532.9 (KHTML, like Gecko) Version/4.0.5 Mobile/8A293 Safari/6531.22.7");

Now open the user.js file for this particular Firefox instance, and you will see the entry for this preference. You should use the following preference in your user.js file:

user\_pref("general.useragent.override", "Mozilla/5.0 (iPhone; U; CPU iPhone OS 4\_0 like Mac OS X; en-us) AppleWebKit/532.9 (KHTML, like Gecko) Version/4.0.5 Mobile/8A293 Safari/6531.22.7");

**Understanding frozen preferences**

Now, let's go back to the big list of frozen preferences that user.js contains, which we have seen earlier. The Firefox Driver thinks that a test script developer doesn't have to deal with them and doesn't allow those values to be changed. Let us pick one frozen preference and try to change its values in our code. Let's consider the preference browser.shell.checkDefaultBrowser, whose value FirefoxDriver implementers thought should be set to false so that the Firefox browser does not ask you whether to make Firefox your default browser, if it is not already, while you are busy executing your test cases. Ultimately, you don't have to deal with the pop up itself in your test scripts. Apart from setting the preference value to false, the implementers of FirefoxDriver also thought of freezing this value so that users don't alter these values. That is the reason these preferences are called frozen preferences. Now, what happens if you try to modify these values in your test scripts? Let's see a code example:

FirefoxProfile profile = new FirefoxProfile();

profile.setPreference("browser.shell.checkDefaultBrowser", true);

FirefoxDriver driver = new FirefoxDriver(profile);

Now when you execute your code, you will immediately see an exception saying you're not allowed to override these values. The following is the exception stack trace you will see:



This is how FirefoxDriver mandates a few preferences that are not to be touched. However, there are a few preferences of our frozen list, which FirefoxDriver allows to alter through code. For that, it explicitly exposes methods in the FirefoxProfile class. Those exempted preferences are for dealing with SSL certificates and native events. Here, we will see how we can override the SSL certificates' preferences.

Let's use a code example that tries to override the default Firefox behavior to handle SSL certificates. The FirefoxProfile class has two methods to handle the SSL certificates; the first one is as follows:

profile.setAssumeUntrustedCertificateIssuer(false);

profile.setAcceptUntrustedCertificates(false);

Here, we have set the values to false, and now if we open the user.js file in the profile directory of this instance of Firefox, you will see the values set to false, as follows:

user\_pref("webdriver\_accept\_untrusted\_certs", false);

user\_pref("webdriver\_assume\_untrusted\_issuer", false);

FirefoxDriver enables native events to be invoked in the Firefox browser by exposing a method named setEnableNativeEvents(). Using this method, you can override the preference webdriver\_enable\_native\_events in the user.js file.

**Firefox binary**

Imagine a situation where you have to test your web application against two different versions of the Firefox browser. By default, when you instantiate FirefoxDriver, the Firefox version that is available on the PATH variable is launched. But if you want to launch a different version of Firefox, we need to use Firefox Binary.

**Installing multiple versions of Firefox**

Now that you have Firefox 17.0.1 version on your machine, let's install Firefox 23.0 version on your system by performing the following steps:

1. Download this version from Mozilla and start installing it.
2. When you reach the following screen in your installation, select **Custom** and install it to a custom path
3. 

Now try to launch Firefox from your code; it will launch Firefox 17.0.1 as it is available in the PATH variable.

So, in order to use Firefox 23.0, try to use Firefox Binary. The following is the code example for it:

**FirefoxBinary binary = new FirefoxBinary(new File("C:\\Mozilla Firefox\\firefox.exe"));**

FirefoxProfile profile = new FirefoxProfile();

**FirefoxDriver driver = new FirefoxDriver(binary, profile);**

**InternetExplorerDriver**

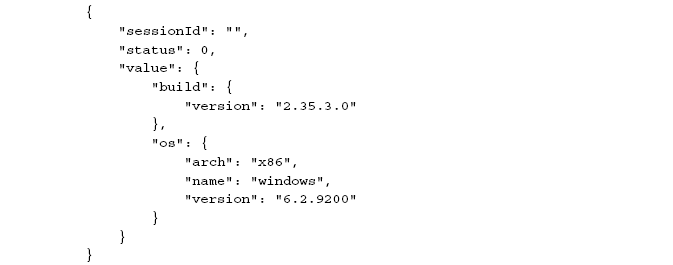
Download the 32-bit or 64-bit version based on the Internet Explorer that is installed on your computer.

Double-click on the file to launch WebDriver's InternetExplorerDriver as a service that listens on a port shown as follows:



Here, the InternetExplorerDriver is running as the service on port 5555.

