INCLUDES REUSABLE CODE REPOSITORY

SELENIUM 4 A QUICK AND PRACTICAL GUIDE

BY
PALLAVI SHARMA
&
ADITYA GARG

SPONSORED BY AGILE TESTING ALLIANCE

Selenium 4 A Quick and Practical Guide

Authors:

Pallavi Sharma
Aditya Garg

Sponsored by:

AGILE TESTING ALLIANCE

Preface

Selenium 4 which is still in beta phase (at the time of writing this) has been in the making for some time. We had been thinking about putting this e-book together for past many months. There are lot of articles on the web which highlight the differences and improvements in Selenium 4 but there is no consolidation in the form of a book and there is no ready to use code. Ours is a humble attempt to put together this simple practical guide on the changes and improvements brought in Selenium 4. Ready to use code is available on ATA's (Agile Testing Alliance) GitHub account shared in below section. We hope that readers will be able to reuse the code and experiment with the new features of Selenium 4.

We dedicate this book to Tokyo 2021 Indian Olympic winners Neeraj Chopra, Mirabai Channu, Ravi Dhaiya, Lovlina Borgohain, PV Sindhu, Bajran Punia, the entire Indian Hockey Team Men. Their dedication, and spirit in these challenging times gave us all a reason to celebrate and cheer! This book is dedicated to also to every Tester out there, who keeps thriving, and never stops learning.

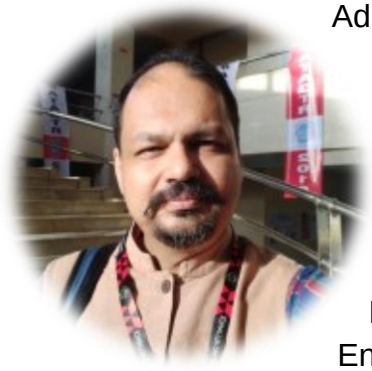
We would like to thank Agile Testing Alliance for sponsoring the book. Special thanks to Harsh Shah and Sanjivv Ssharma for helping us review the book.

Authors Profile

Pallavi Sharma

Pallavi is a multiskilled professional and has donned many hats in her career span. She founded 5 Elements Learning, where she acted as a coach, writer, speaker on test automation solutions and collaborated with learning enthusiasts, organizations, and mentors from across the globe. She is working with Agile Testing Alliance as a Steering Committee member. She is a contributor for Selenium Documentation, at the Selenium Project. She is also the author of the Selenium with Python Beginners book with BPB publications. She has also curated, organized, and acted as a jury for various international conferences and meetups. She is a firm believer in larger good and likes to live by example. She volunteers her resources for Jabarkhet forest reserve, People for animals, and Wildlife SOS. She lives in the National Capital Region with her doctor parents, her husband, two human children [sometimes she wonders though!] and a Labrador. She is also a budding writer and likes to write short stories and try her hand at poetry. Pallavi would like to thank all the wonderful people behind the Selenium project and the entire Selenium community and dedicate this e book to them.

Aditya Garg



Aditya is CEO AvanteNow.com, Steering Committee Member Agile Testing Alliance, Founding Director of QAAgility Technologies private limited. Adi as he fondly wants his friends to call him is an Agile and DevOps coach and loves taking test automation and DevOps solutioning work. He is a ServiceNow enthusiast and loves everything about the tool. Aditya did his Master of Business administration from UMASS Lowell and Engineering in Computer Science from MBM Engineering college Jodhpur. He has a passion for community work. He keeps on taking training/coaching session for Agile Testing Alliance community. He also loves curating and running conferences and meetups. Aditya has a passion for two things, testing is obviously one of them and another is cooking. Aditya was born and brought up in the princely town of Jodhpur Rajasthan India. After spending 25 years of career travelling and staying in different parts of India and world, he is now happily settled back in Jodhpur. If you ever plan on travelling and visiting this beautiful part of India, feel free to ping him and meet him. Aditya would like to thank his wife Prachi Garg, and three lovely Daddy's daughters Arunima, Divi and Dyuti for all the strength they are and to bear with his cooking experiments.

TABLE OF CONTENTS

Chapter 1: Introduction

Selenium is a set of tools to automate the browser. What we do with that, is entirely up to us. A large user base of Selenium uses it to create test scripts to automate the testing process for web browser-based applications. Selenium project consists of three sub projects. **The Selenium IDE, Selenium WebDriver, and the Selenium Grid**.

The table below shows where and how which sub project is used.

Selenium Project	Usage
Selenium IDE	The IDE is used for recording and playing back of scripts in Selenium on browser - Chrome and Firefox.
Selenium WebDriver	The Selenium WebDriver allows us with various functions to automate end to end browser actions. It comes with a support in various programming language bindings.
Selenium Grid	The Selenium Grid allows us to run our tests in parallel by setting up the grid environment.

1.1 Git Repository of the book

This book is written to empower the selenium enthusiasts with the new changes which are upcoming in the selenium version 4. The last released stable version of selenium is 3.141.59. At the time of writing the book, the selenium version 4 had a beta release, and the stable release of this was expected soon. **For this book, we have used the Selenium - 4.0.0-beta-3 release**. We have used Java as a programming language and eclipse as the IDE for creation of our scripts.

You can clone the repository to get code files. The complete project in java is available here https://github.com/AgileTestingAlliance/Selenium4EBook.git

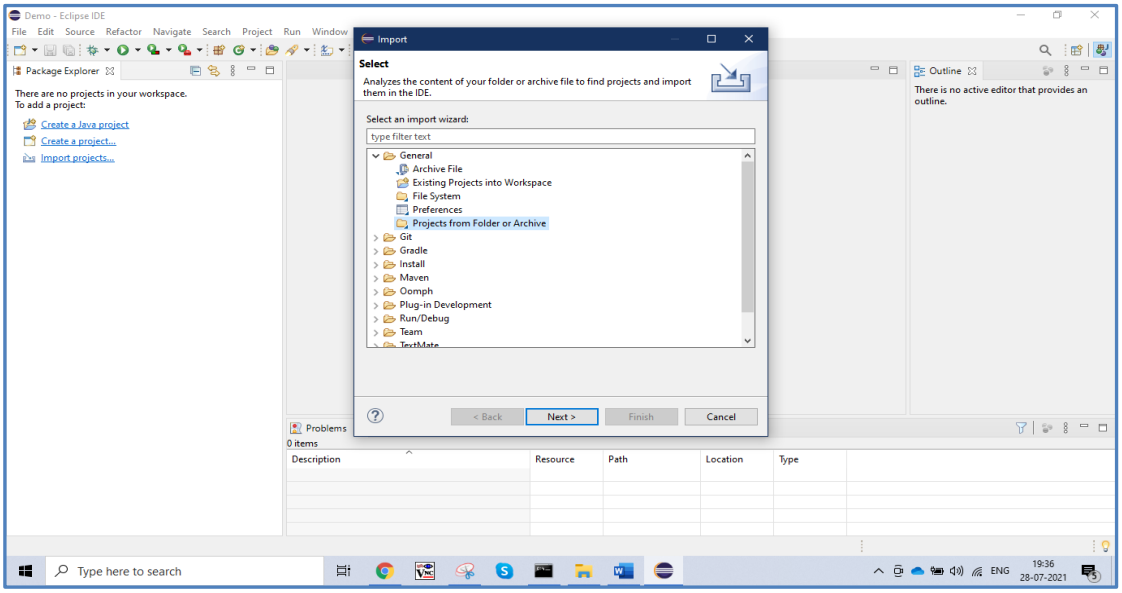
1.2 Cloning and importing the Git repository in Eclipse project

You can clone the repository to get the project. We will use the command

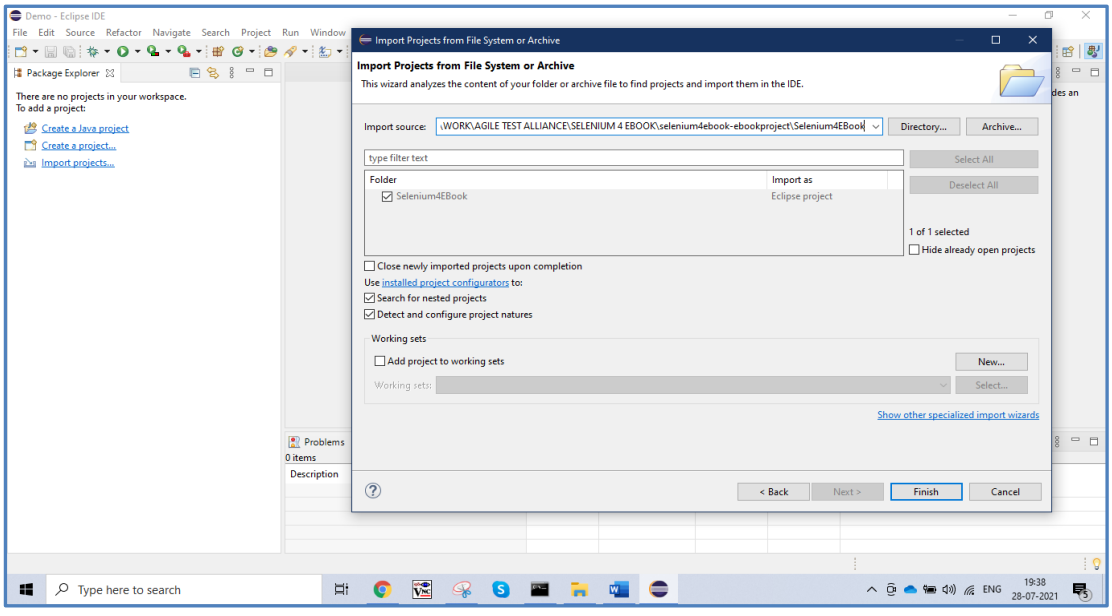
git clone https://github.com/AgileTestingAlliance/Selenium4EBook.git

a. Create a workspace in eclipse or you can use an existing one.

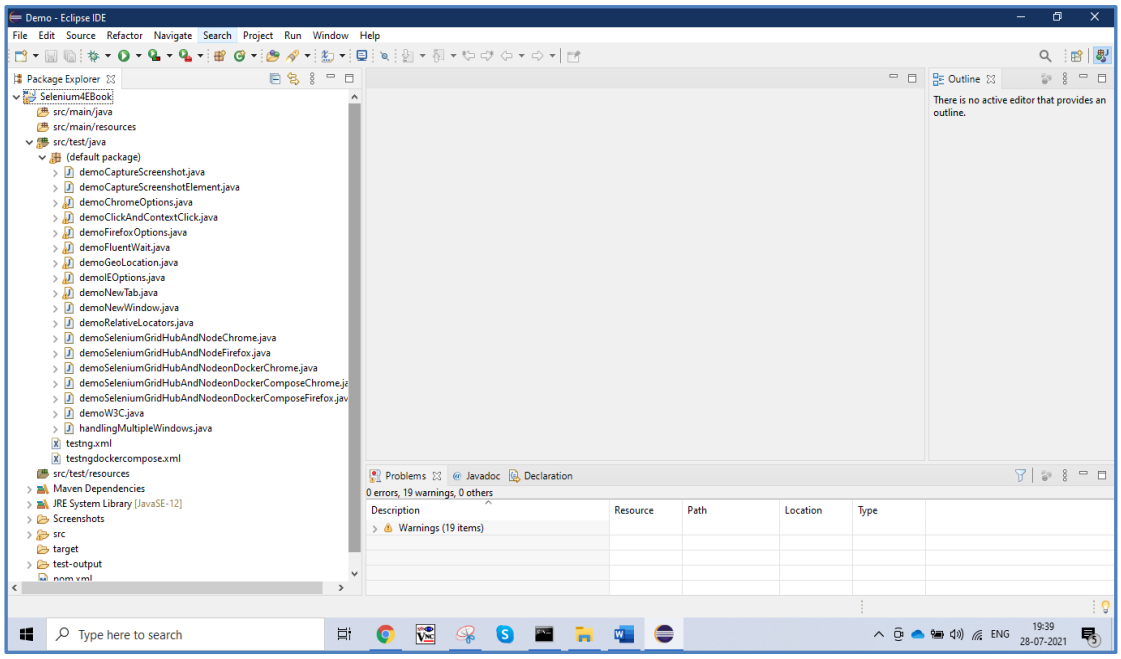
b. Our next step is to import the project, for this we will go to the File menu in eclipse and select import. And then select Project from Folder or Archive



c. Provide the location from your system where the project folder is, as per the image shown below



d. And then click on finish, you will see the project in your eclipse.



e. You can now explore the code files as per the contents in this eBook.

Please note that all the references used in this book are mentioned where the text from them is referred to. And due consideration is given to not miss any source. Still if you find something missing or any errata, please do reach us at below mentioned email address.

atasupport@agiletestingalliance.org

Chapter 2: Relative Locators

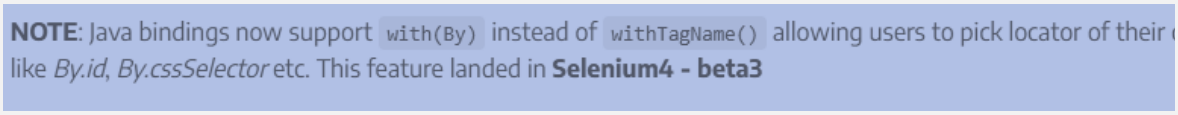
There are many areas which have been impacted/improved in Selenium 4. Locating an element has always been one of the most interesting areas of Selenium and identifying the correct locator which will allow Selenium to identify the element and work on it, a time taking task. With Selenium 4, a new class has been added with the intention to make Locators more friendly to end users. It is known as **RelativeLocators**.

The java documentation of the class is available here

https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/locators/RelativeLocator.html

Note: The javadoc for this class as of writing this book reflects alpha version and not Beta 3 version.

`withTagName(java.lang.String tagName)` is not supported in the latest Beta version of Selenium, rather `with(By)` is the method supported.



[image taken from Selenium Documentation webPage – 16 Aug 2021 - https://www.selenium.dev/documentation/webdriver/locating_elements/]

The concept which is introduced in Selenium is not new in the field of test automation. It has been introduced early in the product Sahi, which is one of the test automation solution available.

With the intent of introducing Relative Locators, the end users are now enabled to identify the object with respect to its location to another element on the web page The ease with which the elements can be identified has been significantly improved by this one new change. We would be describing this in detail below.

2.1 Web Elements and Locators

A html page is made up of various html elements. Each html element has a start tag, an end tag. A html element also has one or more attributes which are used to

describe the properties of the html element. The attributes depend on the type of html element being used.

A generic syntax of the html element look as follows -

<html start tag attribute1 = value1 attribute2 = value2 **>** Display Text **</html end tag>**

We will be using the following web application for our explanation. https://seleniumsummit21.agiletestingalliance.org/.

Let us look at our web application html elements, which we see on the screen and how do they appear at the back end. If we look at the top of the web page (https://seleniumsummit21.agiletestingalliance.org/), we will find different menu items as in image below

If we use the dev tools inspector of chrome, we will find out that these are all anchor tag elements at the backend, where each element has a html start tag, an end tag, and a list of attributes.

Let us look at some of the menu items html syntax

Menu	HTML Syntax
Home	**<a** href="https://seleniumsummit21.agiletestingalliance.org/" aria-current="page">Home****
Speakers	**<a** href="https://seleniumsummit21.agiletestingalliance.org/speakers">Speakers****
Videos	**<a** target="_blank" rel="noopener" href="https://cpsat.agiletestingalliance.org/seleniumsummit21/">Videos****

When we must work with Selenium to automate the browser actions, we need to provide information with respect to selenium to help it identify the web element on the page. Once Selenium can identify the object, it will then perform the action as mentioned in the script on it.

2.2 Selenium 3.X locator types

In the selenium version 3.X series, the following were the different locator types available to us

Locator	Explanation
ID	If the web element html has an ID attribute, we can use it to locate an element
NAME	If the web element html has a NAME attribute, we can use it to locate an element.
XPATH	Xpath could be absolute or relative, and it is the path traversed to find an element on the web page.
CSS	We can use the CSS web technology to identify the html element on the web page
LINKTEXT	It is used for anchor tags, and it is the displayed text of the link [anchor] element.
PARTIALLINKTEXT	We can use a partial string from the display string also to identify the web element.
CLASSNAME	Classname is an attribute of the html element, and we can use the value associated with it to identify the element
TAGNAME	The html tag name of the element can be used to identify the element.

A locator provides information to help selenium identify the web element on the page, and then perform the action on it. We do see that the above locator information can sometimes get too technical, and there was a need to improve the locator strategies which are more obvious to use. By this we mean, a locator strategy which allows us to identify an element based on its position with respect to another element as we see on the web page.

2.3 New Relative Locators in Selenium 4

The new set of locators which are added in Selenium 4, are called as Relative Locators. They allow us to identify an element based on its location on the web page, with respect to another element on the web page.

As mentioned in earlier section a new class called as **RelativeLocator** is added in Selenium 4. It allows to identify the reference object using any of the By locator

strategies, and then use the **RelativeBy** options on it to identify the object we are wanting.

Before we jump to the code, let us look at the different options available. These methods are defined in the **RelativeBy** class of Selenium, which java documentation is available here https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/support/locators/RelativeLocator.RelativeBy.html

The following are the new set of locator methods which are added in Selenium 4.0

RelativeBy methods available to us	Explanation
above	With respect to an element, the element we need is above it. It can be used in two ways- **above(By)** or **above(WebElement ele)**
below	With respect to an element, the element we need is below it. It can be used in two ways- **below(By)** or **below(WebElement ele)**
near	With respect to an element, the element we need is near it. By near we mean utmost 50 pixels away. Or you can the other variants which are available of the near method. We can use the near method as – **near(By loc, int atMostDistanceInPixels)**, **near(WebElement ele, int atMostDistanceInPixels)**, or **near(WebElement element)**, where the utmost distance is 50 pixels.
toLeftOf	With respect to an element, the element we need is at left direction. It can be used in two ways **toLeftOf(WebElement ele)** or **toLeftOf(By locator)**.
toRightOf	With respect to an element, the element we need is at right direction. It can be used in two ways **toRightOf(WebElement ele)** or **toRightOf(By locator)**.

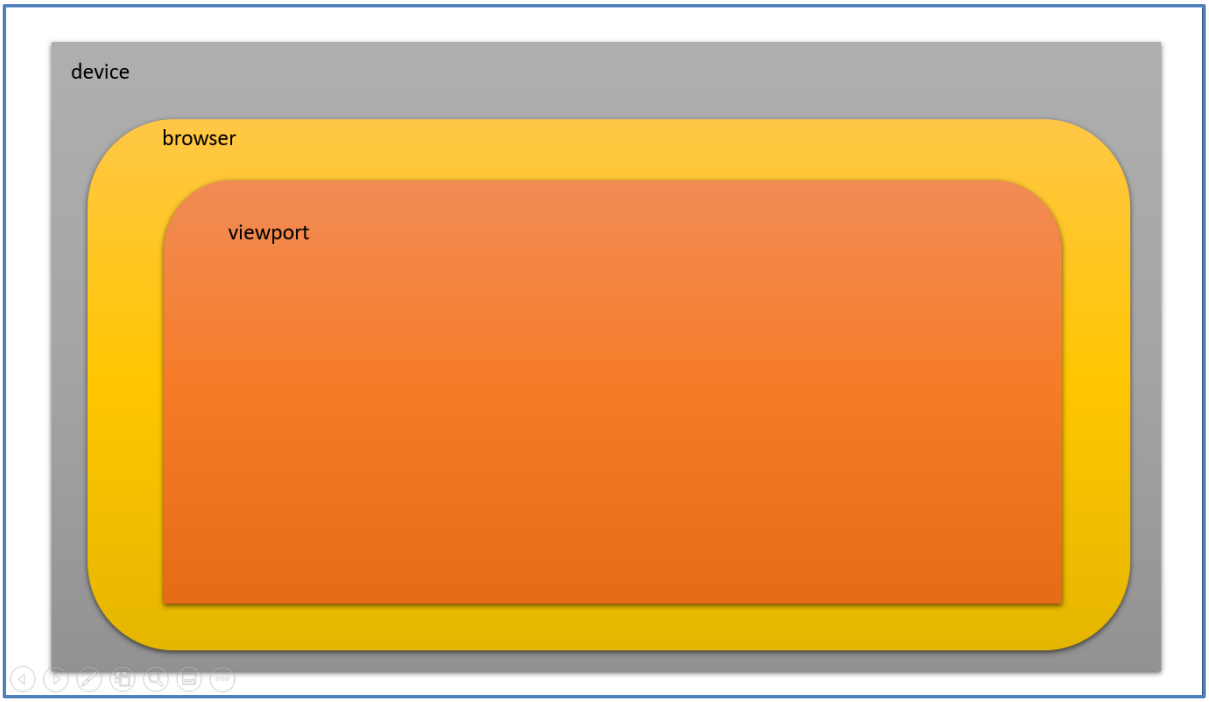
2.4 What are relative locators based on?

To understand the working of these Relative Locators, we will need to understand a javascript function **getBoundingClientRec()**. More details on it are available on the below URL.

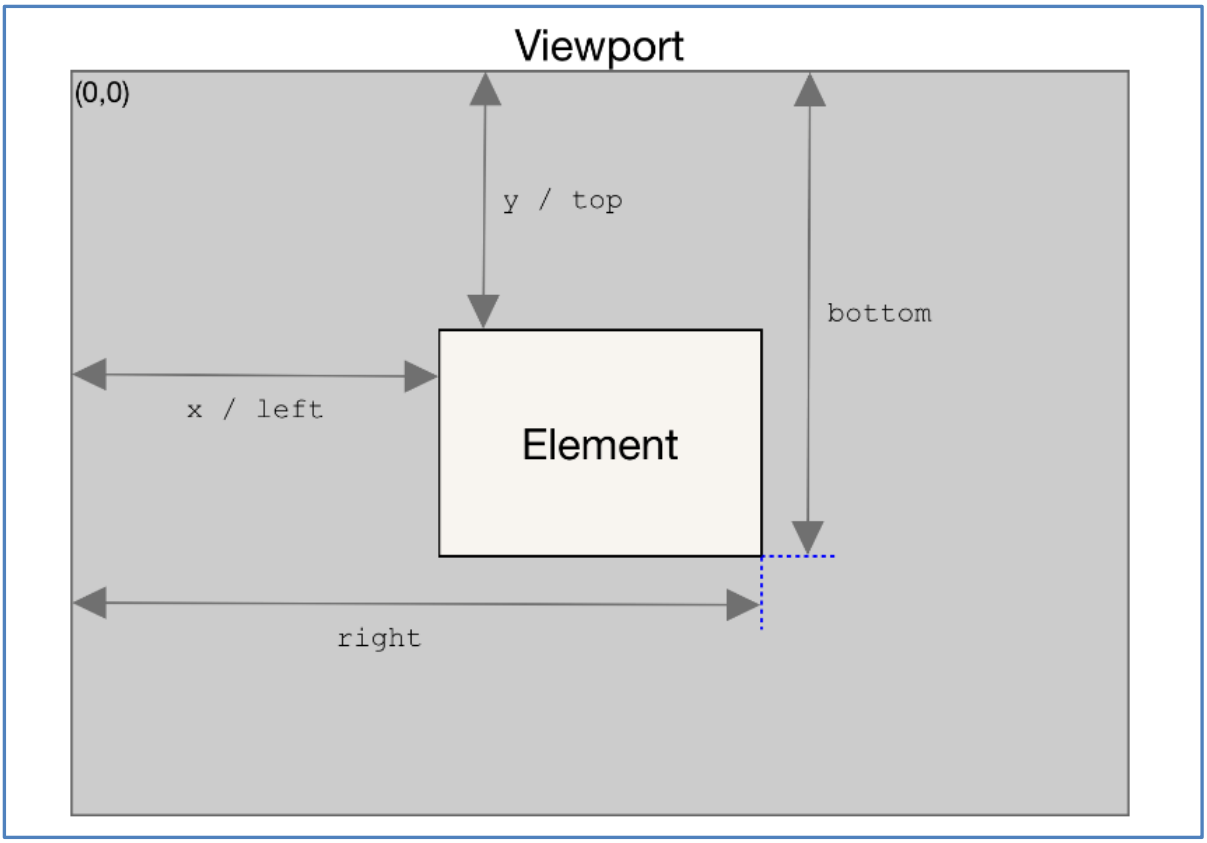
https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect

To understand the above method, we will have to understand a few jargons.

ViewPort - It refers to the part of the document, which is currently visible in the window, which is being viewed. The following image will help us visualize it -



DomRect- It represents the rectangular region an element occupies in a viewport. The method **Element.getBindingClientRect()** returns to us the **DomRect** for that element. The following image will help us visualize it -



We can also state that **DomRect** is the smallest rectangle returned which contains the entire element.

The **top**, **left**, **right** and **bottom** (as per the image above) are the properties associated with the **DomRect** object helps us to identify the web elements with help of Relative Locators. This forms the basis of the relative locators in Selenium 4.

2.5 Practical implementation of Relative locators

To practically implement the relative locators, we will be using the following webpage https://the-internet.herokuapp.com/login]

We see in here three elements - username, password, and Login.

Now let us take the Relative Locators one by one.

above()

This returns to us a web element which appears above the specified element. Let us try to identify Password text box, which is above the Login button. So, we first identify the Login button using the xpath, and in reference to the login button, we are trying to locate an element which is above the login button and has a tagname input. In here we must note that there are two input fields above the login button, but the closest will be returned, which will be the password field.

```
driver.findElement(RelativeLocator.with(By.tagName("input")).above(loginBtn));
```

Entire code is as below

```
 //above
        WebElement
loginBtn=driver.findElement(By.xpath("//button[@class='radius']"));
        WebElement
password=driver.findElement(RelativeLocator.with(By.tagName("input"
)).above(loginBtn));
        password.sendKeys("SuperSecretPassword!");
```

Please note we have used the method **With(By)** of the RelativeLocator class.

This method returns an object of the RelativeBy, on which we can use any of the above mentioned new Relative locator techniques.

The By as argument allows us to identify the referenced object using any of the By locator strategies which we have been using since Selenium 3 onwards.

We need to understand in here that the Relative Locators are used in reference to the location of another object on the web page, which we have identified using the By locator techniques.

This concept is generally useful in identifying those objects whose own locators are difficult to design, or our script will be more reliable if we use Relative Locator strategy by using a referenced object which can be located easily using By locator strategies.

Another point to note in here, is that for the RelativeLocator, we have taken example of the methods using WebElement as argument. We should be aware that there are more variants available of these methods

RelativeLocator.RelativeBy **above**(**By** locator)
RelativeLocator.RelativeBy **above**(**WebElement** element)

below()

This returns to us a web element which appears below the specified element as we see on the viewport. For this, we will take reference element as the username field and with respect to its location find the element password which appears below it. The following code does that.

Please note for both the examples above and below we are trying to identify the Password field only. In this code example we have taken username field as the referenced object, below which we are trying to identify another object which has the tagName value as input.

Thus, Password field will be identified, as it is the closest input html object to the username field.

```
passwd=driver.findElement(RelativeLocator.with(By.tagName("input")).
below(username));
```

The complete code is shown below

```
 //below
        WebElement username=driver.findElement(By.name("username"));
        WebElement
passwd=driver.findElement(RelativeLocator.with(By.tagName("input")).
below(username));
        passwd.sendKeys("SuperSecretPassword!"); //set the text in
password field
        passwd.clear(); //clear the contents of the password field
```

Please note that the variants available for below() method are

RelativeLocator.RelativeBy **below**(**By** locator)
RelativeLocator.RelativeBy **below**(**WebElement** element)

near()

This method helps us find an element, which is at most 50px away from the specified element in any direction. For this we will take as reference the password web element and locate the username element. We identify the password element using the name By locator strategy and use the RelativeBy method near(WebElement) to find the username field.

```
//near
        WebElement pwd=driver.findElement(By.name("password"));
        WebElement
uname=driver.findElement(RelativeLocator.with(By.tagName("input")).
near(pwd));
        uname.sendKeys("tomsmith"); //set the text in username
field
```

near() method has 4 variants. Please see below

RelativeLocator.RelativeBy	**near**(**By** locator)
RelativeLocator.RelativeBy	**near**(**By** locator, int atMostDistanceInPixels)
RelativeLocator.RelativeBy	**near**(**WebElement** element)
RelativeLocator.RelativeBy	**near**(**WebElement** element, int atMostDistanceInPixels)

toLeftOf()

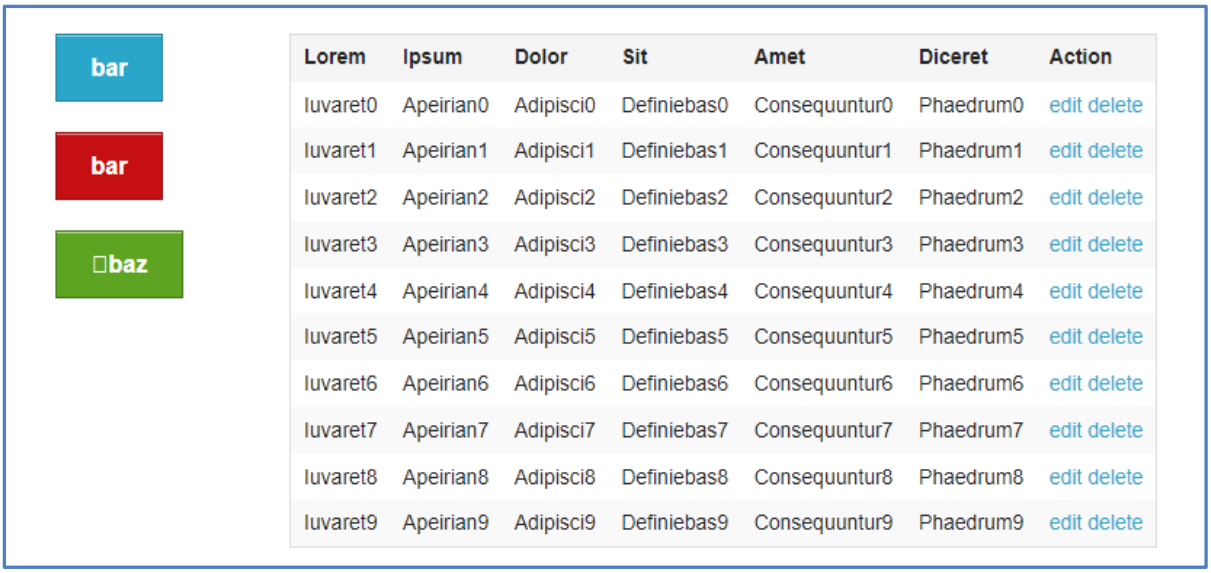
The next we look is at the toLeftOf() Relative Locator.

For this and for the next toRightOf() relative locator, we will take help of the following web page:

https://the-internet.herokuapp.com/challenging_dom.

Please note that the text displayed on these buttons will change as the page is refreshed.

And as we see on the screen, the following image is seen



We see a table and on its left, there are three elements, and similarly on the right of either of the three elements is a web table.

Let us see how we use Relative locator to help us in here.

Let us try and identify the last button which displays the text **baz**. As mentioned, the text of the buttons changes with each page refresh.

So, we take the web table as the referenced object, and try to find the first element which appears to the left of the web table.

For this we first identify the web table, and then use the **toLeftOf(WebElement ele)** method to fetch the element, and print the text displayed on it.

The code for this is below

```
//toLeft
	WebElement table=driver.findElement(By.tagName("table"));
	WebElement
lastButton=driver.findElement(RelativeLocator.with(By.tagName("a"))
.toLeftOf(table));
	System.out.println(lastButton.getText());
```

Additional variants for this method are as below

RelativeLocator.RelativeBy	toLeftOf(By locator)
RelativeLocator.RelativeBy	toLeftOf(WebElement element)

toRightOf()

In here, we take as reference one of the displayed buttons which we identify using the xpath. Once the button is identified we used it as a reference to find the web table which appears to the right of it, on the web page, and print the text of the web table. We use the method variant **toRightOf(WebElement ele)**

The code for it is below

```
//toRight
	WebElement
refElement=driver.findElement(By.xpath("//*/a[3]"));
	WebElement
displayedTable=driver.findElement(RelativeLocator.with(By.tagName("
table")).toRightOf(refElement));
	System.out.println(displayedTable.getText());
```

The variants for this method are

RelativeLocator.RelativeBy	**toRightOf**(**By** locator)
RelativeLocator.RelativeBy	**toRightOf**(**WebElement** element)

In this section we have seen the usage of the Relative Locators. What we need to keep in mind here is that the earlier locator strategies worked based on the html information associated with the web element, whereas the **relative locators work on how we view the element on the web browser**, we identify the object we want in reference to another object which appears on the web page. We can also use more than one relative locator to identify an element on the given page, depending upon the scenario, and web page information we have available to use.

Chapter 3: Windows and Tab Management

In Selenium 3.x, every time we use the selenium get method or the navigate method, it opens a new window, with the url of the application. If we wanted to open a new application url in a new window we would need a new WebDriver object to be created.

With Selenium 4 series we can now launch new windows of different application urls using the same driver object.

For this we use the **switchTo.newWindow()** command. The command **switchTo.newWindow()** takes as argument the **WindowType**. The WindowType can be a **WINDOW**, which open a new Window, or could be a **TAB**, which will then open a new TAB with this existing window.

Let us see this behavior with the following code samples.

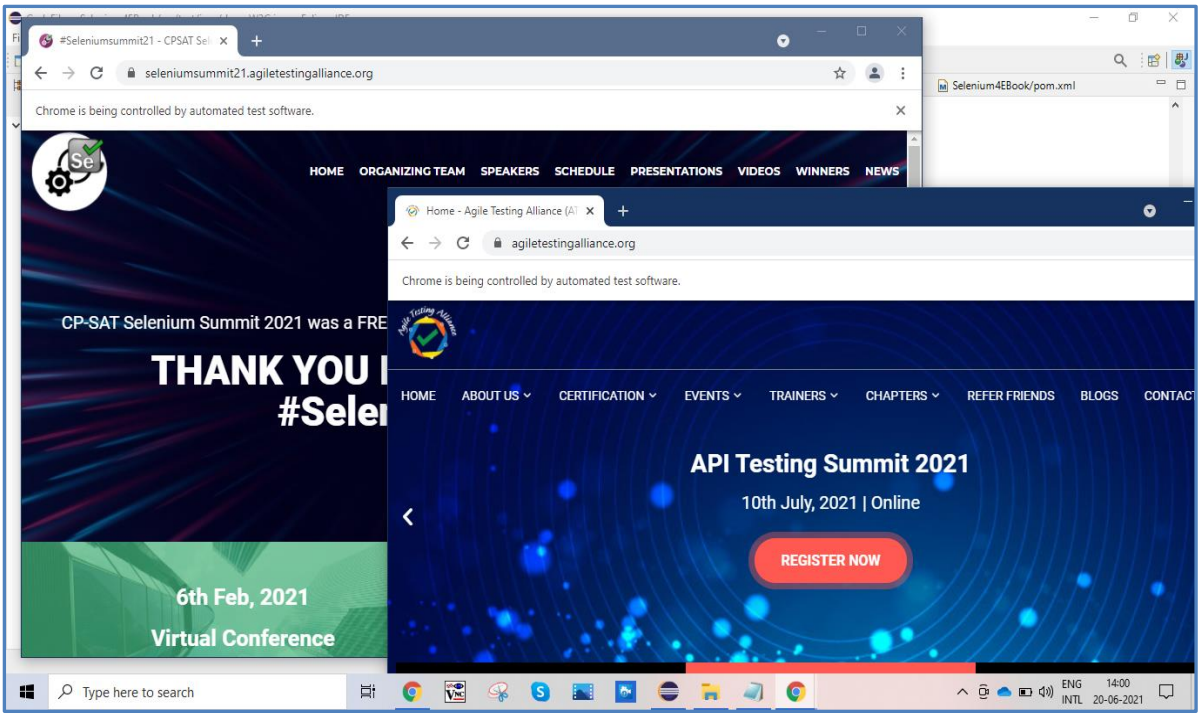
In the first code sample, we will pass the url "https://seleniumsummit21.agiletestingalliance.org/" to the driver, get command. This will open the application on a browser window.

And then we will use the driver.switchTo().newWindow() command and launch a new Window. On this we will use the driver.navigate().To() command to open another application, the url for which is -https://www.agiletestingalliance.org/

The code sample is

```
WebDriverManager.chromedriver().setup();
driver = new ChromeDriver();
driver.get("https://seleniumsummit21.agiletestingalliance.org/");
driver.switchTo().newWindow(WindowType.WINDOW);
driver.navigate().to("https://www.agiletestingalliance.org/")
```

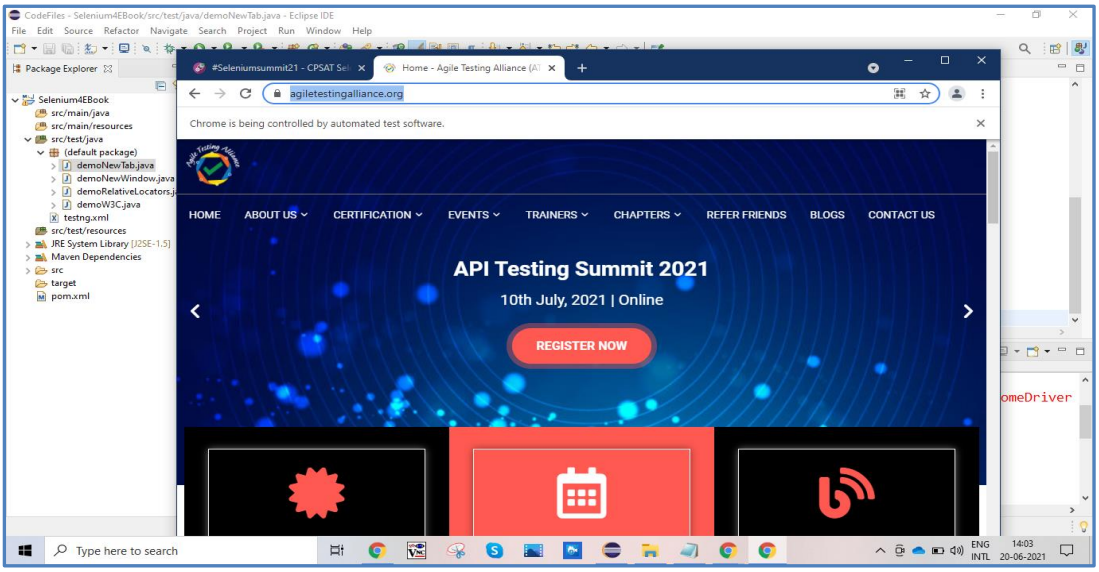
When the above code runs, we will see two different applications being launched on two different browser windows



Let us try the same code, but this time, we will open the other application in a TAB. The code will be as follows

```
driver.get("https://seleniumsummit21.agiletestingalliance.org/");
driver.switchTo().newWindow(WindowType.TAB);
driver.navigate().to("https://www.agiletestingalliance.org/");
```

The image is what we see below as the code executes-



The point to note in here is that the driver, will always refer to the latest window which has been launched. So, if we want to go and work with first application, we will have to switchTo that window.

For this we will use window Ids. The concept to switch between windows is the same as it was in the previous version of Selenium.

Whenever a new window is created with the driver session, it is assignment a window Id. This is known as a window handle. We can use the method getWindowHandles() to fetch a set of all window handles created with the driver object. The first handle will be of the first window launched, the second will be of the second window launched and so on. To see its usage, we will be doing the following steps

a. Launch application 1, using url https://seleniumsummit21.agiletestingalliance.org/
b. Launch application 2, using url - https://www.agiletestingalliance.org/ on a new window
c. Fetch all the window handles and store them in a collection
d. Iterate the collection, print the title of the web page in each window and close it.

The complete code looks as follows

```
import org.TestNG.annotations.*;
import java.util.Iterator;
import java.util.Set;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import io.GitHub.bonigarcia.wdm.WebDriverManager;

public class handlingMultipleWindows {

     WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
       WebDriverManager.chromedriver().setup();
       driver = new ChromeDriver();
  }

  @Test
  public void openApp() throws Exception{
     driver.get("https://seleniumsummit21.agiletestingalliance.org
```

```
/");
driver.switchTo().newWindow(WindowType.WINDOW);

driver.navigate().to("https://www.agiletestingalliance.org/");
Set<String> windows=driver.getWindowHandles();
        Iterator<String> itr = windows.iterator();
      // traversing over windows Set
       while(itr.hasNext()){
             driver.switchTo().window(itr.next());
             System.out.println(driver.getTitle());
             driver.close();
      Thread.sleep(1000); //introducing a small wait of 1 second.
           }
       }
    @AfterMethod
  public void afterMethod() {
        System.out.println("Execution Complete...");
  }

}
```

The output of the execution looks as follows

```
INFO: Found exact CDP implementation for version 91
#Seleniumsummit21 - CPSAT Selenium Summit 2021
Home - Agile Testing Alliance (ATA)
Execution Complete...
PASSED: openApp
```

Thus, in this section we learnt how we can launch different applications/URLs using the same driver object either in a new Window or a new Tab using Selenium 4 methods. We can switch our focus between the applications and work with the web pages within them like we used to in Selenium 3.X.

Chapter 4: Modified Action Class

The action class in Selenium is used to perform complex gestures for keyboard and mouse actions. It is available in the interactions package with Selenium. We can either perform a single action or can combine more than one action to create a composite action and then use a method called as build() and then perform() to make it work.

The Action class in Selenium 4, comes with some new methods, and this replaces the earlier Action class available in the interactions package.

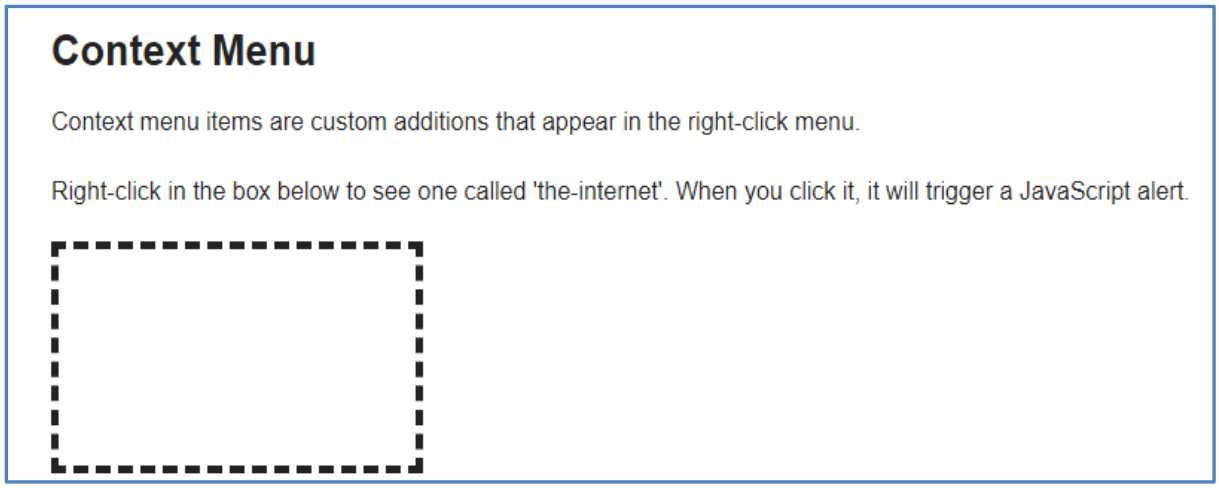
The methods are as follows:

Method	Description
click(WebElement)	It performs click operation on the WebElement given. This method should be used instead of element.moveToElement().click()
doubleClick(WebElement)	It performs double click on the web element passed as argument. This method should be used instead of element.moveToElement().doubleClick()
clickAndHold(WebElement)	It performs click on the web element passed as argument and holds it. This method should be used instead of element.moveToElement().clickAndHold()
contextClick(WebElement)	It performs right click on the element which is passed as argument to this method. It should be used instead of element.moveToElement().contextClick()
release()	This method was earlier implemented in ButtonReleaseAction class method, from the interactions package, it is now moved to the Action class itself.