Start your server, and it might start on a different random port. Now point your browser to http://localhost:5555/status and it will return the details of the IE Server as JSON in the following manner:



Until now, you have started IEDriver as a service that listens on a port and can be communicated over HTTP. Your client library constructs all your test script commands as JSON and hands it over to the IEDriver server. This again confers to JSONWireProtocol. The IEDriver server then uses its IEThreadExplorer class, which is written in C++, to drive the IE browser using the Component Object Model framework. Almost all browsers are written in C++

System.setProperty("webdriver.ie.driver", "C:\\IEDriverServer\_Win32\_2.35.3\\IEDriverServer.exe");

If you are wondering where you can find the list of all the properties, here is the location: http://selenium.googlecode.com/git/docs/api/java/constant-values.html#org.openqa.selenium.ie.InternetExplorerDriver.NATIVE\_ EVENTS

Protected Mode:-

IE7 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 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 order to work around that problem, IEDriver dictates that to work with IE, all zones must have the same Protected Mode setting.

The IEDriver server provides a way for the test script developer to configure it; that is, the port it should run on, the location where the temporary files should be extracted, and so on via the client library. The InternetExplorerDriverService. Builder class can be used to achieve this.

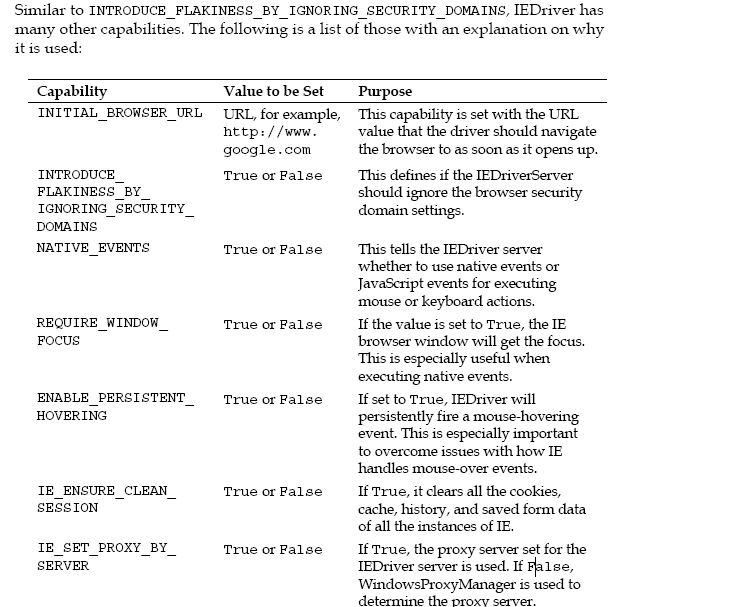
The IEDriverServer is started on a different random port. Suppose you want to make sure your server always started on the same port, you can do that using this builder class. Similarly, if you want to execute your tests pointing to an IEDriver server running on a different machine, you can do that as well by pointing to the machine's IP address

void quit()

This will kill the driver, driver's server, deletes temp files, and all associated browser windows for you.

Close()

IEDriverServer.exe is not shut down, server is still running – used in pop up window switching



**Using ChromeOptions**

ChromeOptions are similar to Firefox profiles. You can add extensions to your Chrome browser, specify the binary location of the Chrome browser if you have multiple versions of Chrome browsers installed on your machine, and so on

ChromeDriverService.Builder builder = **new** ChromeDriverService.Builder();

ChromeDriverService srvc = builder.usingDriverExecutable(**new** File("C:\\chromedriver\_win32\_2.2\\chromedriver.exe")).usingPort(65423).build();

Builder is a static inner class

ChromeOptions opts = **new** ChromeOptions();

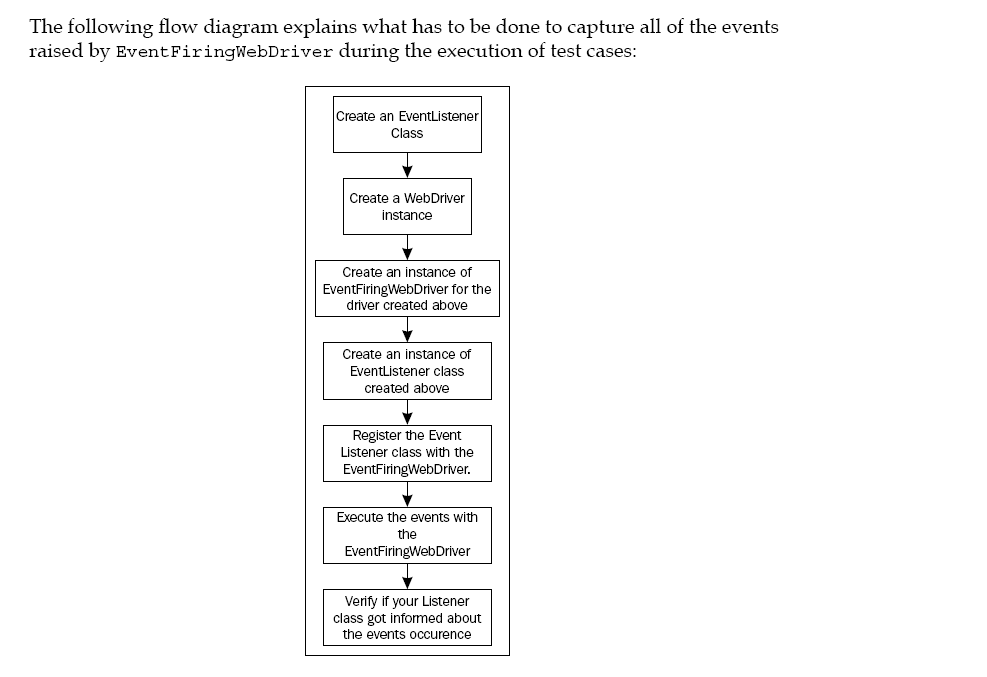
opts.addExtensions(**new** File("C:\\firebug.crx"));