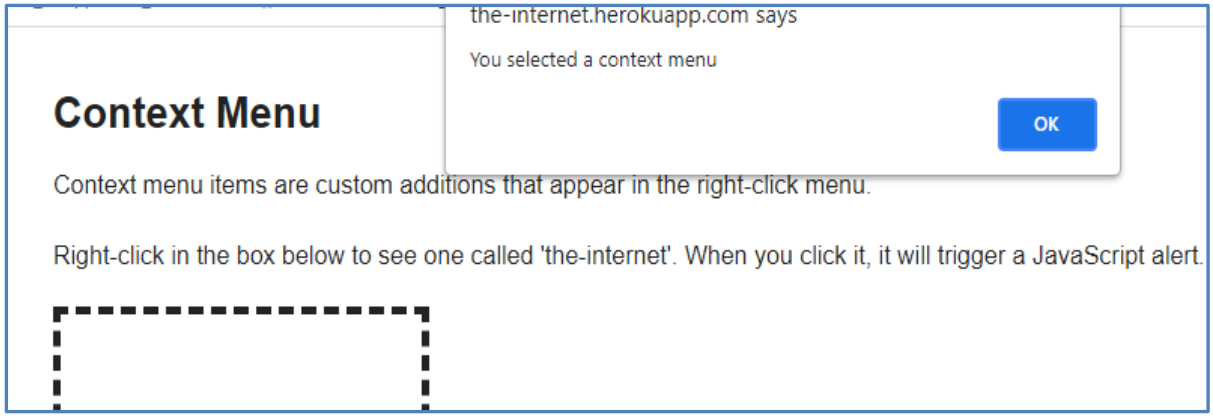
To show usage of these methods, let us take some examples.

click(WebElement) and contextClick(WebElement)

To see the function of above methods, we will go to the application https://the-internet.herokuapp.com/ and from here we will choose the option Context Menu. For this we will use the **click(WebElement)** action. On the Context Menu page which looks as follows -



We will perform **contextClick(WebElement)** in the box, which causes a popup to appear, and then the page looks as follows -



Let us see the code sample for this below.

```
import org.TestNG.annotations.*;
import java.util.Iterator;
import java.util.Set;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import io.GitHub.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.interactions.*;

public class demoClickAndContextClick {

     WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
```

```
        //create webdriver for chrome driver
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
  }

  @Test
  public void openApp() throws Exception{
        //open application
        driver.get("https://the-internet.herokuapp.com/");
        WebElement menuLink=driver.findElement(By.linkText("Context
Menu"));
        //Create object for action class
        Actions act= new Actions(driver);
        //call the click(WebElement) method
        act.click(menuLink).perform();
        WebElement box=driver.findElement(By.id("hot-spot"));
        //call the contextClick(WebElement) method
        act.contextClick(box).perform();
        //wait for 2 seconds before browser closes to see the
action done
        Thread.sleep(2000);
  }

  @AfterMethod
  public void afterMethod() {
        //close the browser
       driver.quit();
  }

}
```

In the same manner we can use the doubleClick(WebElement), clickAndHold(WebElement) as and where required while trying to automate the script.

This new implementation removes the unnecessary need to move to the element and then click. This is not a big change but a welcome change in Selenium 4

Chapter 5: Enhanced Selenium Grid

This chapter has following sections

1. Introduction to selenium grid components
2. Setting up of Selenium Grid
3. How to run a standalone Selenium Grid
4. How to run selenium grid using docker running on a linux virtual machine on a GCP (Google Cloud platform)

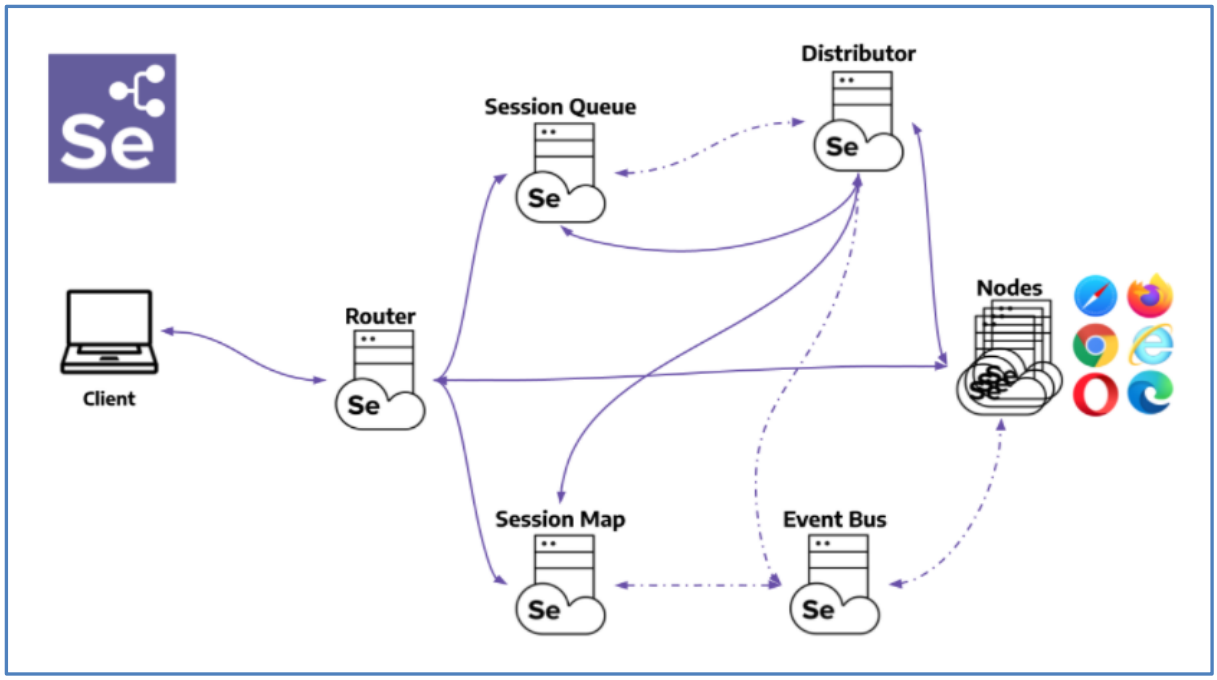
5.1 Introduction to selenium grid components

Selenium Grid is an important component of the Selenium project. It allows us to run the scripts created on a remote machine. It does so by routing the commands sent by the client to the remote browser instance. Selenium grid also allows us to run our scripts in parallel on different machine browser configurations, which can also be centrally managed, instead on the script level. According to the information available on the Selenium documentation website, the Selenium Grid 3 is no longer supported, and we are advised to use Selenium Grid 4. More is available here - https://www.selenium.dev/documentation/en/grid/

In the earlier versions of Selenium grid, there were generally two major components, a hub which acted as the central server which received the request for execution, and a node, which effectively was a browser machine instance on which the script execution would take place.

But in the new Selenium Grid 4 the components have changed, and some new ones are added.

Let us have a look at the following image



The above image is taken directly from the Selenium documentation website[https://www.selenium.dev/documentation/grid/components_of_a_grid/].

We see in the above image a lot of jargon has been introduced. Let us try and understand where each component belongs and what it does. Traditionally the Grid consisted of the hub and the node.

From Grid 4, the hub now is made of the **Router**, the **Distributor**, **Session Map**, **Session Queue,** and the **Event Bus (all these components together make the erstwhile hub – even now the command to start these components is using the hub keyword)**. And then we have the **Node**. Let us look these components one by one.

Client - Client is the component (which is same as in earlier Selenium 3.x) which sends the request for execution to the hub, the hub now contains many components, so the component which basically receives the request is the Router.

Router- It acts as the main entry point to the Grid and receives all requests. If the request received is for a new session it will be handled by the Session Queue. If the request is for an existing session, it will be sent to the Session Map. The Session Map will send the information of the Node back where the session is running. The Router then sends the request to that Node.

Distributor - This component carries the information about all nodes, and their capabilities. Once a request for the new session is made, it finds out the node where the session can be executed. It then maps the session id, with the node information and stores it in the Session Map

Session Map- The Session Map is the place where the information of the session id and the node is stored. The information is sent to the Session Map by the Distributor. It is then used by the Router to send the request to the node, by matching the session id, if it already exists.

Event Bus- All the communication among the nodes, distributor, Session Queue and Session Map is handled by the Event Bus.

Session Queue- The session queue receives the requests for all new sessions from the Router. All the received requests are added in the Queue. Once the response for a request is received, it is successfully added to the queue. If the timeout occurs it is discarded. Once the request is accepted, the Event Bus triggers an event, which is picked by the Distributor. The Distributor tries to create a session based on the node information. If the node with the requested capabilities is not found, the request is rejected, and information is sent to the client. Once the session is created the information is sent to the Session Map by the Event Bus.

Node- A node is a machine browser combination, where actual execution of the commands takes place. An example of node in a grid could be a machine with windows OS, running edge browser, or a mac OS machine running safari browser.

5.2 Setting up of Selenium Grid

The Selenium Grid 4 version provides us with four different ways to set it up for the execution.

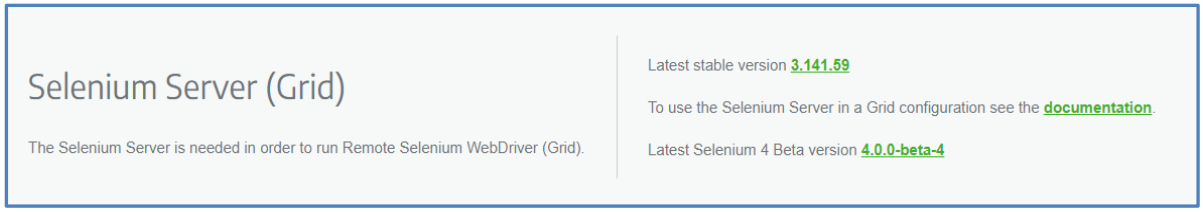
a. Standalone
b. Hub and Node
c. Distributed
d. Docker

In this book we will be showing the **Hub and Node mode**, and the **Docker mode** to setup the Selenium Grid.

For the remaining two setups, you can refer to the link available here - https://www.selenium.dev/documentation/en/grid/grid_4/setting_up_your_own_grid/

5.2.1 Hub and Node Mode to setup selenium grid

Before we proceed to see selenium grid in action in different setup version, let us first download the Selenium Grid jar file from this link - https://www.selenium.dev/downloads/
Scroll down to the Selenium Grid section, and while writing this eBook, the latest version of Grid which is available is - **4.0.0-beta-4**, and we will be showing demo examples using this version. Please note that it is possible that the version will change in coming days and a few things can change with it as Selenium 4 is currently in beta state.



Download the jar file and keep it some folder location in your system which is accessible.

The next step is to start the hub, and then the node. Let us see it -

Start the Hub

```
>java -jar selenium-server-4.0.0-beta-4.jar hub
```

The next step is to open the url, to check if the hub is up and running. For this we will use the URL - http://192.168.99.1:4444 , (the IP address will be relevant to the machine on which you have started the hub, if it is your local machine, you can replace the IP address by **localhost** or **127.0.0.1**). By default, the port which is used by hub is 4444. When we open the url on the browser, the page which we see is as follows:

Start the Node

The next step would be to start the node. And before we do that, in the same folder we will be downloading and storing the chromedriver, and the geckodriver. The version used in this section is chromedriver version 91.X and the geckodriver version 0.29.

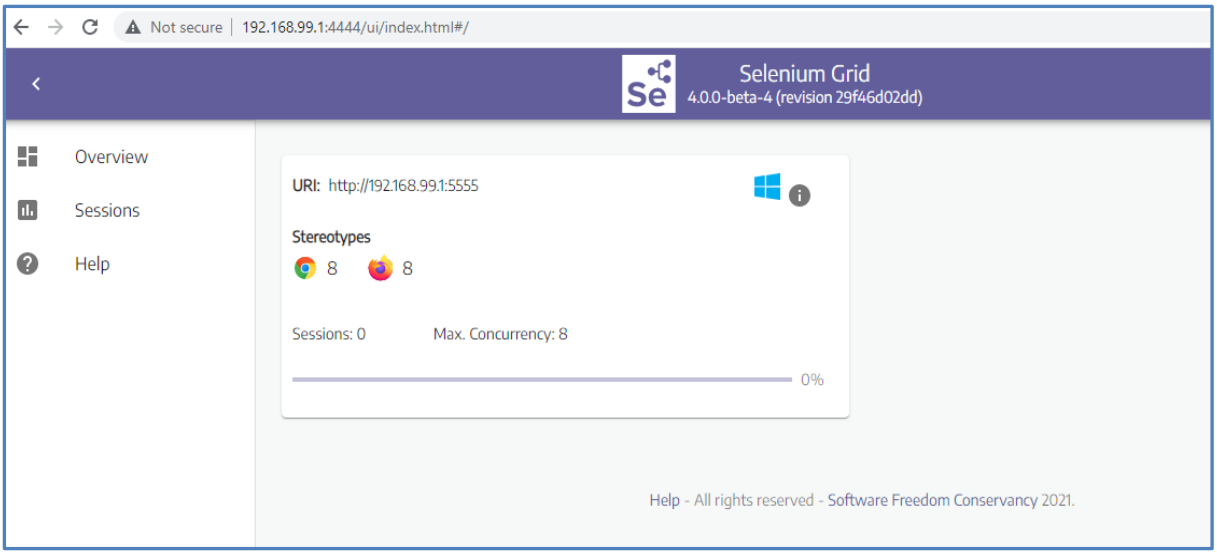
Following command will be used to start the node.

```
java -jar selenium-server-4.0.0-beta-4.jar node --detect-drivers true
```

As we see in the above command, we start the node, and allow it to automatically detect the browser drivers available in the system. Here it will be able to detect the chromedriver and the geckodriver. Please note that the simplified node command automatically detects the driver in the same folder and registers itself with the hub running on the same machine. The –detect-drivers option is new in Selenium 4 which eases the entire setup to a very large extent.

If it is running on a different machine, we would have to use a distributed setup of grid, which is quite different in Selenium 4 due to the various components getting introduced in the latest version of selenium grid. Please refer the following URL for setting up grid in distributed mode.
https://www.selenium.dev/documentation/grid/setting_up_your_own_grid/#distributed-mode

If we connect to the same URL used to connect to the Hub earlier, it will now show that the nodes are connected. Please see the following image



In our next step, we will run our java code. Please note that instead of using WebDriver instance we will be creating instance of the **RemoteWebDriver** and point it to the hub URL. The capabilities which we will pass will be for Chrome browser on the windows machine. So let us see the following script.

In this script we will open the Selenium Summit page using the url - https://seleniumsummit21.agiletestingalliance.org/ and print all the links on the page. We will create two scripts one for chrome and another for firefox and execute it using a TestNG suite file, where we will take class execution in parallel.

Chrome Script

```
import org.TestNG.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.remote.*;
import java.net.URL;
import java.util.List;


public class demoSeleniumGridHubAndNodeChrome {
      WebDriver driver;
      String hubURL = "http://192.168.99.1:4444";

  @BeforeMethod
  public void beforeMethod() throws Exception {
        DesiredCapabilities caps = new DesiredCapabilities();
      caps.setBrowserName("chrome");
      driver = new RemoteWebDriver(new URL(hubURL), caps);
  }
```

```
  @Test
  public void launchApplication() throws Exception{

driver.get("https://seleniumsummit21.agiletestingalliance.org/");
        //find all links on the home page and print information
        List<WebElement> links = driver.findElements(By.xpath("//a"));
        for(WebElement link: links) {
              System.out.println(link.getText());
              System.out.println(link.getAttribute("href"));
        }
  }

  @AfterMethod
  public void afterMethod() {
        driver.close();
  }

}
```

Firefox Script

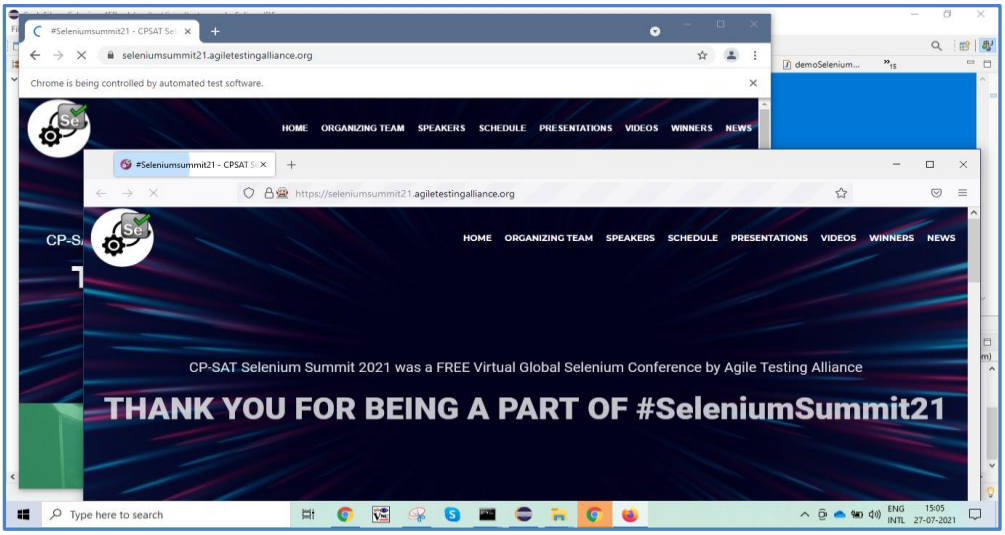
```
import org.TestNG.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.remote.*;
import java.net.URL;
import java.util.List;


public class demoSeleniumGridHubAndNodeFirefox {
      WebDriver driver;
      String hubURL = "http://192.168.99.1:4444";

  @BeforeMethod
  public void beforeMethod() throws Exception {
        DesiredCapabilities caps = new DesiredCapabilities();
      caps.setBrowserName("firefox");
      driver = new RemoteWebDriver(new URL(hubURL), caps);
  }

  @Test
  public void launchApplication() throws Exception{

driver.get("https://seleniumsummit21.agiletestingalliance.org/");
```

```
        //find all links on the home page and print information
        List<WebElement> links = driver.findElements(By.xpath("//a"));
        for(WebElement link: links) {
              System.out.println(link.getText());
              System.out.println(link.getAttribute("href"));
        }
  }

  @AfterMethod
  public void afterMethod() {
        driver.close();
  }

}
```