**SafariDriver**

SafariDriver is implemented as an extension to the Safari browser. SafariDriver communicates with this extension using web sockets, which is slightly different in implementation from the rest of the WebDrivers. SafariDriver comes bundled default with the Selenium.jar file just as with FirefoxDriver. You do not have to download it separately.But we need extz

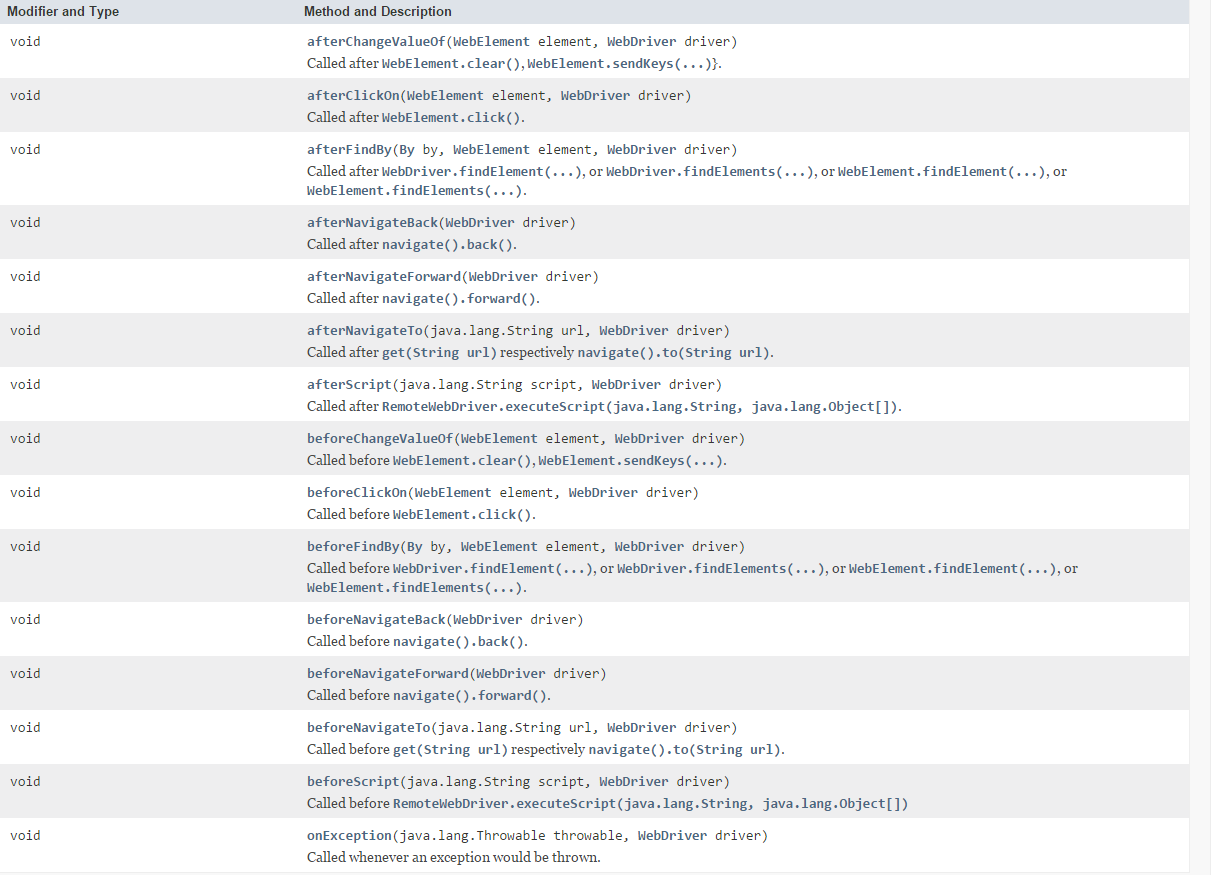
Chapter 5

Selenium WebDriver provides a very good framework for tracking the various events that happen while you're executing your test scripts using WebDriver. Many navigation events that get fired before and after an event occurs (such as before and after navigating to a URL, before and after browser back-navigation, and so on) can be tracked and captured. To throw an event, WebDriver gives you a class named EventFiringWebDriver, and to catch that event, it provides an interface named WebDriverEventListener. The test script developer should provide their own implementations for the overridden methods from the interface.