The TestNG suite file is as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<suite parallel="classes" name="Suite">
  <test name="Test">
    <classes>
      <class name="demoSeleniumGridHubAndNodeChrome"/>
      <class name="demoSeleniumGridHubAndNodeFirefox"/>
    </classes>
  </test> <!-- Test -->
</suite> <!-- Suite -->
```

Upon execution of the TestNG suite file we will find both chrome and firefox browser opening simultaneously, and the same is shown in the following screenshot.



Thus, we have seen how we can start the Grid in the hub and node mode. The next we will see for Docker.

5.2.2 Selenium grid using Docker

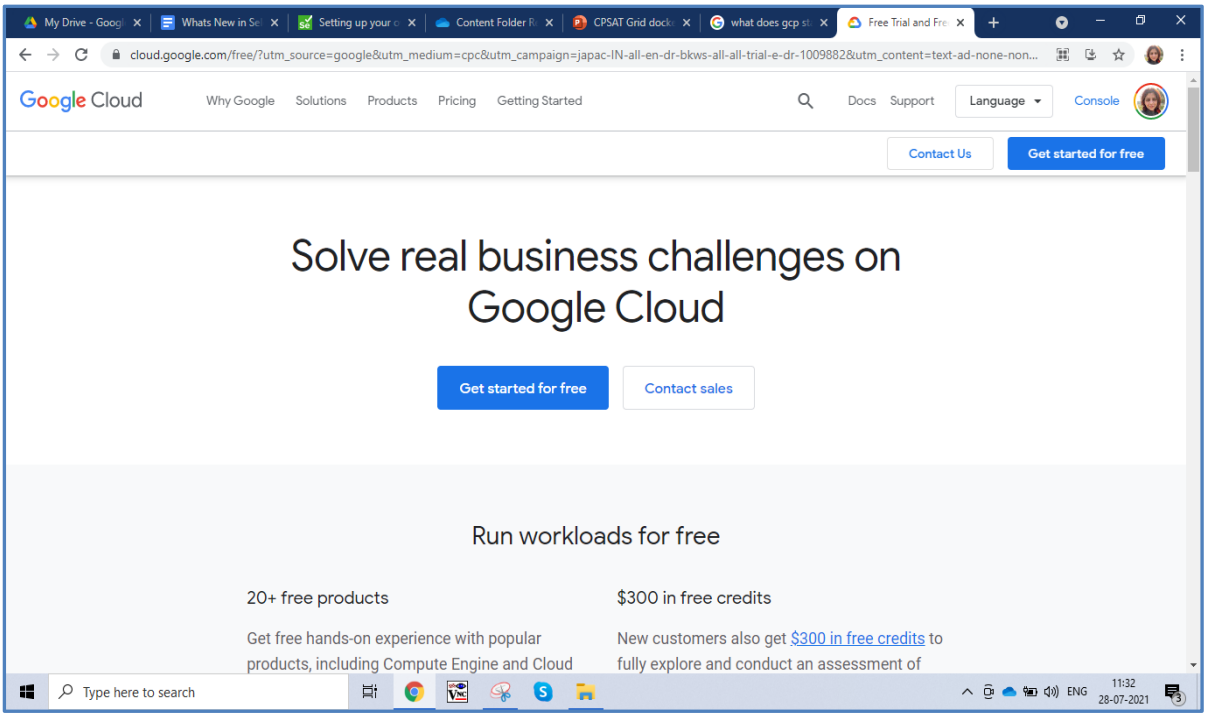
We will be using a Linux machine for installing and setup of docker first. This entire section of setting up of grid using docker has been broken further into following sub sections

a. Creating a Linux virtual machine on Google Cloud platform and setting up docker on that virtual machine
b. Setting up selenium grid in debug mode using docker selenium
c. Setting up selenium grid using docker compose

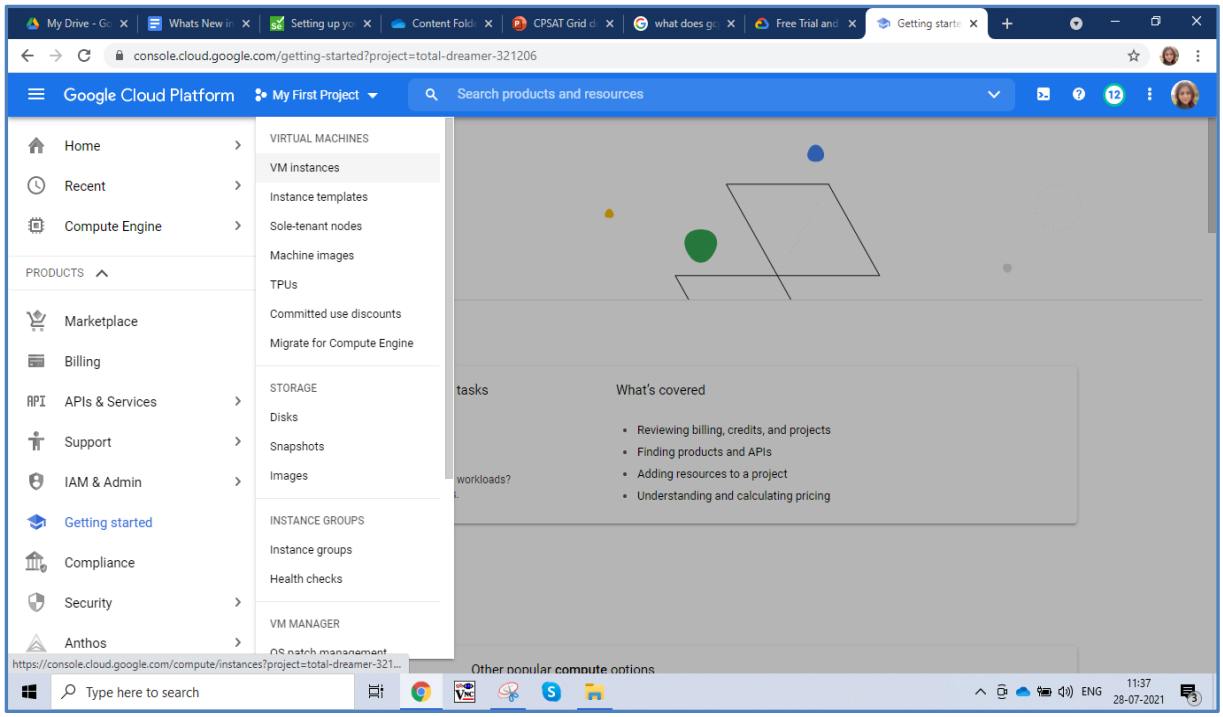
5.2.2.1 Creating a Linux virtual machine on Google Cloud platform and setting up docker on that virtual machine

To understand how we can run Selenium grid using Docker, we will first create a virtual machine using GCP [Google Cloud Platform]. Following are the steps which we will follow-

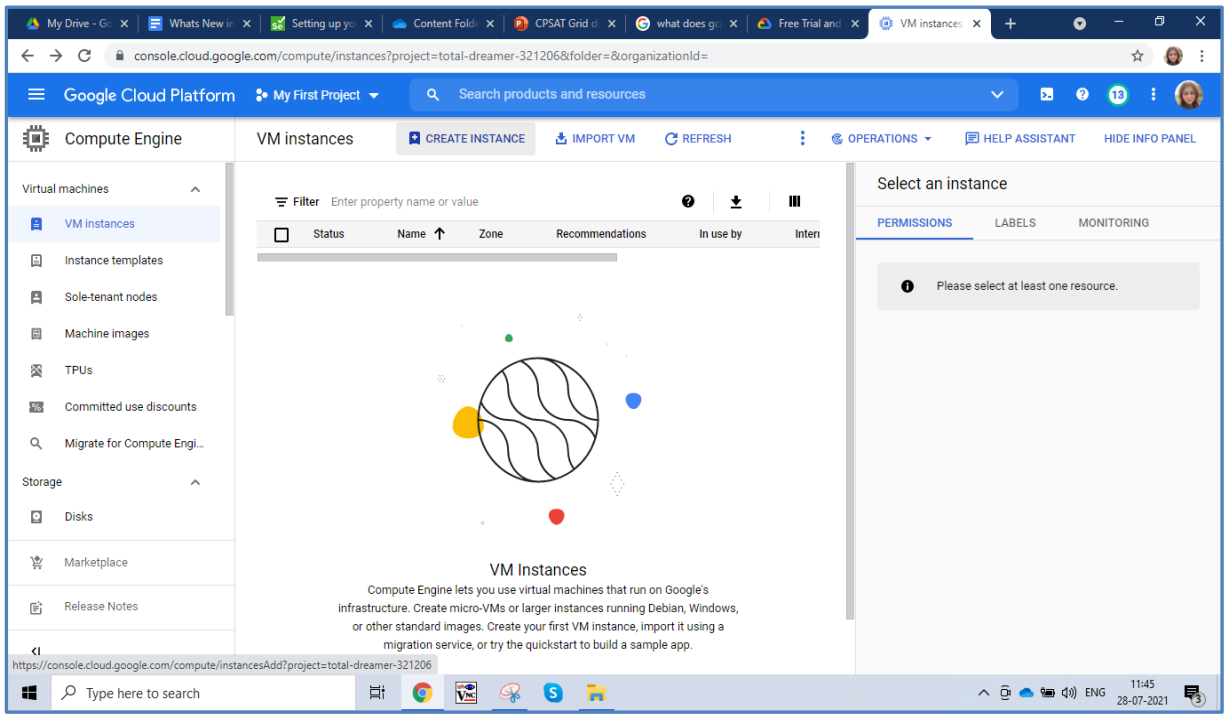
1. If you don't have an account on GCP, you can create one for free using the url - https://cloud.google.com/free



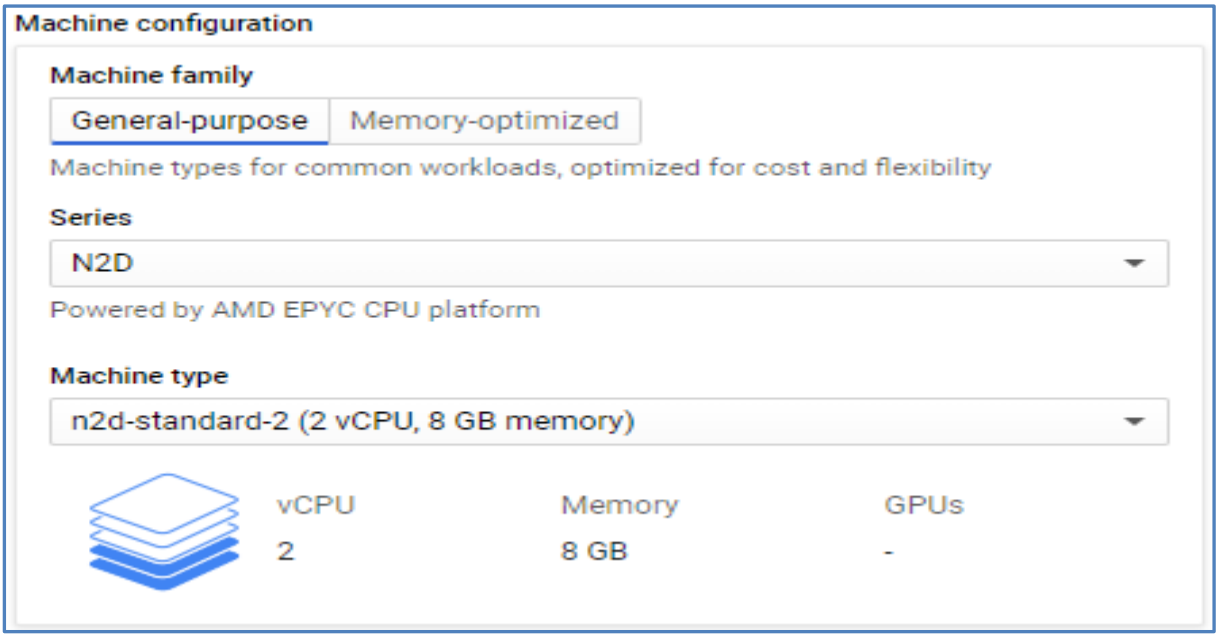
2. Once the account is created you get free credits to try GCP
3. The next step is to create a Virtual Machine Instance, for this we go to the main menu, and select Compute Engine, and from there chose VM Instances, as shows in the figure below -



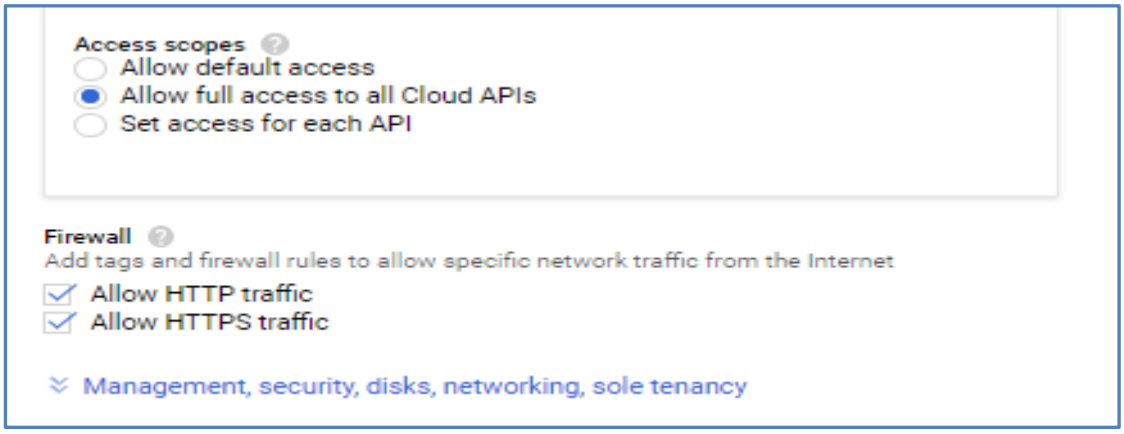
4. The next step is to click on Create Instance, from the Menu bar.



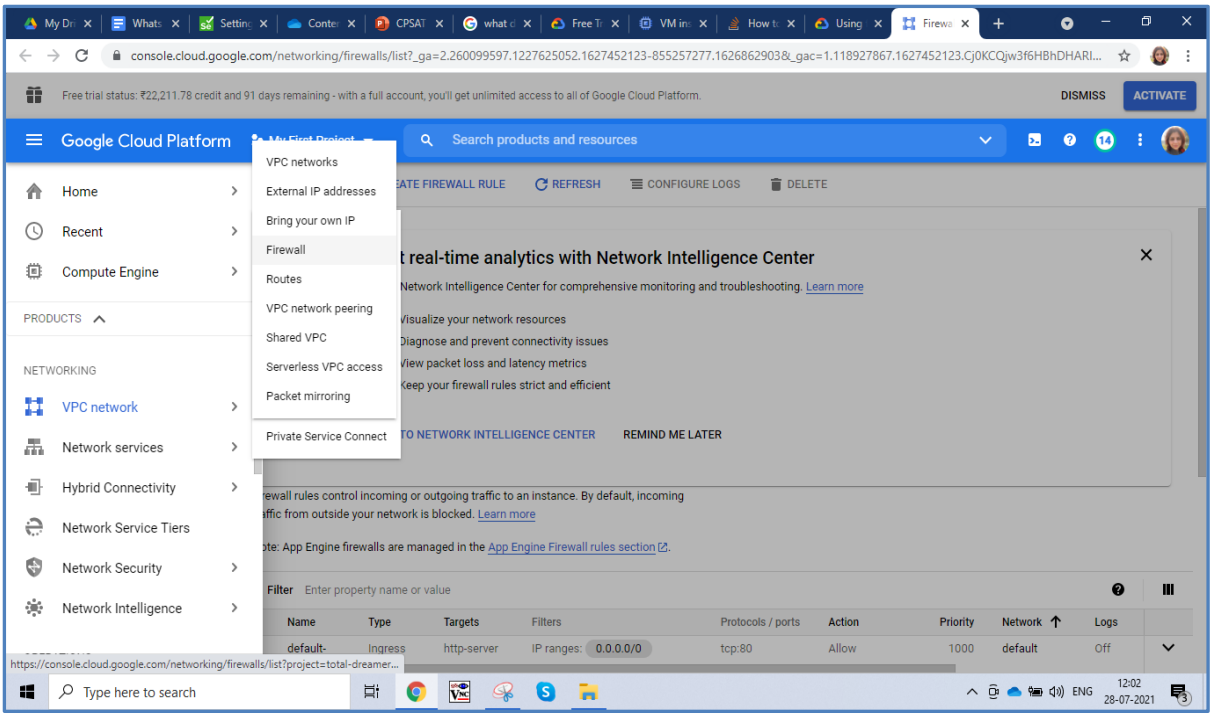
5. The next is to ensure that we select the machine configuration to be of AMD



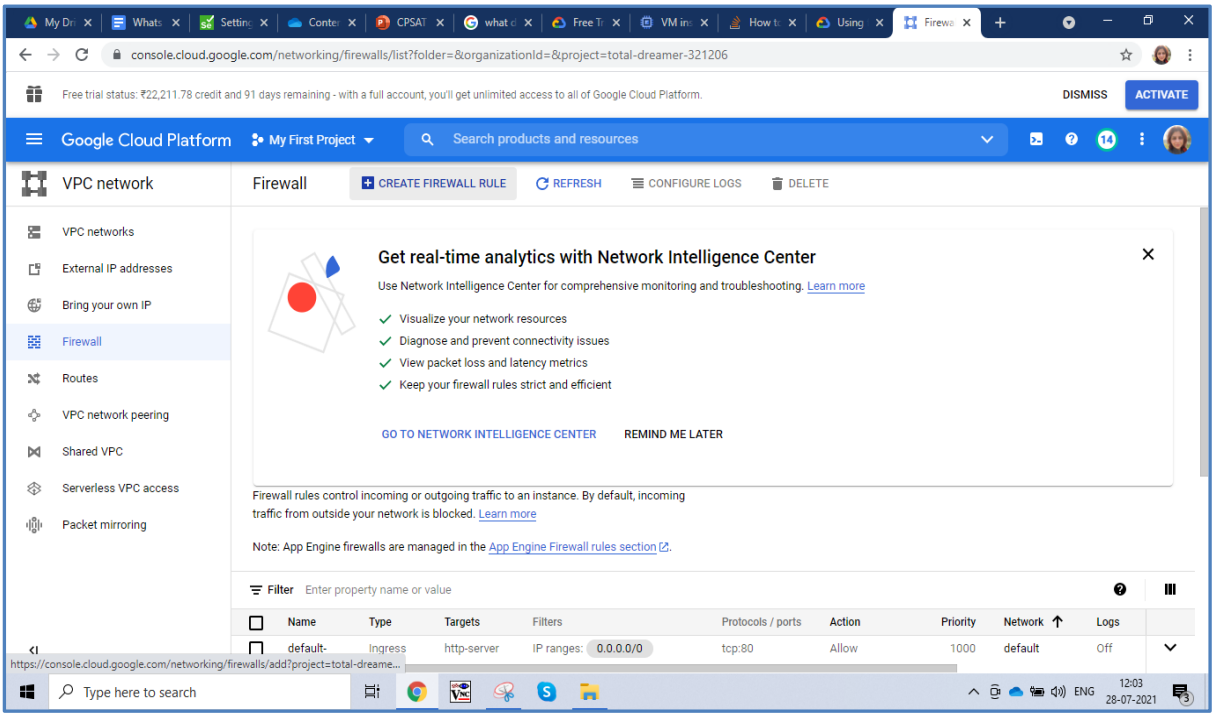
6. Fill in all the asked information. Provide a name to the instance, and ensure to select as per the following image in the questions asked while creating the instance



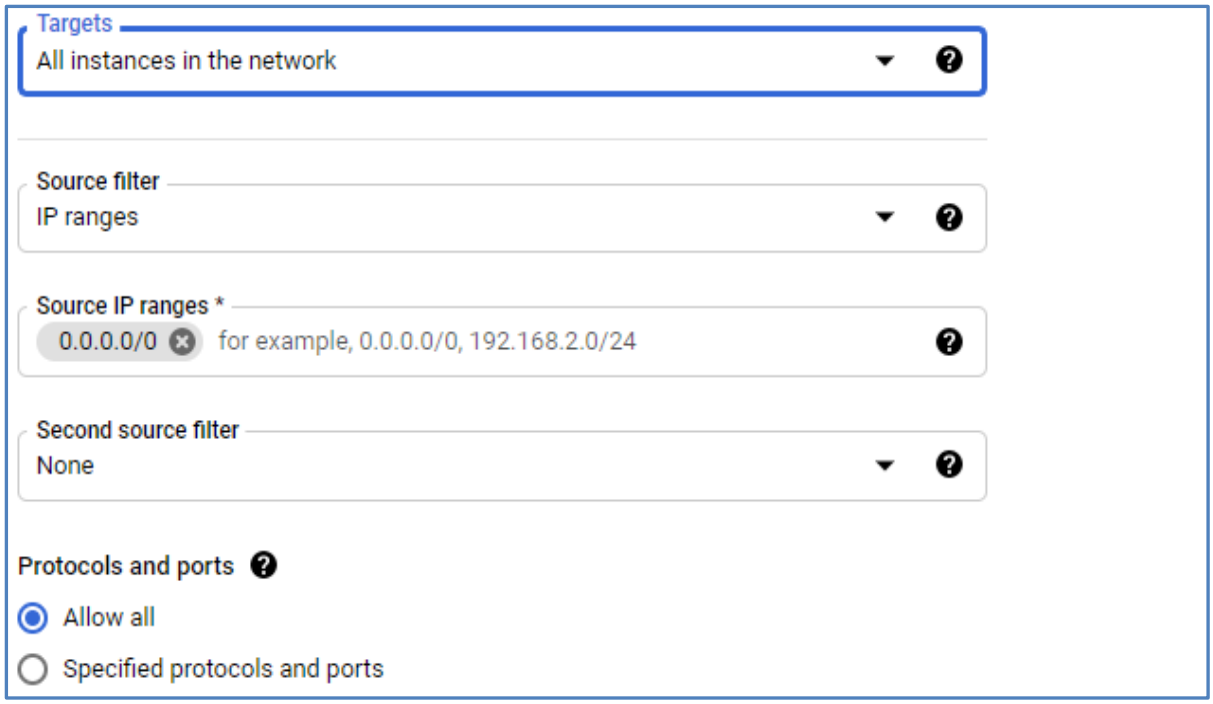
7. Once the instance is created, we will need to allow the following ports in the firewall. For this we go back to main menu, and select VPC Network and there we chose the option Firewall, as shown in the image below-



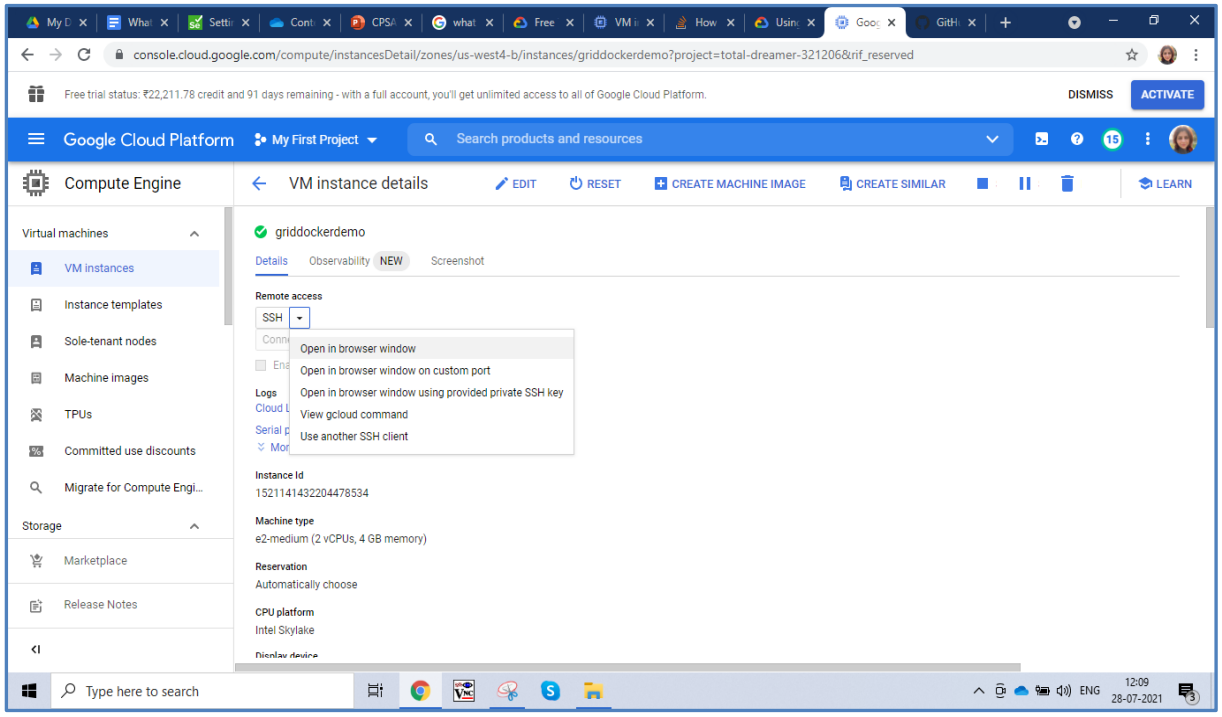
8. The next is to click on Create Firewall Rule on the top



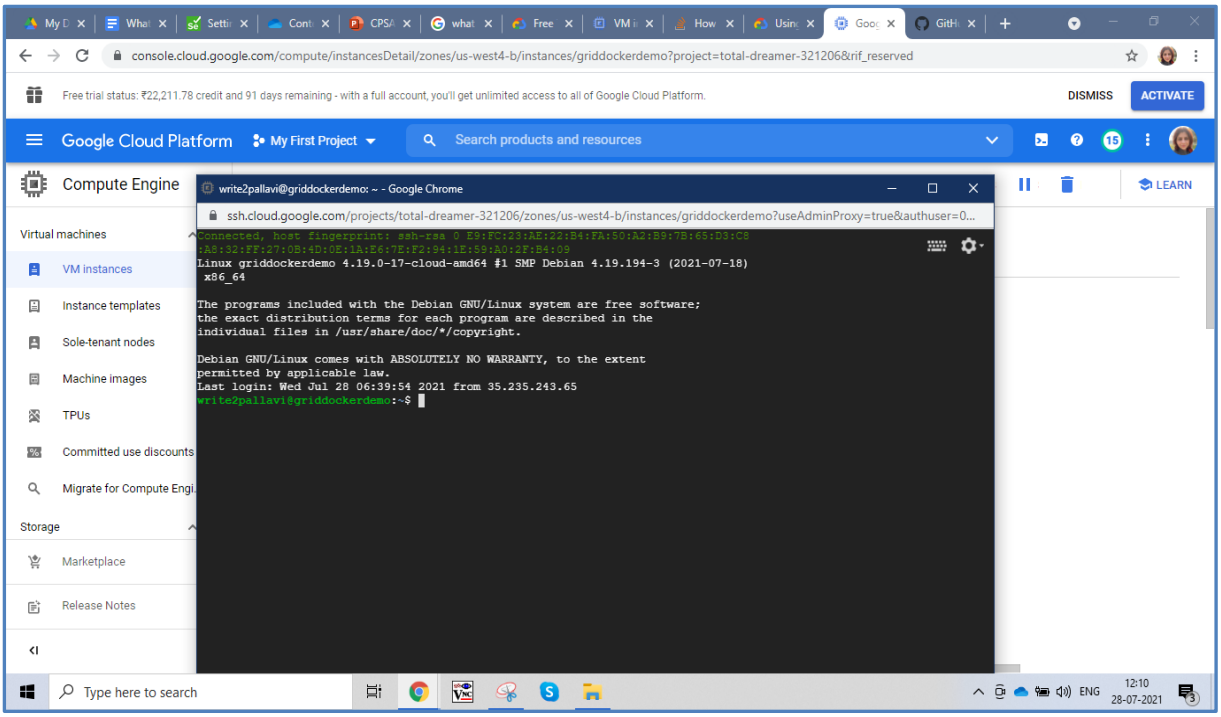
9. Give a name to the firewall rule and provide the settings as per the image below-



10. The next is to select SSH from the virtual machine instance and choose open in the browser window as per the image below



11. It will open a new pop-up window with a console, which will look as follows -



12. Our next step is to get docker on the VM, but for that we will need to change the super user password. For this we will type the command - **sudo passwd.** This will ask us to then provide a new password, which we will retype for the password to change.

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Wed Jul 28 06:39:54 2021 from 35.235.243.65
write2pallavi@griddockerdemo:~$ pwd
/home/write2pallavi
write2pallavi@griddockerdemo:~$ su
Password:
su: Authentication failure
write2pallavi@griddockerdemo:~$ sudo passwd
New password:
Retype new password:
passwd: password updated successfully
write2pallavi@griddockerdemo:~$
```