The EventFiringWebDriver class is a wrapper around your normal WebDriver that gives the driver the capability to fire events. The EventListener class, on the other hand, waits to listen from EventFiringWebDriver and handles all of the events that are dispatched. There can be more than one listener waiting to hear from the EventFiringWebDriver class for an event to fire. All of the event listeners should be registered with the EventFiringWebDriver class to get notified.

findeEle, sendkeys,click, to,back,forward, script onException



Clear and sendkeys for changeValueOf

public void onException(java.lang.Throwable throwable, WebDriver driver)

This event occurs when the WebDriver comes across some exceptions. For instance, if you try to search for a WebElement using findElement(), and that element doesn't exist on the page, the driver throws an exception (NoSuchElementException). At this point, this event is triggered, and the following method gets notified:

In all the after<<event>> methods, we have seen that they will not be invoked if the driver comes across any exception. In that case, instead of those after<<events>> methods, the onException() method is invoked and the throwable object and the WebDriver object are sent to it as parameters.

WebDriver webDriver= **new** FirefoxDriver();

EventFiringWebDriver driver=**new** EventFiringWebDriver(webDriver);

IAmTheEventListener eventListener1=**new** IAmTheEventListener();

IAmTheEventListener2 eventListener2=**new** IAmTheEventListener2();

driver.register(eventListener1);

driver.register(eventListener2);

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

driver.unregister(eventListener2);

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

driver.quit();

adds all unimplemented methods

**public** **class** IAmTheEventListener **implements** WebDriverEventListener {………}

Though it doesn't really provide any implementation for the methods in the WebDriverEventListener interface, it creates a dummy implementation such that the listener class that you are creating doesn't have to contain all of the methods; only the ones that you, as a test script developer, are interested.

**public** **class** IAmTheEventListener2 **extends** AbstractWebDriverEventListener{

@Override

**public** **void** afterNavigateTo(String url, WebDriver driver) {

System.***out***.println("afterNavigateTo of Listener 2:-"+driver.getCurrentUrl());

}

}

Chapter 6

\* createDirec, delete, copy

\* isZipped, readAsString

\* can, make

File Handler // Selenium api

FileHandler.*copy*(**new** File("src/source"), **new** File("src/dest")); // all files from source to dest

FileHandler.*copy*(**new** File("src/source"), **new** File("src/dest"),"1.txt");//all files‘suffix’ with 1.txt

FileHandler.*createDir*(**new** File("C:/SelDir"));

FileHandler.*delete*(**new** File("C:/SelDir"));

FileHandler.*isZipped*("C:/SelDir/some.zip"); //boolean true

FileHandler.*isZipped*("C:/SelDir");

String s=FileHandler.*readAsString*(**new** File("C:/SelDir/some.zip"));

FileHandler.*makeExecutable*(**new** File("C:/SelDir/some.zip")); //properties of a file- read ,read ,'execute'

FileHandler.*makeWritable*(**new** File("C:/SelDir/some.zip"));

FileHandler.*canExecute*(**new** File("C:/SelDir/some.zip")); //boolean

Zip //Selenium Api

Zip zip = **new** Zip();

zip.zip(**new** File("C:\\TmpFS"), **new** File("C:\\TmpFS.zip"));

zip.unzip(**new** File("C:\\TmpFS.zip"), **new** File("C:\\TmpFSExtracted\\"));

Client

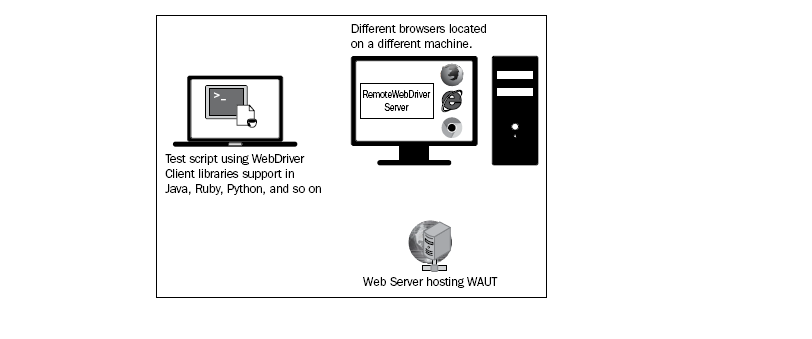
* Methods 🡪Json🡪 req sent to appropriate API 🡪 Json to Object at Driver.exe🡪Native

Sel-java

Chapter 7:-

Selenium-standalone-server

The test script is located on a local machine, while the browsers are installed on a remote machine. In this scenario, RemoteWebDriver comes into the picture. There are two components associated with RemoteWebDriver: the server and the client. Let us start with the RemoteWebDriver server.