13. Our next step is to install docker on the machine, for this we will perform the following steps, all the steps are taken from - https://docs.docker.com/engine/install/debian/

a. The first step is to set up the repository

```
$ sudo apt-get update

$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release
```

b. The next is then add the official Docker GPG key-

curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

c. We then setup a stable repository

```
echo \
  "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/debian \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

d. And finally, we install the docker engine using the following commands one after the other-

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

14. The next step is to check the docker installation for this we can type the below command.

```
sudo docker run hello-world
```

```
write2pallavi@griddockerdemo:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest: sha256:df5f5184104426b65967e016ff2ac0bfcd44ad7899ca3bbcf8e44e4461491a9e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

15. We also need to add root permission for Docker. For this run the following command -

    sudo usermod -a -G docker [user]

    In here for user, pass the username.

    If you do not give the root permissions, you may get error. Or you have to login as a root or super user to execute the docker commands

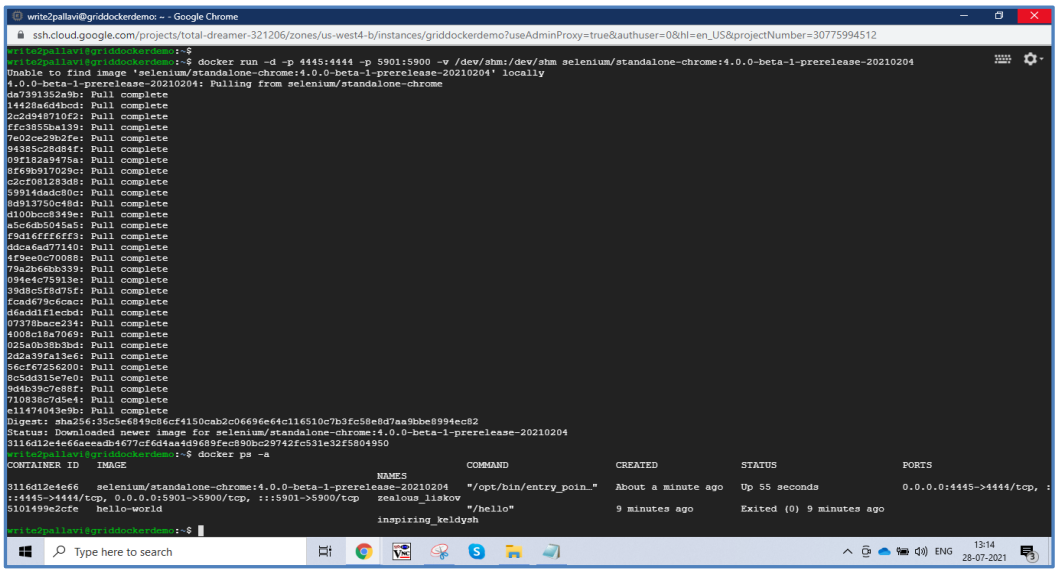
    [Reference link - https://stackoverflow.com/questions/47854463/docker-got-permission-denied-while-trying-to-connect-to-the-docker-daemon-socke]

5.2.2.2 Setting up selenium grid in debug mode using docker selenium

Once we have a linux machine with docker installed, we can go ahead and create grid. We will first start the grid in debug mode, this helps us view the execution.
Let us create a Grid, with Chrome in Debug mode, and we will connect the machine using the VnC viewer to see the execution happening.

The command to start grid with chrome in debug mode is as below.

docker run -d -p 4445:4444 -p 5901:5900 -v /dev/shm:/dev/shm selenium/standalone-chrome:4.0.0-beta-1-prerelease-20210204



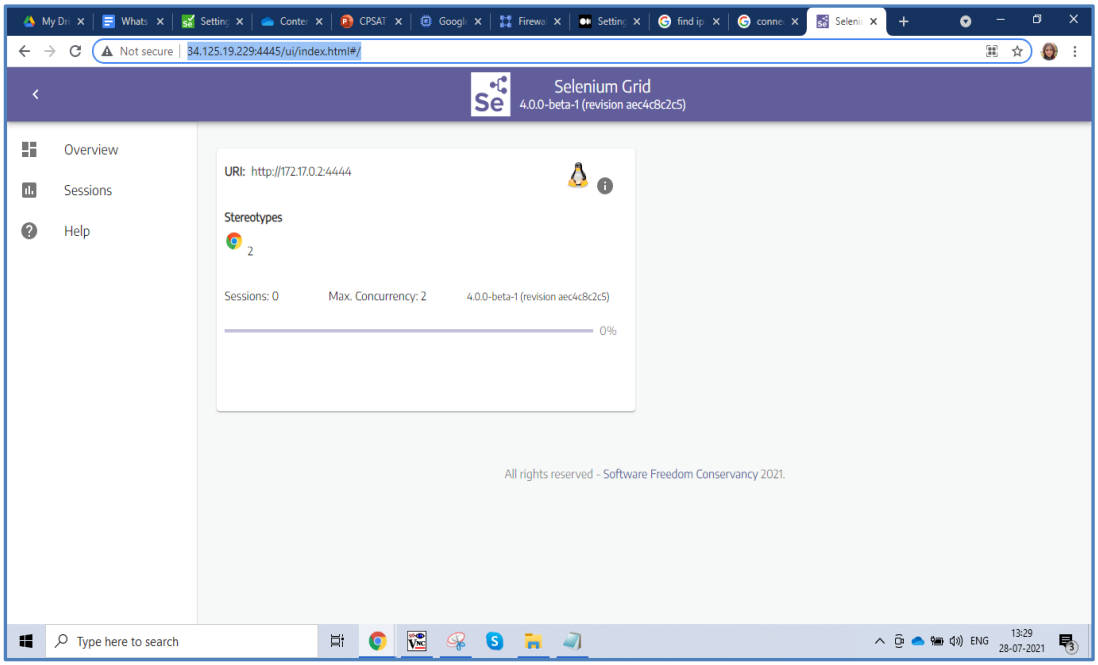
Some quick notes on the docker command used

-d (it is running in the daemon mode) : A daemon is **a type of program on Unix-like operating systems** that runs unobtrusively in the background
-p 4445:4444 (the hub port exposed is 4445 – this is where we will connect our client to)
-p 5901:5900 (the internal VNC viewer port is exposed on port 5901 – this is the port where we will see the node execution in action)
-v /dev/shm:/dev/shm selenium/standalone-chrome:4.0.0-beta-1-prerelease-20210204
(This is the version of the standalone or debug docker hub image that we are using)

If the command ran successfully, it would start the hub components and just like in our previous section, we can see it running if we type the below url. Please note that the IP address is the IP address of your Linux virtual machine.

http://34.125.19.229:4445/ui/index.html#/

And when you type the IP address: exposed port 4445 in the browser you will see the following

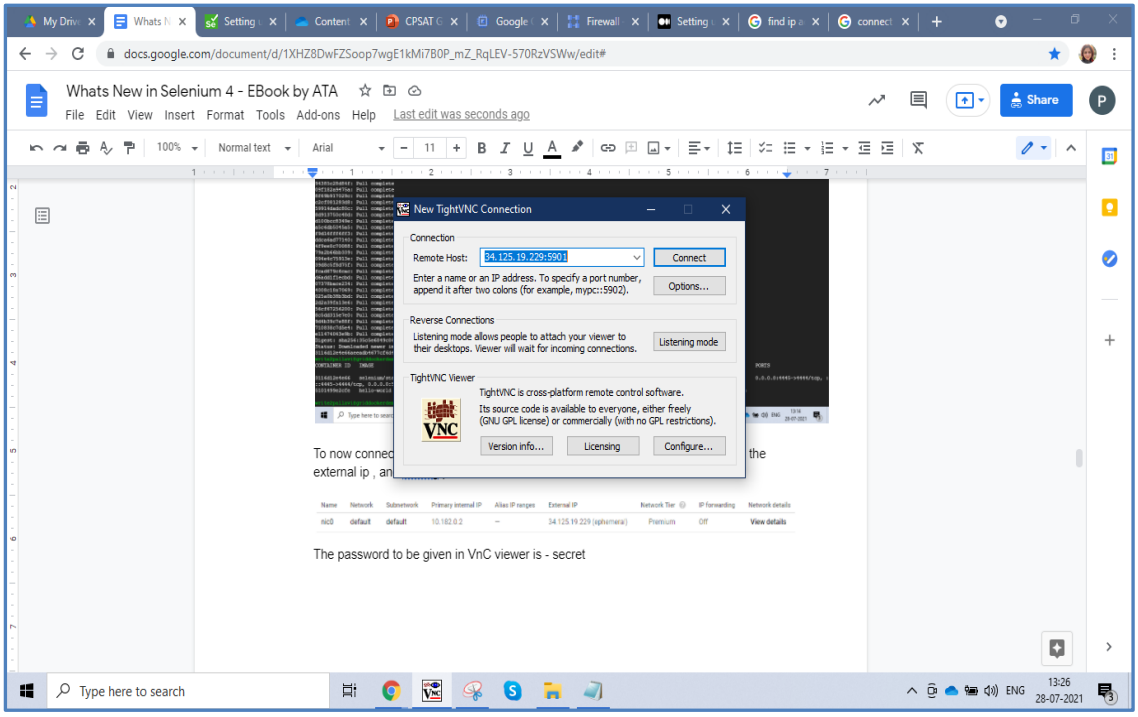


We can now connect to the standalone node using VnC viewer on the client machine (you will have to install VnC Viewer)

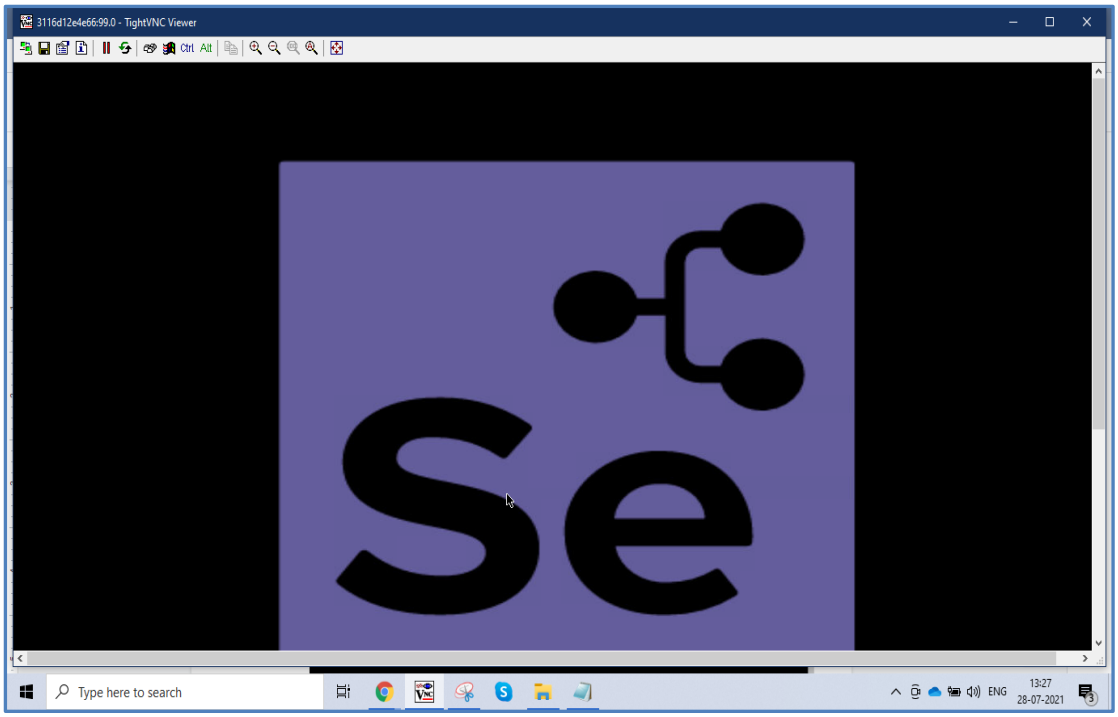
To now connect to the instance using VnC viewer, we will pass the address using the external IP, and adding port number 5901 to it. (Remember we used in our docker command and exposed port 5901)

Name	Network	Subnetwork	Primary internal IP	Alias IP ranges	External IP	Network Tier	IP forwarding	Network details
nic0	default	default	10.182.0.2	–	34.125.19.229 (ephemeral)	Premium	Off	View details

The password to be given in VnC viewer is – secret (This is hardcoded and provided by the standalone selenium docker image that we have used)



We will see the following image once connection is established (This is the node where our selenium scripts will get executed)



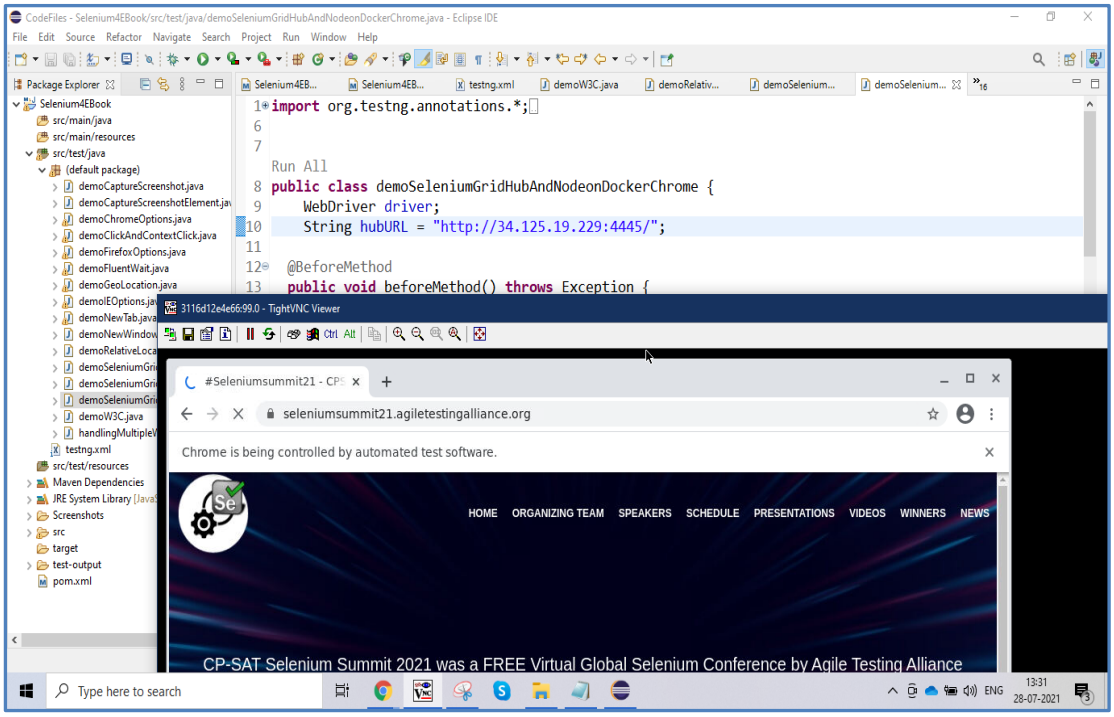
The next step is to run our script from the eclipse. We need to ensure to pass the correct hub url which will be - http://34.125.19.229:4445/ - Rest of the code is same as our earlier code.

Now let us run the script available to us in eclipse

```
import org.TestNG.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.remote.*;
import java.net.URL;
import java.util.List;


public class demoSeleniumGridHubAndNodeonDockerChrome {
      WebDriver driver;
      String hubURL = "http://34.125.19.229:4445/";

  @BeforeMethod
  public void beforeMethod() throws Exception {
        DesiredCapabilities caps = new DesiredCapabilities();
      caps.setBrowserName("chrome");
      driver = new RemoteWebDriver(new URL(hubURL), caps);
  }

  @Test
```

```
  public void launchApplication() throws Exception{

driver.get("https://seleniumsummit21.agiletestingalliance.org/");
        //find all links on the home page and print information
        List<WebElement> links =
driver.findElements(By.xpath("//a"));
        for(WebElement link: links) {
              System.out.println(link.getText());
              System.out.println(link.getAttribute("href"));
        }
  }

  @AfterMethod
  public void afterMethod() {
        driver.close();
  }

}
```

As the execution begins, we will see the following on the VnCviewer (the node)



To summarize what we have done so far - we created a GCP Virtual machine, and installed docker on it. Then we created a docker instance of the grid-chrome debug (Standalone selenium docker hub image) and executed selenium code from our local machine on the node running using docker commands and we viewing it using VnCViewer.

5.2.2.3 Setting up selenium grid using docker compose

We will be using Docker - Compose to start the grid with chrome and firefox nodes on the fly, and then execute our TestNG suite. For this we will need a docker-compose.yml file, and we will need to install docker-compose on the virtual machine. The steps are as follows -

1. First, we must install docker compose on the Virtual machine.
   Please follow the steps below

`sudo curl -L "https://GitHub.com/docker/compose/releases/download/1.28.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose`

```
write2pallavi@griddockerdemo:~$ sudo curl -L "https://github.com/docker/compose/releases/download/1.28.2/docker-com
pose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   633  100   633    0     0   2096      0 --:--:-- --:--:-- --:--:--  2089
100 11.6M  100 11.6M    0     0  20.8M      0 --:--:-- --:--:-- --:--:-- 20.8M
write2pallavi@griddockerdemo:~$
```

After this we will perform the following action on the ssh console

```
sudo chmod +x /usr/local/bin/docker-compose
```

In the above command we have given the executable rights to the docker-compose file available at the /usr/local/bin path. By using the above two commands we have been able to install docker compose

2. The next step is to check the docker compose version for this we will do the following -

```
write2pallavi@griddockerdemo:~$ docker-compose --version
docker-compose version 1.28.2, build 67630359
```

If the command `docker-compose` fails after installation, check your path. You can also create a symbolic link to `/usr/bin` or any other directory in your path.

```
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
```

Please refer the following url for docker compose installation in case things are not working out properly. https://docs.docker.com/compose/install/

3. Create the docker-compose.yml file. The contents of it will look as follows –

```
version: '2'
services:
```

```
chrome:
  image: selenium/node-chrome:4.0.0-rc-1-prerelease-20210804
  shm_size: 2gb
  depends_on:
    - selenium-hub
  environment:
    - SE_EVENT_BUS_HOST=selenium-hub
    - SE_EVENT_BUS_PUBLISH_PORT=4442
    - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
  ports:
    - "6900:5900"

edge:
  image: selenium/node-edge:4.0.0-rc-1-prerelease-20210804
  shm_size: 2gb
  depends_on:
    - selenium-hub
  environment:
    - SE_EVENT_BUS_HOST=selenium-hub
    - SE_EVENT_BUS_PUBLISH_PORT=4442
    - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
  ports:
    - "6901:5900"

firefox:
  image: selenium/node-firefox:4.0.0-rc-1-prerelease-20210804
  shm_size: 2gb
  depends_on:
    - selenium-hub
  environment:
    - SE_EVENT_BUS_HOST=selenium-hub
    - SE_EVENT_BUS_PUBLISH_PORT=4442
    - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
  ports:
    - "6902:5900"

selenium-hub:
  image: selenium/hub:4.0.0-rc-1-prerelease-20210804
  ports:
    - "4442:4442"
    - "4443:4443"
    - "4444:4444"
```

4. According to the above docker compose file, our hub will be on the port 4444, the nodes will be created and listening on port 6900, 6901 and 6902 respectively (chrome / edge / firefox in the order we have listed in the .yml file)

5. The next is to execute using the docker-compose command.

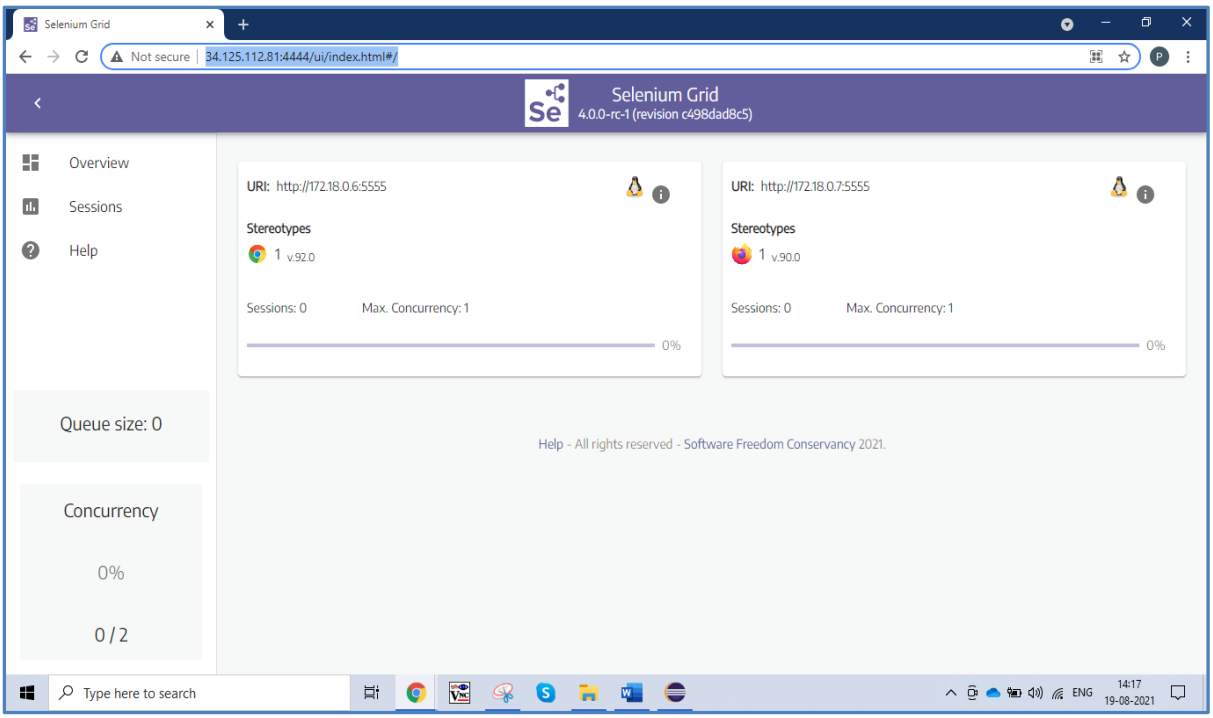
```
write2pallavi@griddockerdemo:~$ /usr/local/bin/docker-compose -f dcompose.yml up -d
```

6. And if we now run the docker ps- a command we will find the instances up and running

```
CONTAINER ID   IMAGE                                                COMMAND                  CREATED              STATUS
       PORTS                                              NAMES
5ee471e54821   selenium/node-chrome:4.0.0-rc-1-prerelease-20210804          "/opt/bin/entry_poin…"   20 seconds ago       Up 18 seconds
       0.0.0.0:9001->5900/tcp, :::9001->5900/tcp          write2pallavi_chrome1_1
6147d28159dc   selenium/node-firefox:4.0.0-rc-1-prerelease-20210804         "/opt/bin/entry_poin…"   20 seconds ago       Up 18 seconds
       0.0.0.0:9002->5900/tcp, :::9002->5900/tcp          write2pallavi_firefox1_1
f49b0bef49bd   selenium/hub:4.0.0-rc-1-prerelease-20210804                  "/opt/bin/entry_poin…"   About a minute ago   Up 19 seconds
       4442-4443/tcp, 0.0.0.0:4446->4444/tcp, :::4446->4444/tcp   write2pallavi_hub_1
```

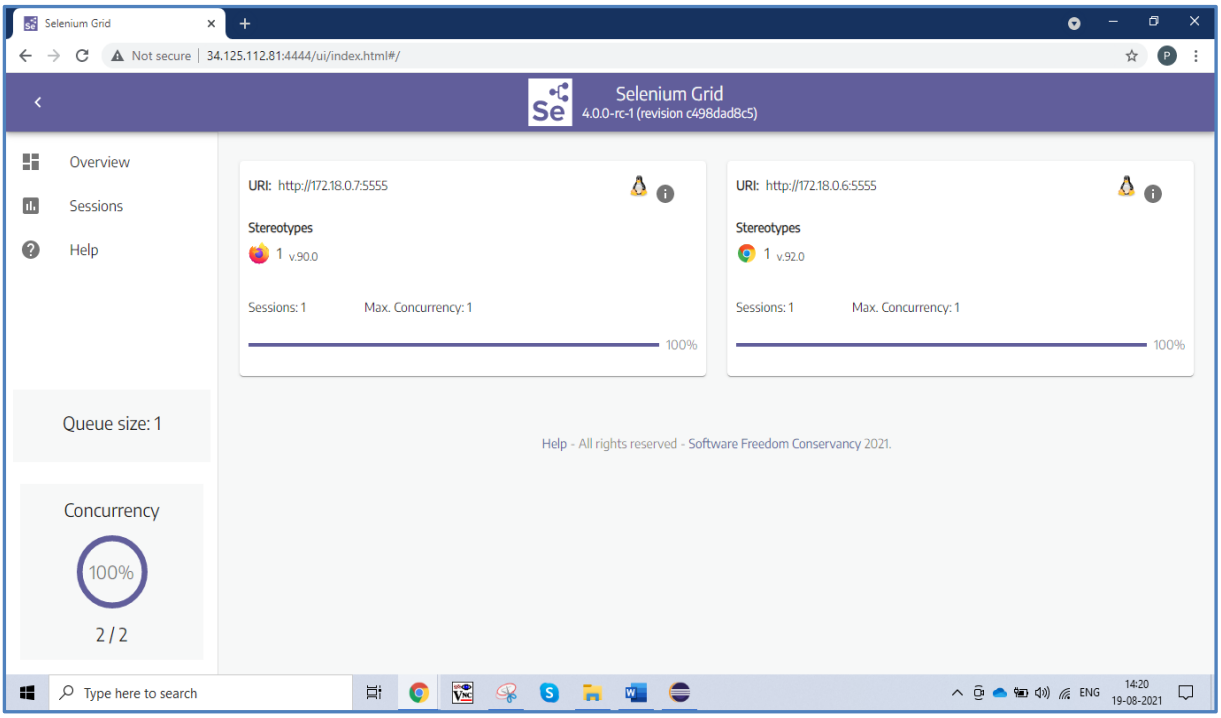
7. The next step is to check if the hub is running, we will be using the url - http://34.125.112.81:4444/ui/index.html#/

It will show the instance of chrome and firefox running



We can now run our TestNG.xml suite file to execute the test on chrome and firefox browser. Remember to change the url of the hub in the scripts to `http://34.125.112.81:4444/` .

Please note the ipaddress which you will have to use, will depend on your virtual machine instance. Once we run the earlier TestNG suite file with the correct hub IP address, we can see the execution under progress in the grid page. We see both the nodes are occupied as our TestNG file executes the program on both chrome and firefox. Unlike the debug mode, there the execution is happening in background. We will see the console output on eclipse if any. The Selenium java script code does not change except the proper IP address for the hub has to be included in the TestNG xml file.



Thus, in the above section, we displayed how using docker compose file and docker compose command, we can create a grid instance on the fly with a hub, and a chrome, edge, and a Firefox node, and running the TestNG suite file on it.

Chapter 6: Deprecation of Desired Capabilities

The Desired Capabilities in Selenium allows us to set the test environment settings with respect to the browser we chose to use. From Selenium 4, instead of using desired capabilities, we will have to use the **Options** object which is made available for different browser. These are as follows

a. FirefoxOptions
b. ChromeOptions
c. InternetExplorerOptions
d. EdgeOptions
e. SafariOptions

More information on the above is available on this page:
https://www.selenium.dev/documentation/en/driver_idiosyncrasies/driver_specific_capabilities/

6.1 InternetExplorerOptions

Let us take few examples for different browsers. The first browser we will select is Internet Explorer. Generally, when we are working with IE, we need to ensure that the zoom setting is ignored. In case it is not, and it the browser launched has any other zoom setting and not 100, our scripts won't run. In the earlier version of Selenium, we used to do the same using the Desired capabilities. But from Selenium 4, we will have to use the **InternetExplorerOptions** to achieve it.

We are using the `ignoreZoomSettings();` method of `InternetExplorerOptions();`
So that the IE opens correctly. Please see the code below.

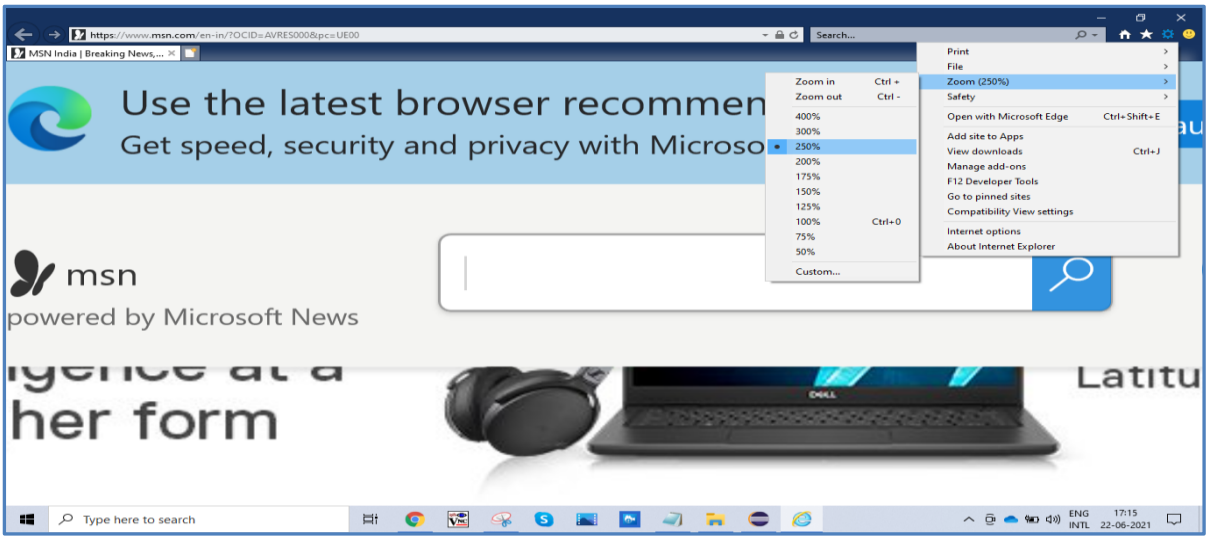
```
import org.TestNG.annotations.*;
import java.util.Iterator;
import java.util.Set;
import org.openqa.selenium.*;
import org.openqa.selenium.ie.*;
import io.GitHub.bonigarcia.wdm.WebDriverManager;

public class demoIEOptions {

     WebDriver driver;

  @BeforeMethod
```

```
  public void beforeMethod() {
        //create webdriver for ie driver
        WebDriverManager.iedriver().setup();
        //Create option class object to ignore zoom settings
        InternetExplorerOptions options = new
InternetExplorerOptions();
        options.ignoreZoomSettings();
        driver = new InternetExplorerDriver(options);
  }

  @Test
  public void openApp() throws Exception{
        //open application

driver.get("https://seleniumsummit21.agiletestingalliance.org/");
        //wait for 2 seconds before browser closes to see the
action done
        Thread.sleep(2000);
  }

  @AfterMethod
  public void afterMethod() {
        //close the browser
       driver.quit();
  }
}
```

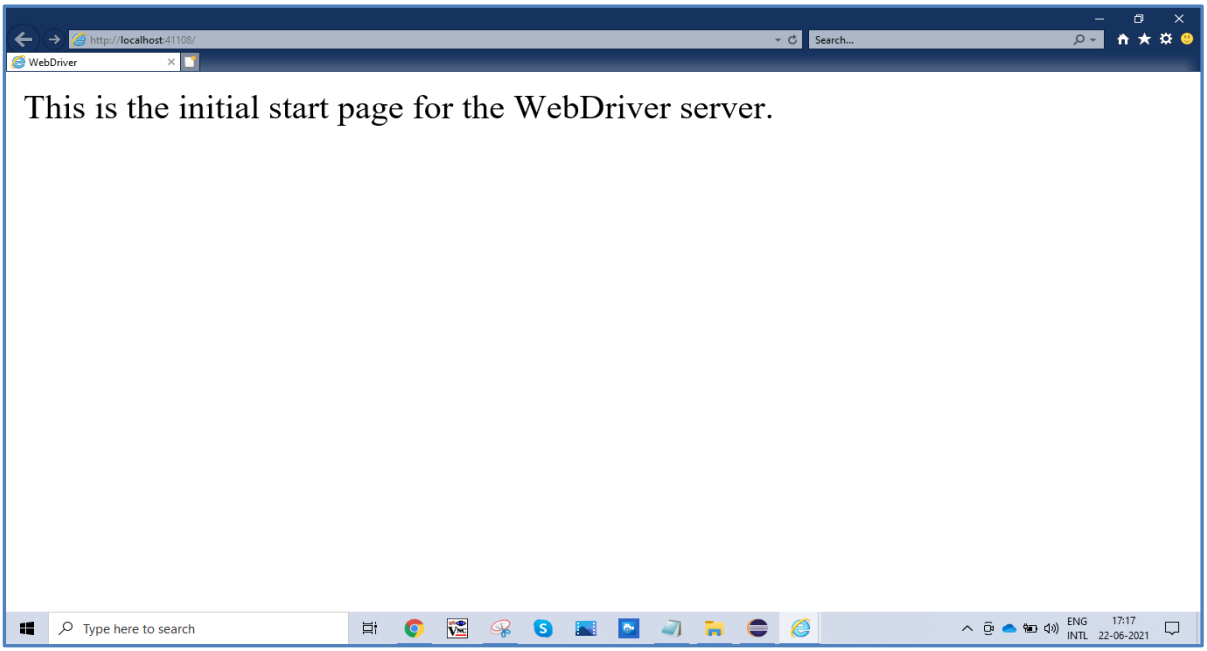
What would happen if we do not use this? Run the same code after commenting out the below line

```
 // options.ignoreZoomSettings();
```

Before you execute the code first open the IE browser on our system and set its zoom setting to 250% like in the image below.



If we now execute the code (with the commented line for `ignoreZoomSettings`) you will see an error like in the image below.



All the methods for the InternetExplorerOptions class can be looked at from the Javadoc link below.

https://www.javadoc.io/doc/org.seleniumhq.selenium/selenium-ie-driver/latest/org/openqa/selenium/ie/InternetExplorerOptions.html

6.2 FirefoxOptions

Let us now take Firefox browser example, where we want to set a custom profile, whenever the browser launches to execute test script using Selenium. We will have to use the **FirefoxOptions** class, and its method. We use it to create a fresh profile for firefox to use it while launching for new test run. The code look as follows

```
import org.TestNG.annotations.*;
import java.util.Iterator;
import java.util.Set;
import org.openqa.selenium.*;
import org.openqa.selenium.firefox.*;
import io.GitHub.bonigarcia.wdm.WebDriverManager;


public class demoFirefoxOptions {

     WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
        //create webdriver for ie driver
        WebDriverManager.firefoxdriver().setup();
        //Create option class object to set custom profile
        FirefoxProfile profile = new FirefoxProfile();
        FirefoxOptions options = new FirefoxOptions();
        options.setProfile(profile);
        driver = new FirefoxDriver(options);
  }

  @Test
  public void openApp() throws Exception{
        //open application

driver.get("https://seleniumsummit21.agiletestingalliance.org/");
        //wait for 2 seconds before browser closes to see the
action done
        Thread.sleep(2000);
  }

  @AfterMethod
  public void afterMethod() {
```

```
        //close the browser
       driver.quit();
  }

}
```

Please note that FirefoxProfile() class and FirefoxOptions() have been there in Selenium 3.x. We are just highlighting that rather than using Desired Capabilities, we would now be using the FirefoxOptions() class for setting the profile.

Javadoc for FirefoxOptions() and FirefoxProfile() are available on the following url's

https://www.javadoc.io/doc/org.seleniumhq.selenium/selenium-firefox-driver/latest/org/openqa/selenium/firefox/FirefoxOptions.html

https://www.javadoc.io/doc/org.seleniumhq.selenium/selenium-firefox-driver/latest/org/openqa/selenium/firefox/FirefoxProfile.html

6.3 ChromeOptions

For the chrome browser, we will have to use the **ChromeOptions** class, and in here we will use the method **headless()** and pass true in it. If we do that, and we run the script the Chrome browser will be launched in headless mode, which means we will not see it opening and performing any steps. The code for it is as follows -

```
import org.TestNG.annotations.*;
import java.util.Iterator;
import java.util.Set;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.*;
import io.GitHub.bonigarcia.wdm.WebDriverManager;


public class demoChromeOptions {

     WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
       //create webdriver for chrome driver
       WebDriverManager.chromedriver().setup();
```

```
        //Create option class object and make chrome browser run in
headless mode.
        ChromeOptions options = new ChromeOptions();
        options.setHeadless(true);
        driver = new ChromeDriver(options);
  }

  @Test
  public void openApp() throws Exception{
        //open application

driver.get("https://seleniumsummit21.agiletestingalliance.org/");
        //wait for 2 seconds before browser closes to see the
action done
        Thread.sleep(2000);
  }

  @AfterMethod
  public void afterMethod() {
        //close the browser
       driver.quit();
  }

}
```

Javadoc for ChromeOptions is on the following URL.

https://www.javadoc.io/doc/org.seleniumhq.selenium/selenium-chrome-driver/latest/org/openqa/selenium/chrome/ChromeOptions.html

It is recommended that you also look at ChromiumOptions() as ChromeOptions() class extends ChromiumOptions() class. E.g., setHeadless() method is inherited from ChromiumOptions()

https://www.selenium.dev/selenium/docs/api/java/index.html?org/openqa/selenium/chromium/ChromiumOptions.html

In this chapter we have seen, that from Selenium 4 onwards we will need to use the browser respective Options class to set their settings for the test execution, and desired capabilities will no longer work.

Chapter 7: Change in Fluent Wait Method

There are different wait methodologies available to us in Selenium. These are implicit wait, explicit wait, and fluent wait. For more information on these please visit the section - https://www.selenium.dev/documentation/en/webdriver/waits/. Here we will mainly understand what Fluent wait is, and what has changed for this method in Selenium 4.