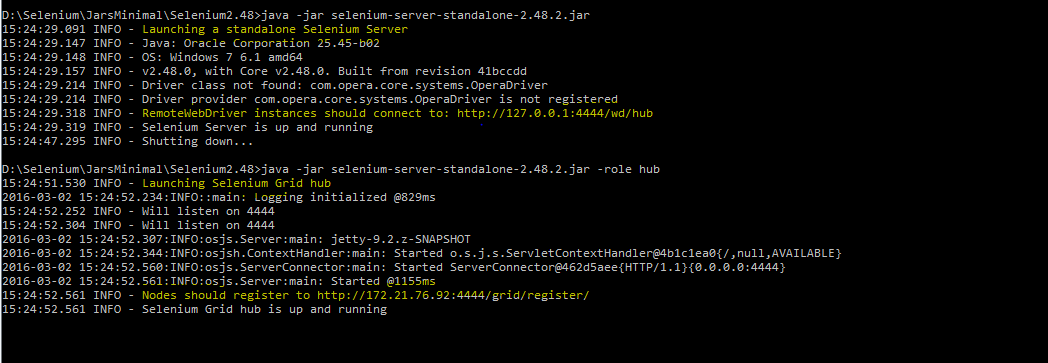


The RemoteWebDriver **server** is a component that listens on a port(4444) for various requests from a RemoteWebDriver client. Once it receives the requests in the form of JSON over the socket, it forwards them to any of the following: Firefox Driver, IE Driver, or Chrome Driver, as JSON request over the socket

**Running the server**

Open your command-line tool on the remote machine and navigate to the location to which you have downloaded the JAR file. Now, to start the RemoteWebDriver server, execute the following command:

**java –jar selenium-server-standalone-2.45.0.jar**



Now, the server has started and is listening on the <remote-machine-ip>:4444 address for remote connections from the RemoteWebDriver client.

**Understanding the RemoteWebDriver client**

Fortunately, we don't have to do anything much to create a RemoteWebDriver client. It's nothing but the language-binding client libraries that serve as a RemoteWebDriver client. The client, as it used to when executing tests locally, translates your test script requests to JSON payload and sends them across to the RemoteWebDriver server using the JSON wire protocol.

When you execute your tests locally, the WebDriver client libraries talk to your Firefox Driver, IE Driver, or Chrome Driver directly.

Now, when you try to execute your tests remotely, the WebDriver client libraries talk to the RemoteWebDriver server and the server talks to either the Firefox Driver, IE Driver, or Chrome Driver

RemoteWebDriver(java.net.URL remoteAddress, Capabilities desiredCapabilities)

DesiredCapabilities capabilities = new DesiredCapabilities();

RemoteWebDriver remoteWD =new RemoteWebDriver(new URL("http://10.172.10.1:4444/wd/hub"), capabilities);

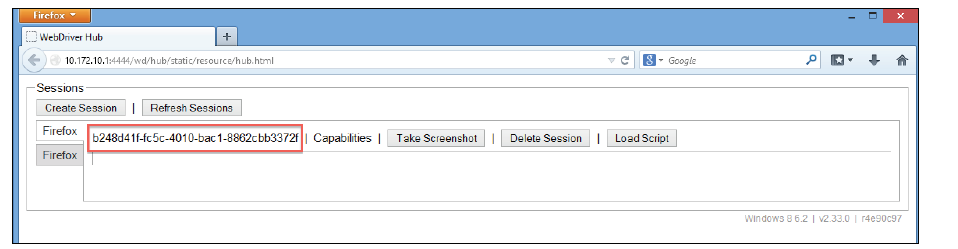
We have created a RemoteWebDriver instance that tries to connect to http://10.172.10.1:4444/wd/hub, where the RemoteWebDriver server is running and listening for requests. Having done that, we also need to specify which browser your test case should get executed on. This can be done using the DesiredCapabilities instance

Now execute this test script from your local machine to establish a connection between the RemoteWebDriver client and the RemoteWebDriver server. The RemoteWebDriver server will launch the Firefox browser. The following is the output you will see in the console where the RemoteWebDriver server is running:

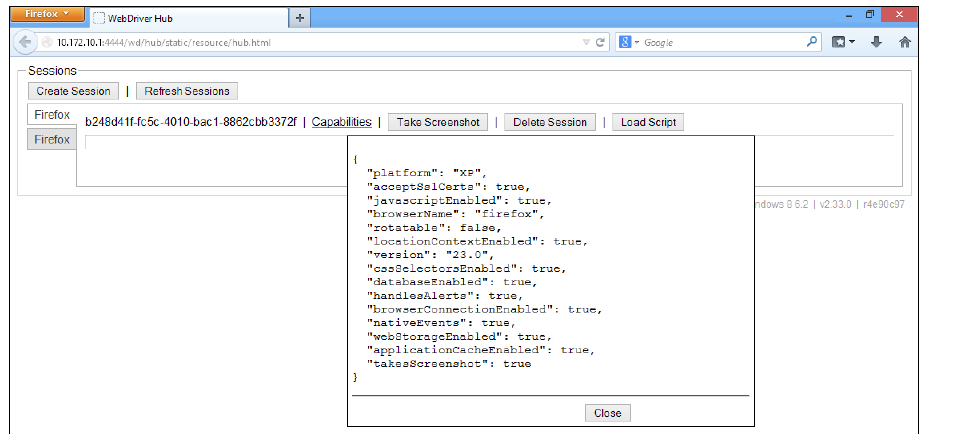


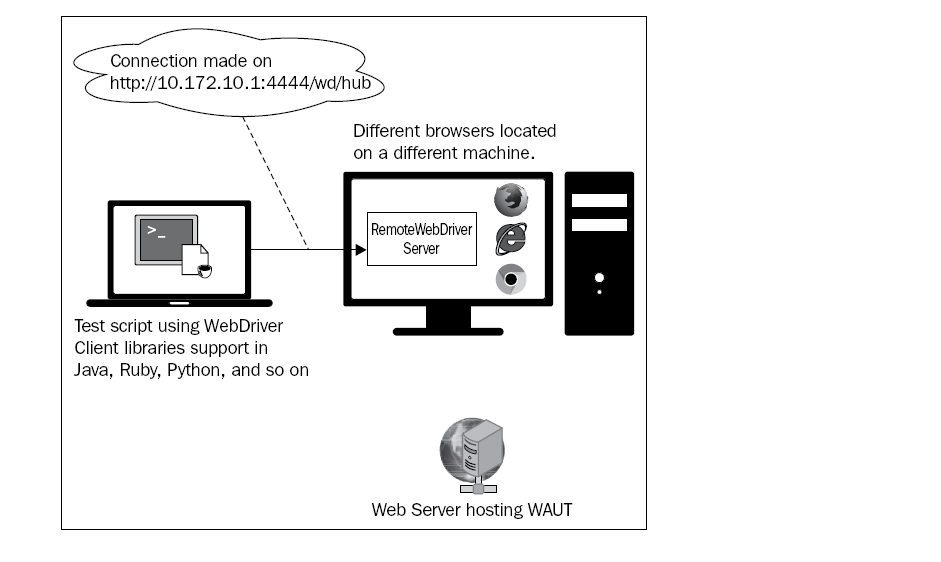
It says that a new session with the desired capabilities is being created, which, after being created, prints the session ID on to the console. At any point in time, you can view all of the sessions that are established with the RemoteWebDriver server by navigating to http://10.172.10.1:4444/wd/hub.

It will give the entire list of sessions that the RemoteWebDriver server is currently handling. (Not available with grid)



This is a very basic portal that lets the test script developer see all of the sessions created with the RemoteWebDriver server and perform some basic operations on it, such as terminating a session, taking a screenshot of a session, loading a script to a session, and seeing all of the desired capabilities of a session. The following screenshot shows all of the default desired capabilities of our current session.

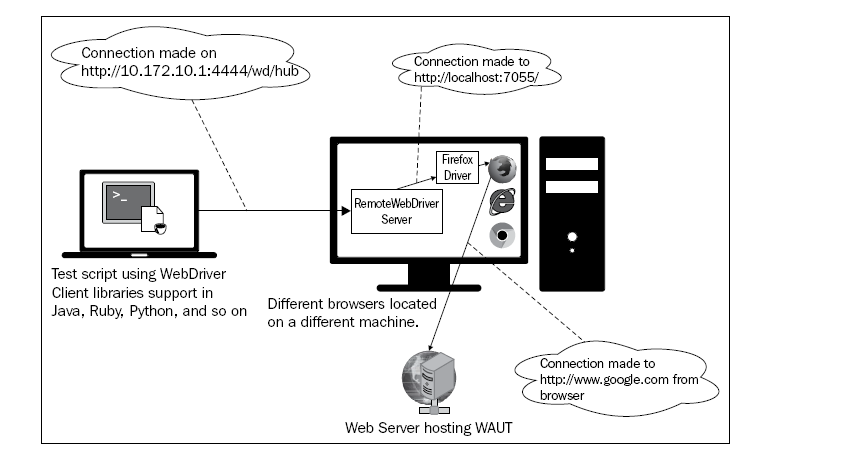




As soon as the server receives a request on port 4444, it verifies which browser has asked for the desired capabilities. When the server figures out that the request is for the Firefox browser, it launches the Firefox Driver as an extension to the Firefox browser, as discussed in *Chapter 4*, *Different Available WebDrivers*.

The RemoteWebDriver server opens a socket connection, usually to the Firefox Driver, on port 7055. From then on, all of your test script commands are handed over by the RemoteWebDriver server to Firefox Driver through this socket. So, from where did the RemoteWebDriver server find the Firefox Driver? Firefox Driver comes along with the RemoteWebDriver server JAR file. You don't have to download or start it explicitly, unlike with IE Driver or Chrome Driver.

Now, our initial diagram of the process of running test scripts looks as follows:



**Extending the RemoteWebDriver client to take screenshots**

If you compare the signatures of RemoteWebDriver to Firefox Driver and others you will observe that all of the other drivers implement the TakesScreenshot interface. But, if you try to do the same thing using the instance of a RemoteWebDriver, your test script will fail, throwing a ClassCastException. RemoteWebDriver doesn't implement the TakesScreenshot interface.