Fluent wait in Selenium, allows us to wait for the maximum time, until a certain condition is met. It also defines how many times the webdriver will poll to check if the condition appears before throwing the "**ElementNotVisibleException**". This method primarily looks for a condition to happen for a web element, by polling at regular intervals for a given time. And if the condition is not met within the given time duration it throws the exception mentioned above.

In the Selenium 3, the syntax to be used for the FluentWait looked as follows -

```
FluentWait wait = new FluentWait(driver)
                    .pollingEvery(500, TimeUnit.MILLISECONDS)
                    .withTimeout(20, TimeUnit.SECONDS)
                    .ignoring(NoSuchElementException.class);
```

The above fluent wait method will poll every 500 milliseconds and will timeout after 60 seconds.

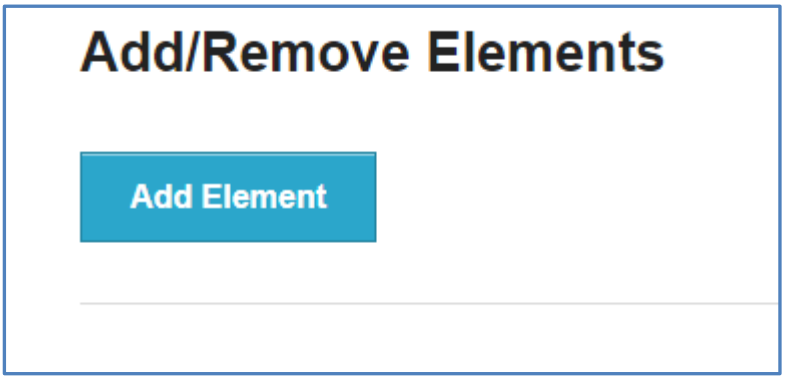
But with Selenium 4, we now only pass one parameter to both pollingEvery(), and the withTimeout() method. The parameter we pass now is that of the **Duration** class in java, which is available in the java.time package. Similarly, we pass a Duration class in the withTimeout().

So now, in Selenium 4 the FluentWait will look as follows -

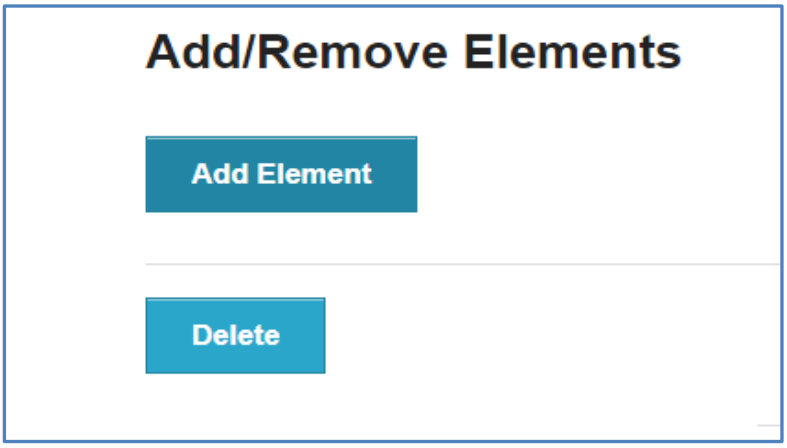
```
FluentWait wait = new FluentWait(driver)
                    .pollingEvery(Duration.ofMillis(500))
                    .ignoring(NoSuchElementException.class)
                    .withTimeout(Duration.ofSeconds(20));
```

Both the Selenium 3.x and Selenium 4 fluent wait method will poll every 500 milli seconds and will timeout after 60 seconds. Difference is the way the polling and timeout now takes Duration class. Please see the highlighted part in the both the code to understand the difference.

Let us look at an example of using Fluent Wait. We will take the scenario available on the url to add and delete elements. The url is - https://the-internet.herokuapp.com/add_remove_elements/



As per the above image if we click on the Add Element, it adds an element on the page, which is a button with text Delete. And when we click on Delete, that element gets removed. So, we go back to the page as shown above.



In the scenario which we are going to automate using Selenium, we will perform the following steps

a. Launch the application using the url- https://the-internet.herokuapp.com/add_remove_elements/
b. Click on the button Add element.
c. We will use Fluent Wait to wait for the element Delete to appear.
d. When Delete appears we click on it
e. Close the application.

The code for the above scenario looks as follows -

```
import org.TestNG.annotations.*;
import org.openqa.selenium.*;
```

```
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.FluentWait;
import io.GitHub.bonigarcia.wdm.WebDriverManager;
import java.time.Duration;

public class demoFluentWait {
      WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
  }

  @Test
  public void fluentMethodExample() throws Exception{
driver.get("https://the-internet.herokuapp.com/add_remove_elements/");
        Thread.sleep(2000); //this wait is introduced so that we are able
to observe add element being clicked.
        driver.findElement(By.xpath("//*[contains(text(),'Add
Element')]")).click();
        //creation of fluent wait object with
        FluentWait wait = new FluentWait(driver)
              .pollingEvery(Duration.ofMillis(500))
              .ignoring(NoSuchElementException.class)
              .withTimeout(Duration.ofSeconds(20));

wait.until(ExpectedConditions.presenceOfElementLocated(By.xpath("//*[cont
ains(text(),'Delete')]")));
        Thread.sleep(2000);//this wait is introduced so that we are able
to observe delete element being clicked.
driver.findElement(By.xpath("//*[contains(text(),'Delete')]")).click();
       }

  @AfterMethod
  public void afterMethod() {
        driver.close();
  }

}
```

Thus, as the code executes, we can see the page getting launched, appearance of the Delete button as we click on the Add Elements. And then the Delete button disappears as we click on the Delete button.

Chapter 8: Improved Screenshot Method

As we automate our test scenarios, an important element when the test executes is to capture the screenshot, which helps in debugging and helps logging more informative bugs in the system. In Selenium 3.0 we had the ability to capture the screenshot of the screen by using the TakesScreenShot interface, and we used the method getScreenshotAs().

We should ideally capture screenshot during error and exception situations which will help us find issues faster. The issue could be either in the application or it could be in the code we generated to automate the process. Screenshot capturing during unattended test runs is crucial and it should be done.

From Selenium 4.0 onwards besides capturing the screenshot of the screen, we can also capture the screenshot of a particular web element. In this section we will see the two examples where we take the screenshot of the entire web page, and when we only pass the element instance to the TakesScreenshot to capture the web element only.

The TakesScreenShot interface details are available here

https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/TakesScreenshot.html

The method which we use in this interface is - getScreenshotAs(OutputType<X> target) throws WebDriverException. When we need to capture the entire page our code snippet looks as follows

```
WebElement element = driver.findElement(By.id("flash-messages"));
File scrScreen= ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
```

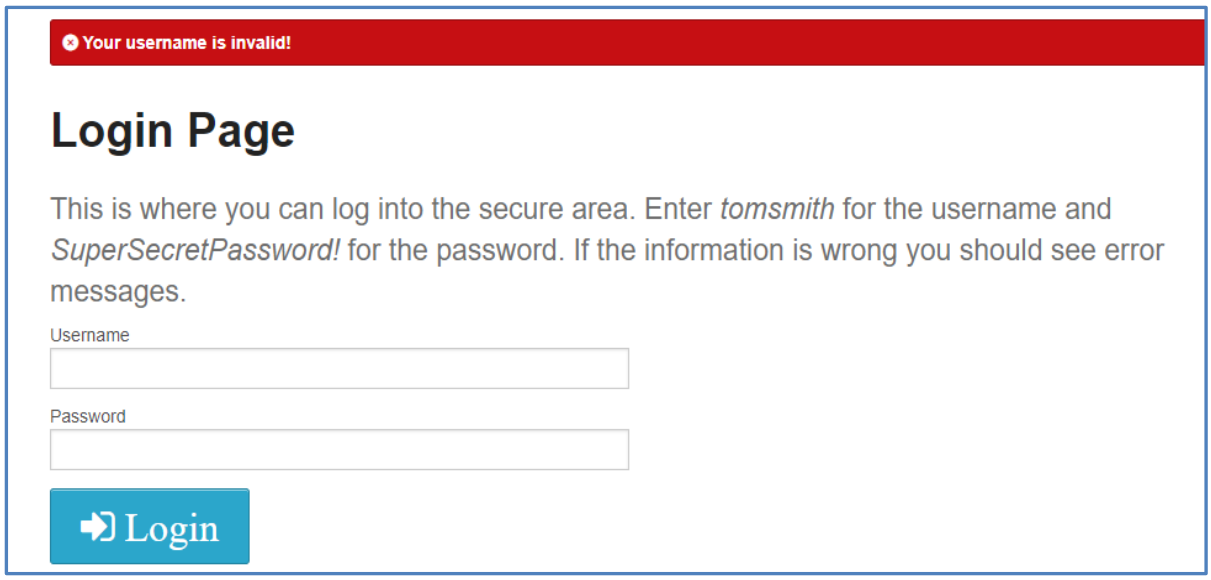
The OutputType is separate interface in Selenium, which captures the screenshot in different formats. The OutputType could be as follows -

a. a BASE64 data
b. As raw bytes
c. As a file.

Let us see an example of screenshot capture. The steps which we will follow are as follows -

a. Open the link -https://the-internet.herokuapp.com/login
b. Enter the username as admin and password as admin

c. Click on Login, as it is a wrong username password combination, we will get the user invalid message and the following screenshot -

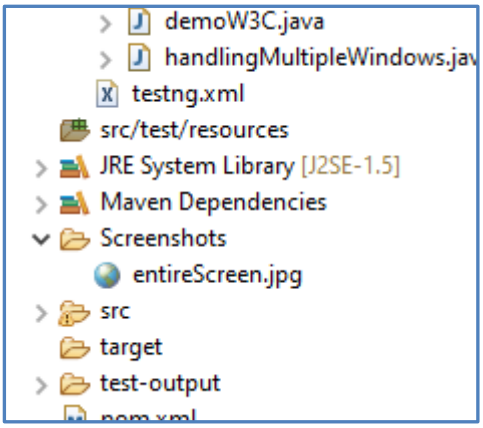


The script for the above will look as follows -

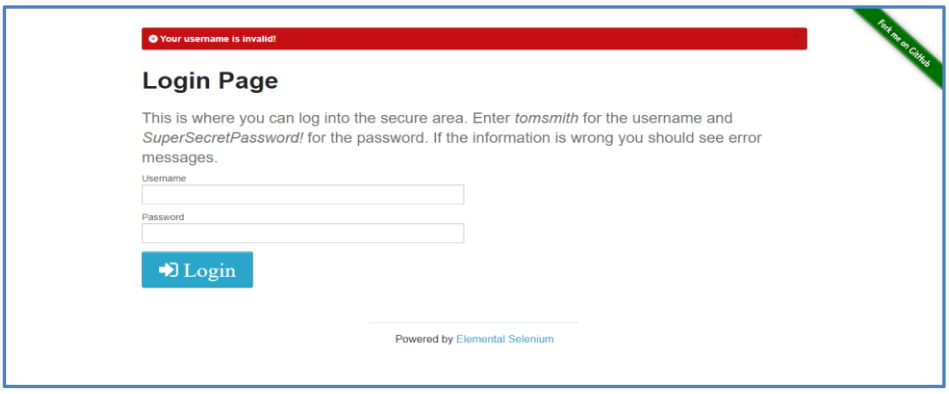
```
import org.TestNG.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import io.GitHub.bonigarcia.wdm.WebDriverManager;
import java.io.File;
import org.apache.commons.io.FileUtils;


public class demoCaptureScreenshot {
      WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
  }

  @Test
  public void openApp() throws Exception{
        driver.get("https://the-internet.herokuapp.com/login");
        //Click on social media icon
        driver.manage().window().maximize();
        driver.findElement(By.name("username")).sendKeys("admin");
        driver.findElement(By.name("password")).sendKeys("admin");
```

```
        driver.findElement(By.className("radius")).click();
        if(driver.getPageSource().contains("invalid")) {
              //Capturing Screenshot of entire screen
              TakesScreenshot scrShot =((TakesScreenshot) driver);
              File scrScreen=scrShot.getScreenshotAs(OutputType.FILE);
              //Move image file to new destination
              File dstScreen=new File("Screenshots\\entireScreen.jpg");
              //Copy file at destination
              FileUtils.copyFile(scrScreen, dstScreen);
              Thread.sleep(2000);
            }
      }

  @AfterMethod
  public void afterMethod() {
        driver.close();
  }

}
```

When we run this code, we will find in the Screenshots folder, a file with name entireScreen.jpg available.



And the image captured is as follows



Please note that the above code is same as in Selenium 3.x. It is important for us to go through the above code to understand the difference that has been brought in Selenium 4.

Since Selenium 4 allows us to capture individual elements as well, every WebElement object can be now casted to TakeScreenshot, and hence individual element screenshots are feasible.

In the example code below, we need to take screen shot of the flash message element, hence we cast the flash message element object to the TakesScreenshot instead of the driver object.

The modified code looks like below

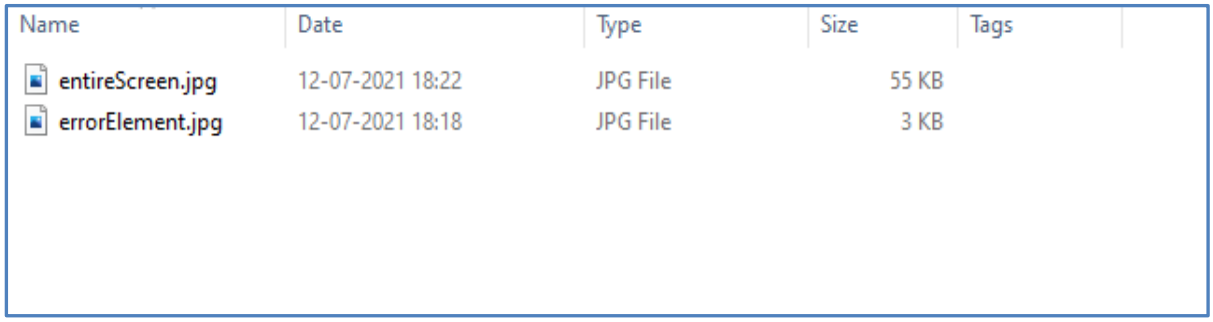
```
WebElement element = driver.findElement(By.id("flash-messages"));
File scrElement = ((TakesScreenshot)
element).getScreenshotAs(OutputType.FILE);
```

The rest of the code will remain the same. The complete code snippet is as follows.

```
import org.TestNG.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import io.GitHub.bonigarcia.wdm.WebDriverManager;
import java.io.File;
import org.apache.commons.io.FileUtils;


public class demoCaptureScreenshotElement {
      WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
  }

  @Test
  public void openApp() throws Exception{
        driver.get("https://the-internet.herokuapp.com/login");
        //Click on social media icon
        driver.manage().window().maximize();
        driver.findElement(By.name("username")).sendKeys("admin");
        driver.findElement(By.name("password")).sendKeys("admin");
        driver.findElement(By.className("radius")).click();
        if(driver.getPageSource().contains("invalid")) {
            //Capturing Screenshot of an element
              WebElement element = driver.findElement(By.id("flash-
messages"));
              File scrElement = ((TakesScreenshot)
element).getScreenshotAs(OutputType.FILE);
              File dstElement = new
File("Screenshots\\errorElement.jpg");
              FileUtils.copyFile(scrElement, dstElement);
              Thread.sleep(2000);
            }
       }

  @AfterMethod
  public void afterMethod() {
        driver.close();
  }

}
```

The above code will generate the image file called as errorElement in the ScreenShots folder, and it will look as follows

We will find in here that by capturing only the screenshot of the error web element, we also get a reduced size of image files generated which can help us in the longer run. Look at the following image



Thus, by using the new improved TakesScreenshot interface in Selenium 4 we can generate better test code, and logging information required in form of images.

Chapter 9: Chrome Dev Tools Protocol

The Chrome Dev Tools Protocol or also known as CDP allows tools to inspect, debug, instrument and profile Chromium, Chrome and Blink based browsers. And Chrome Dev Tools is a set of components build together to help developers investigate websites on the fly during development. There are various things we can do using Chrome Dev Tools; a detailed list and information is available on this link - https://developer.chrome.com/docs/devtools/ . Some of the things we can do are listed below as well.

a. Inspect Element in the DOM
b. Mock network speeds to understand website behavior
c. Mock geolocations to mock user behavior
d. Check and monitor website performance
e. Edit element and its CSS at run time.
f. Execute JavaScript and many more as available on the link shared above.

Selenium with its version 4 release will now support Chrome Dev Tools. This means that now using Selenium we can access the different features set the Chrome Dev Tools make available for the website. Thus, as the script written using Selenium executes, we at run time will be able to use features like - fetching application cache, monitor performance, mock networks, mock geolocations, profile the web application, change element information, change element CSS etc.
More details on it are available on the Selenium documentation. https://www.selenium.dev/documentation/en/support_packages/chrome_devtools/

From Selenium version 4 onwards, we get a new class introduced which is called as the **ChromiumDriver()** class. More details on it are available here.

https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/chromium/ChromiumDriver.html .

It has various methods, the basic two methods which allows access to the Chrome Dev Tools are **getDevTools**() and the **executeCdpCommand**().

The executeCdpCommand() method executes a direct CDP command and returns the result.
The other way is to use the getDevTools() method. This method returns the object of the DevTools class

https://www.selenium.dev/selenium/docs/api/java/org/openqa/selenium/devtools/DevTools.html

This class has a method send(), which we can then use to send the commands to the Chrome Dev Tools using the built in Selenium wrapper commands. Let us understand them a bit more by using these commands in a sample scenario.

9.1 Mocking GeoLocation using Chrome Dev Tools

The first scenario we will take from Chrome Dev Tools is to mock the GeoLocation.

Let us understand the basic scenario first.

GeoLocation word comes by combining Geography and Locations. Many a times our web applications are designed to behave in a different manner based on from which location they are being accessed. A simple example is of any search website like Google, which produces different search results based on from which location user is trying to access it. This makes it as an important test scenario, and something which we can now automate using Selenium, as the version 4 provides the support. So let us first understand the application example scenario we are going to pick for it.

To showcase GeoLocation testing, we will pick up a web application of a popular sports apparel Puma and click on the link to find a store near us. Based on the user location it should display the puma store near to the user current geographical location.

To perform this, we will first have to create a Map object which takes into consideration the latitude, the longitude, and the accuracy. We will be basically calling the Chrome Dev Tools method **Emulation.setGeoLocationOverride()** and call it using the **executeCdpCommand()** method.

More details on the CDP method are available here - https://chromedevtools.GitHub.io/devtools-protocol/tot/Emulation/#method-setGeolocationOverride

To pass the information for the latitude and longitude, we will use google map to find it. The first we will find for Delhi, India, the details are as per the url we get is:

https://www.google.com/maps/place/Delhi/@**28.6472799,76.8130644**,10z/data=!3m1!4b1!4m5!3m4!1s0x390cfd5b347eb62d:0x37205b715389640!8m2!3d28.7040592!4d77.1024902

The part in bold shows the co-ordinates. And the next we will do for Dubai, UAE. The url we get is as follows:

https://www.google.com/maps/place/Dubai+-+United+Arab+Emirates/@**25.0763815,54.9475464**,10z/data=!3m1!4b1!4m5!3m4!1s0x3e5f43496ad9c645:0xbde66e5084295162!8m2!3d25.2048493!4d55.2707828

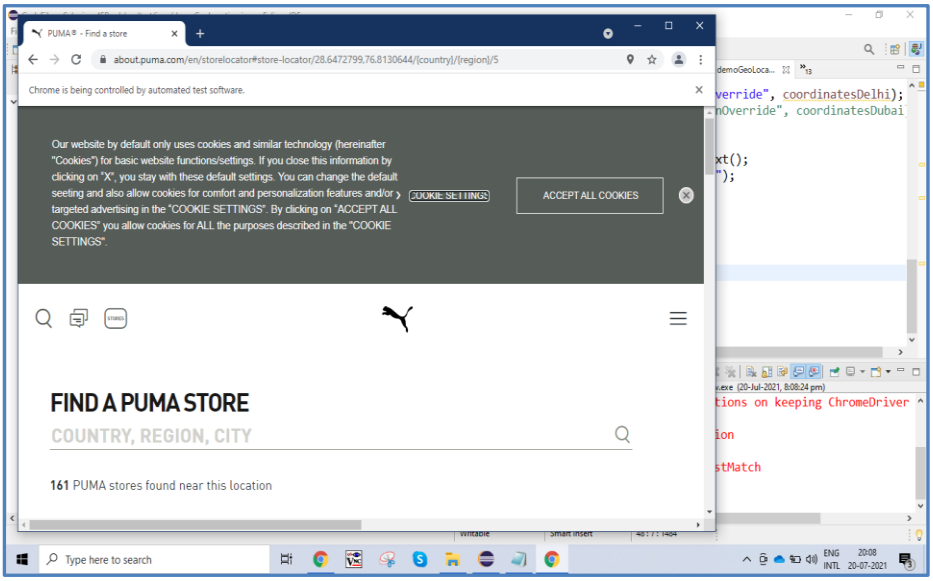
We need to note in here is that the first is the latitude and the second number is the longitude.

Let us see the code below, and the output we get as we pass the data for the latitude and longitude fetched, in the code we have taken for Delhi and Dubai, and as we comment one and execute another, we will be able to see different output. So, for Delhi we as per writing of this book get 161 store locators and for Dubai, we get 12 store locators. The code for the same is as follows.