The first approach is to create your own WebDriver class that extends the RemoteWebDriver class and implements the TakesScreenshot interface

The second approach is to use the Augmenter class. This will enhance the RemoteWebDriver instance based on the set DesiredCapabilities. This is still in its early stages of implementation, so using it may result in unexpected results sometimes.

**remoteWD = new Augmenter().augment(remoteWD);**

File scrFile = ((TakesScreenshot)remoteWD). getScreenshotAs(OutputType.FILE);

**Understanding the JSON wire protocol**

https://code.google.com/p/ selenium/wiki/JsonWireProtocol

**JavaScript Object Notation** (**JSON**) is used to represent objects with complex data structures. It is used primarily to transfer data between a server and a client on the web. It has very much become an industry standard for various REST web services, playing a strong alternative to XML.

A sample JSON file, saved as a .json file, will look as follows:

{

"firstname": "John",

"lastname": "Doe",

"address": {

"streetnumber":"678",

"street":"Victoria Street",

"city":"Richmond",

"state":"Victoria",

"country":"Australia"

}

"phone":"+61470315430"

}

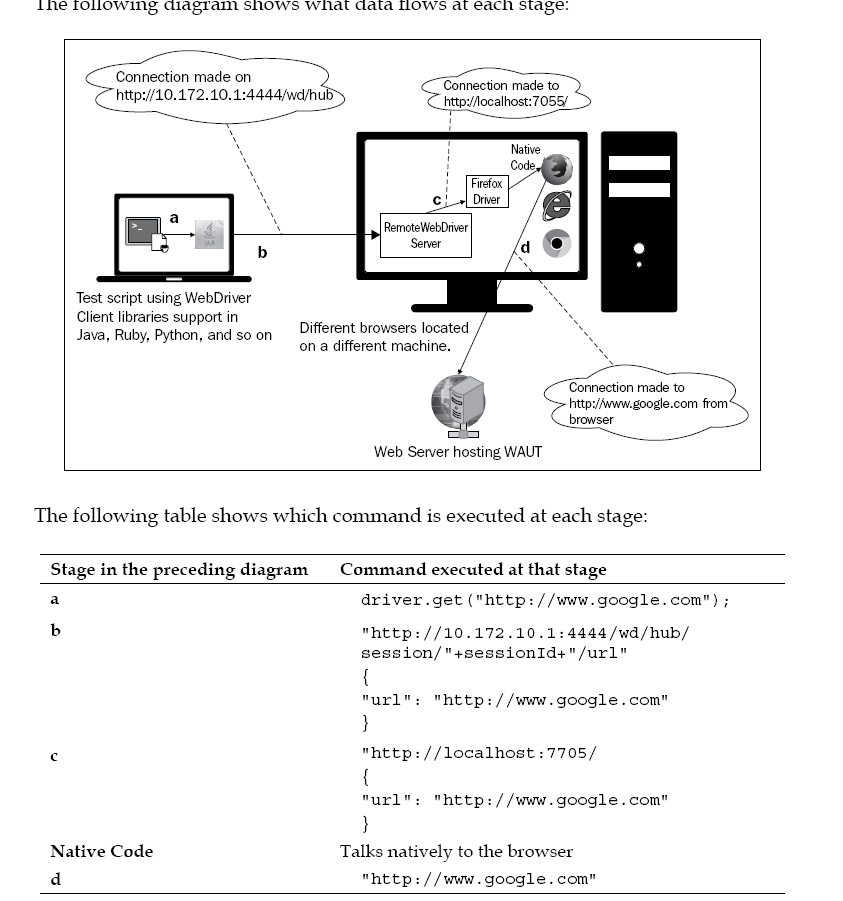
A client can send a person's details to a server in the preceding JSON format, which the server can parse and create an instance of the Person object for use in its execution. Later, the response can be sent back by the server to the client in the JSON format, the data of which the client can use to create an object of a class. This process of converting an object's data to the JSON format and JSON-formatted data to an object is named **serialization** and **de-serialization**, respectively, which is quite common in REST web services these days.

Our WebDriver uses the same approach to communicate between client libraries and drivers. Similarly, the RemoteWebDriver client and the RemoteWebDriver server use the JSON wire protocol to communicate among themselves.

The complete documentation is available at https://code.google.com/p/ selenium/wiki/JsonWireProtocol.



The **client libraries** will translate your test script commands to the JSON format and **send the requests to the appropriate WebDriver API**. The WebDriver will parse these requests and take necessary actions on the web page.



**WebDriverBackedSelenium**

The main reason you modify your test scripts to use WebDriverBackedSelenium is because if you want to extend or implement new test scripts, from now on, you can use WebDriver APIs while not breaking the existing stuff.

In the Selenium 1 code, consider the following lines of code:

Selenium sel = new DefaultSelenium("localhost",4444,"\*firefox", "");

sel.start();

They will be replaced with:

WebDriver driver = new FirefoxDriver();

Selenium sel = new WebDriverBackedSelenium(driver, http://www.google.com); //2 arguments

From that point forward, the rest of the test script commands will go to the DefaultSelenium instance via the WebDriverBackedSelenium class as it extends the DefaultSelenium class. At this point, if you want to extend your test scripts to use some of the WebDriver APIs, you can use the following method to get the underlying WebDriver:

public WebDriver getWrappedDriver()

replacing the DefaultSelenium code with the WebDriverBackedSelenium code will not make your existing test script commands use WebDriver APIs; they still go through the Selenium 1 libraries, and you need to replace those methods with the WebDriver API methods. However, using the getWrappedDriver() method of WebDriverBackedSelenium, you can extend your test script to use the WebDriver APIs

Chapter 7:-

**Understanding the hub**

The hub is the central point of a Selenium Grid. It has a registry of all the available nodes that are part of a particular grid. The hub is again a Selenium server running in the hub mode listening on port 4444 of a machine by default. The test scripts will try to connect to the hub on this port, just as any Remote WebDriver. The hub will take care of rerouting the test script traffic to the appropriate test platform node.

**Queuing up the request if the node is busy**

By default, you can send five test script requests to any node. Although it is possible to change that configuration, let us see what happens when a node is already serving five requests, and you fire up another request for that node via the hub. The hub will keep polling the node until it gets a free test slot from the node. The test scripts are made to wait all this while. The log output you will see on the console for the sixth request would be as follows:



The hub says there no free slots for the sixth session to be established with the same node, and the Grid Console UI on the browser says that too, as shown in the following screenshot: 

Upon serving the five test script requests, the hub will establish the waiting sixth session with the node, and the sixth request will be served.

**Dealing with two nodes with matching capabilities**

When two nodes of the same capabilities are registered with a hub, a test script request receives the node that is registered first with the hub. If the first registered node is busy handling other test script requests, only then the hub directs the request to the second node with matching requested capabilities.

**Setting node timeouts( -nodeTimeOut) seconds**

java -jar selenium-server-standalone-2.33.0.jar -role node -hub http://172.16.87.131:1111/grid/register **-nodeTimeout 300**

The hub will terminate the test script if it doesn't perform any activity on the node for more than 300 seconds

**Setting node health-check time (-nodePolling) seconds**

Using this configuration parameter, we can specify how frequently the hub can poll a node for its availability. The parameter that is used to achieve this is nodePolling. By specifying this to the hub at the node level, each node can specify its own frequency at which it can be health checked. The command to configure your node is as follows:

java -jar selenium-server-standalone-2.33.0.jar -role node -hub http://172.16.87.131:1111/grid/register **-nodePolling 10**

Now, the hub will poll this node every 10 seconds to check its availability.

**Setting the browser timeout (-browserTimeout) seconds**

This configuration is to let the node know how long it should wait before it ends a test script session when the browser seems to hang. Beyond this time, the node will abort the browser session and start with the next waiting test script. The configuration parameter for this is browserTimeout. The command to specify that is as follows:

**java -jar selenium-server-standalone-2.33.0.jar -role node -hub http://172.16.87.131:1111/grid/register –browserTimeout 60**

**Reregistering the node automatically (-registerCycle) milliseconds**

If the hub crashes or restarts after a node registers to it, all the information of the nodes that are already registered is lost. Going back to each of the nodes and reregistering them manually would prove to be tedious. The impact will be even more if we haven't realized that the hub has restarted because all the test scripts would fail as a result. So, to handle this kind of situation, Selenium Grid provides a configuration parameter to a node through which we can specify the node to reregister itself automatically to the hub after a specified amount of time. If not specified, the default time of reregistration is five seconds. This way, we really don't have to worry; even if the hub crashes or restarts, our node will try to reregister every five seconds.

java -jar selenium-server-standalone-2.33.0.jar -role node -hub http://172.16.87.131:1111/grid/register **-registerCycle 10000**

The output you will see on the node log console during startup is as follows:

17:47:01.231 INFO - starting auto register thread. Will try to register every 10000 ms.

**WaitTimeout for a new session (-newSessionWaitTimeout) milliseconds**

When a capability-matched node is busy executing other test scripts, the latest test script will wait for the node to be available. By default, there is no wait timeout; that is, the test script will wait for the node to be available indefinitely. To alter that behavior and to let the test script throw an exception if it doesn't get the node within a limited time, Selenium Grid opens a configuration that enables the test script to do so. The configuration parameter controlling that behavior is newSessionWaitTimeout. The command for that is as follows:

java -jar selenium-server-standalone-2.33.0.jar -role hub -port 1111 **-newSessionWaitTimeout 120000**

**Different ways to specify the configuration (-nodeConfig) .json**

There are two ways to specify the configuration parameter to the Selenium Grid'. hub and node. The first one is what we have been seeing all this time; that is, specifying the configuration parameters over the command line. The second way of doing it is providing a JSON file that contains all these configuration parameters.

Once node and hub json files are configured, they can be provided to the node and hub using the following command:

java -jar selenium-server-standalone.jar -role node  **-nodeConfig nodeconfig.json**

java -jar selenium-server-standalone.jar-role hub **-hubConfig hubconfig.json**