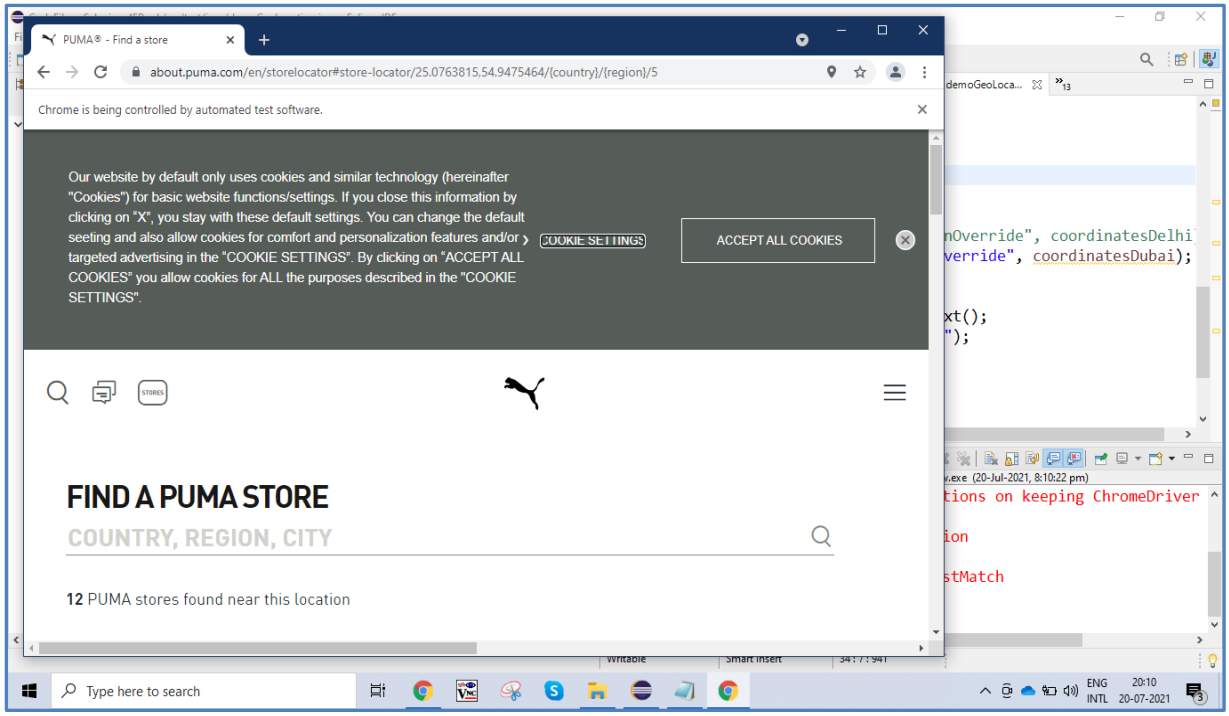
```
import org.TestNG.annotations.*;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chromium.ChromiumDriver;
import io.GitHub.bonigarcia.wdm.WebDriverManager;
import java.util.Map;
import java.util.concurrent.TimeUnit;

public class demoGeoLocation {
      WebDriver driver;

  @BeforeMethod
  public void beforeMethod() {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
        driver.manage().timeouts().implicitlyWait(20, TimeUnit.SECONDS);
  }

  @Test
  public void findPumaStore() throws Exception{

        //Map coordinates Delhi India
        Map coordinatesDelhi = Map.of(
              "latitude", 28.6472799,
              "longitude", 76.8130644,
              "accuracy", 1
      );
      //Map coordinates Dubai UAE
      /*  Map coordinatesDubai = Map.of(
              "latitude", 25.0763815,
              "longitude", 54.9475464,
              "accuracy", 1
```

```
      );
        */
      String totalStores;

       ((ChromiumDriver)
driver).executeCdpCommand("Emulation.setGeolocationOverride",
coordinatesDelhi);
       ///((ChromiumDriver)
driver).executeCdpCommand("Emulation.setGeolocationOverride",
coordinatesDubai);
     driver.navigate().to("https://about.puma.com/en/storelocator");
     Thread.sleep(5000);
     totalStores=driver.findElement(By.className("number-of-
stores")).getText();
     System.out.println(totalStores+ " PUMA stores found near this
location");
}


  @AfterMethod
  public void afterMethod() {
        driver.close();
  }

}
```

As we run the program by passing Delhi coordinates, we will get the following image as the web application launches based on the geolocation given.



And when we run the same code, but now passing the geolocation for Dubai, UAE the image we get while executing the code is as follows -



If you also notice carefully, you will find in the url the store locator coordinates being passed are different for both locations. Thus, using the version of Selenium 4, we can use the Chrome Dev Tools feature to emulate GeoLocation for our web applications and verify the behaviour as per business requirements.

This is just one of the examples of how we can use the Chrome Dev Tools features. As shared in the starting of the section, we can use it for various other scenarios like monitoring performance, executing javascript, changing element style, simulating network speed etc. More such can be found directly on the Selenium website: https://www.selenium.dev/documentation/en/support_packages/chrome_devtools/

Chapter 10: Improved Selenium IDE

Selenium IDE stands for Selenium integrated development environment. It is one of the major components of the Selenium project and is used primarily for recording and playback. The earlier version of Selenium IDE was only supported on the Firefox browser. From the Firefox browser version 55 onwards the Selenium IDE no longer worked. The reason for this was the change from the Mozilla specific XPI format to Web extension mechanism for the Add-ons. A few years back the Selenium IDE was revived and made again available to the Selenium community. The effort behind this was driven by the organization Applitools. More details about this are available here - https://applitools.com/blog/selenium-ide-with-dave-haeffner/

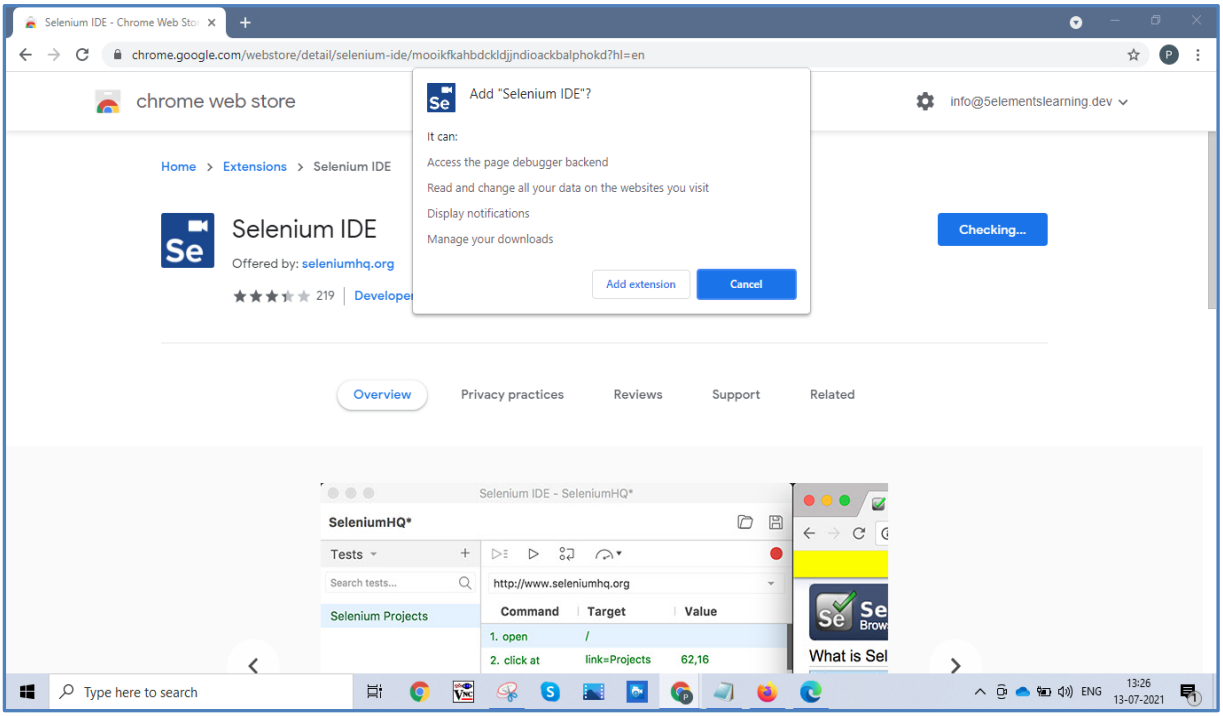
Nevertheless, the new improved Selenium IDE has following features:

a. A new improved user interface which is intuitive to work with.
b. Available as add on all three major browsers - Chrome, Firefox and now also on Microsoft Edge. The links for those are here -
    i. Chrome- https://chrome.google.com/webstore/detail/selenium-ide/mooikfkahbdckldjjndioackbalphokd?hl=en
    ii. Firefox- https://addons.mozilla.org/en-US/firefox/addon/selenium-ide/
    iii. Microsoft Edge- Selenium IDE - Microsoft Edge Addons
c. This means we can now record and playback test using Selenium IDE on all the three browsers. Earlier only firefox was supported.
d. We need to also know in here, that a project created in Selenium IDE is saved with an extension **.side.**
e. We can export the scripts which we record in the Selenium IDE, into desired language bindings which could be java, python, c# etc. This feature was also available in the earlier version of Selenium IDE.
f. The control flow mechanism is also improved with the new Selenium IDE which means we can write better scripts and play them back.
g. The new Selenium IDE also comes with a new Selenium IDE runner. This means it allows us to run the recorded tests on a node.js platform. This means we can run the recorded tests of Selenium ide on the local grid and as well cloud.

Let us take a scenario and use Selenium IDE to record it, playback it. We will export the test in java language binding.

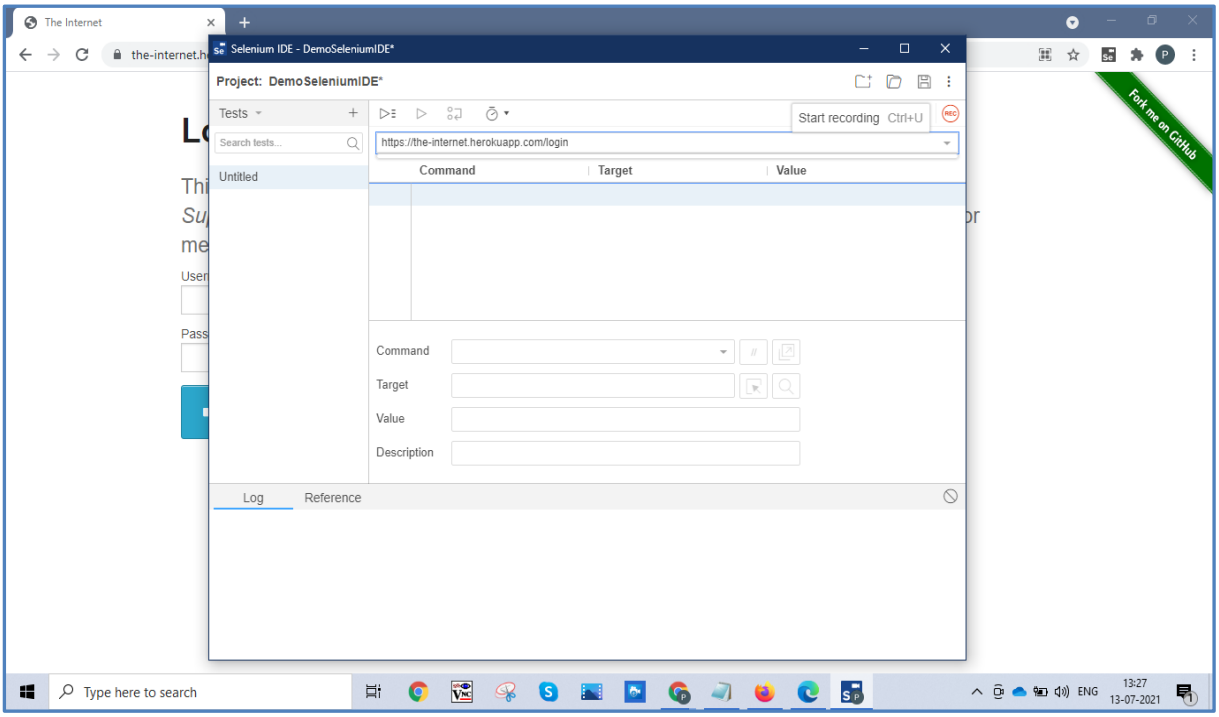
Requirement

a. Latest version of chrome browser
b. Selenium IDE extension available on it.

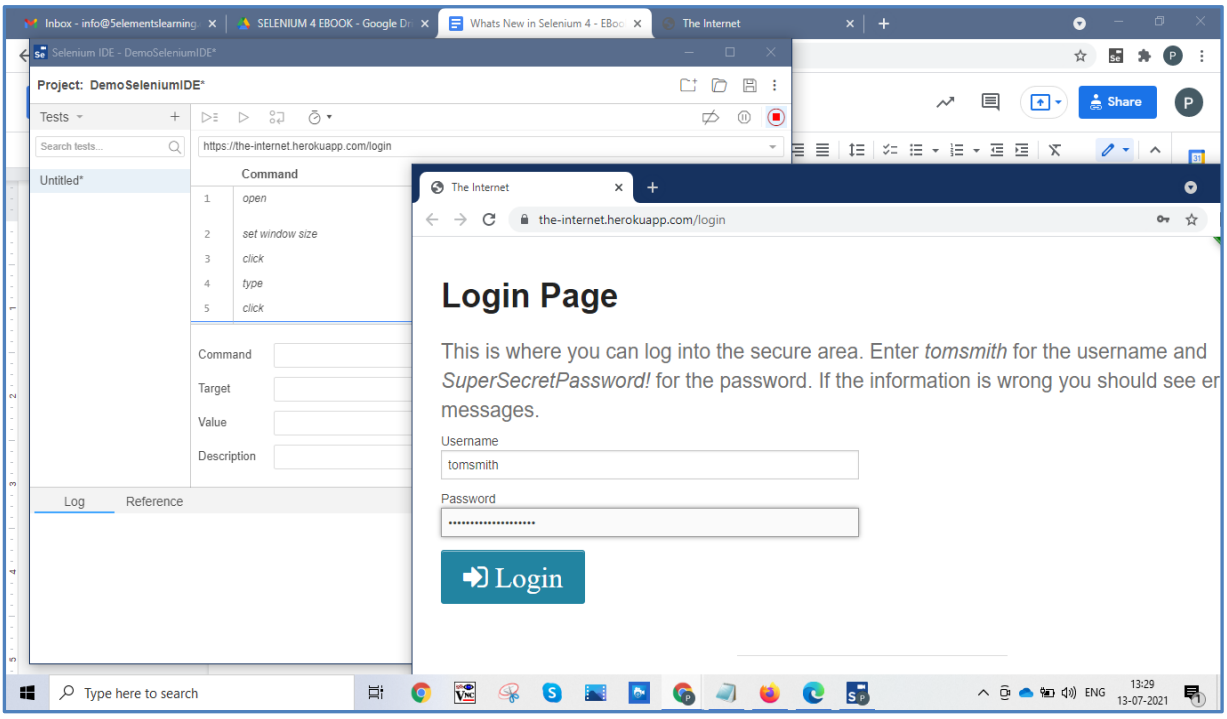


Scenario

a. Open chrome and open Selenium ide
b. Provide the url - https://the-internet.herokuapp.com/login and start recording. This will launch a new browser window with Selenium IDE recording the actions on it -



c. As we press record, a new page will get launched and whatever actions we do on that page, they will be captured by the Selenium IDE. We are going to record the steps to login to the application.

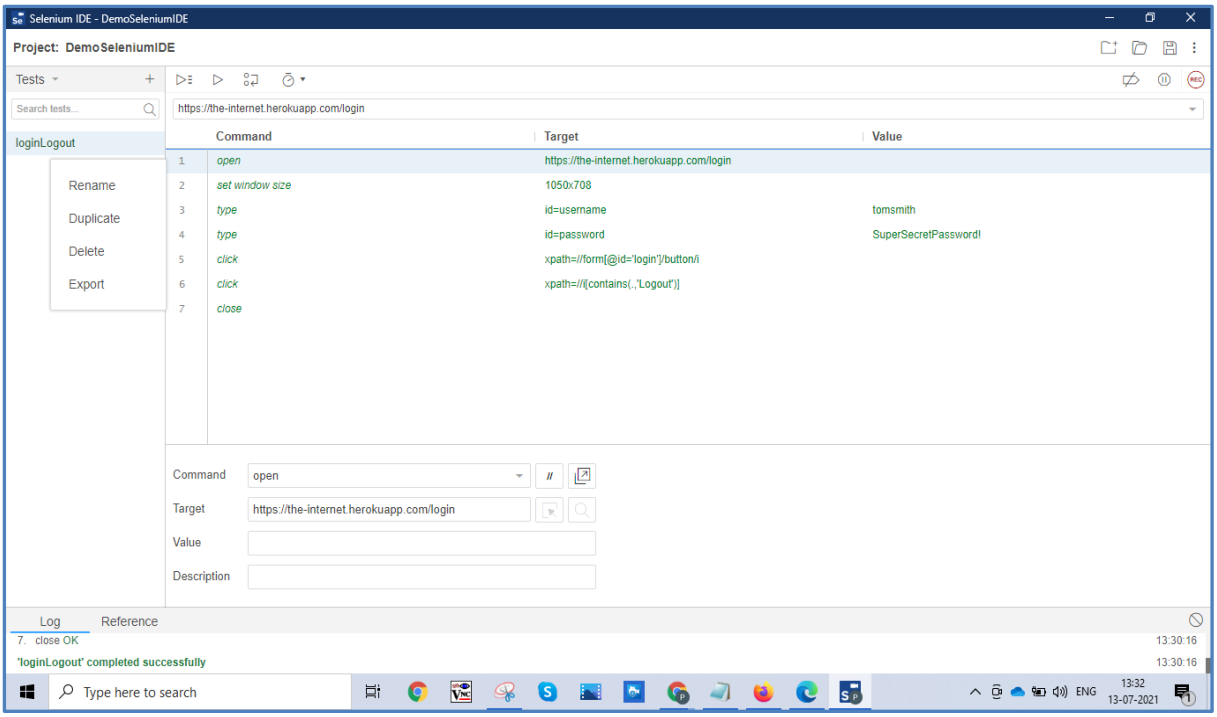


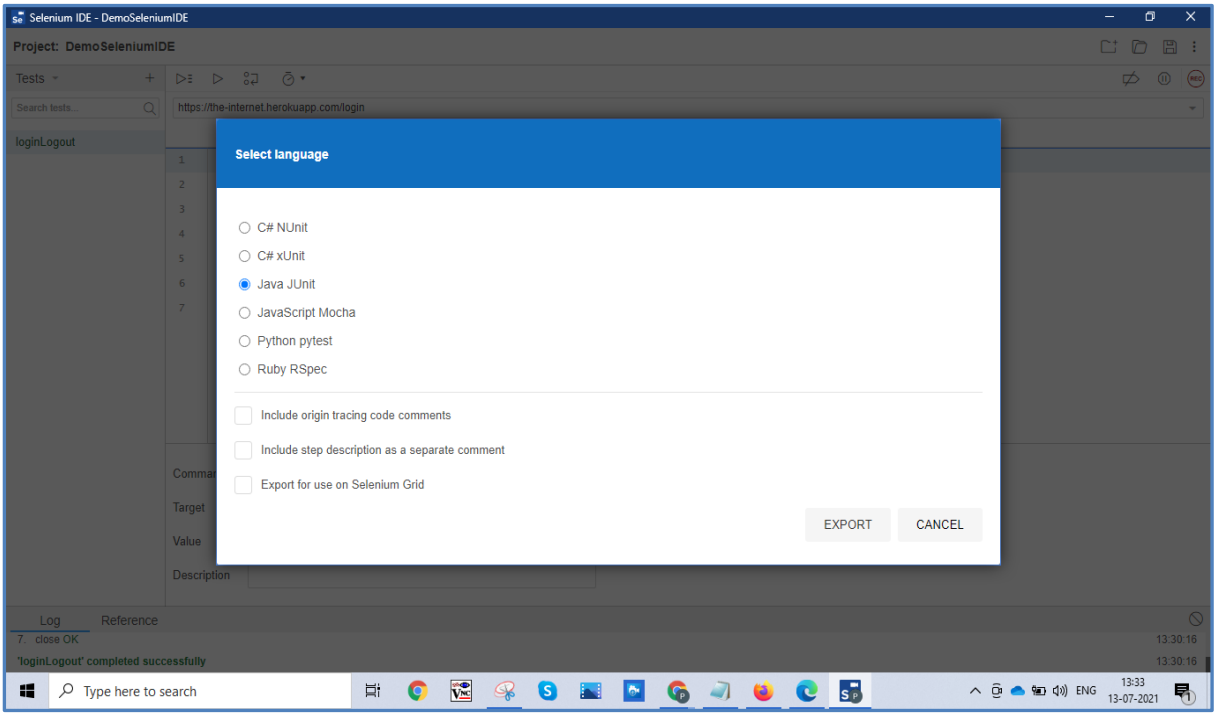
d. The recorded script looks as follows

https://the-internet.herokuapp.com/login

	Command	Target	Value
1	open	https://the-internet.herokuapp.com/login	
2	set window size	1050x708	
3	type	id=username	tomsmith
4	type	id=password	SuperSecretPassword!
5	click	xpath=//form[@id='login']/button/i	
6	click	xpath=//i[contains(.,'Logout')]	
7	close		

e. We can now export it in the java language bindings. Click on the test file and select the export option from the menu item. Select the Java Junit binding from the window which opens.





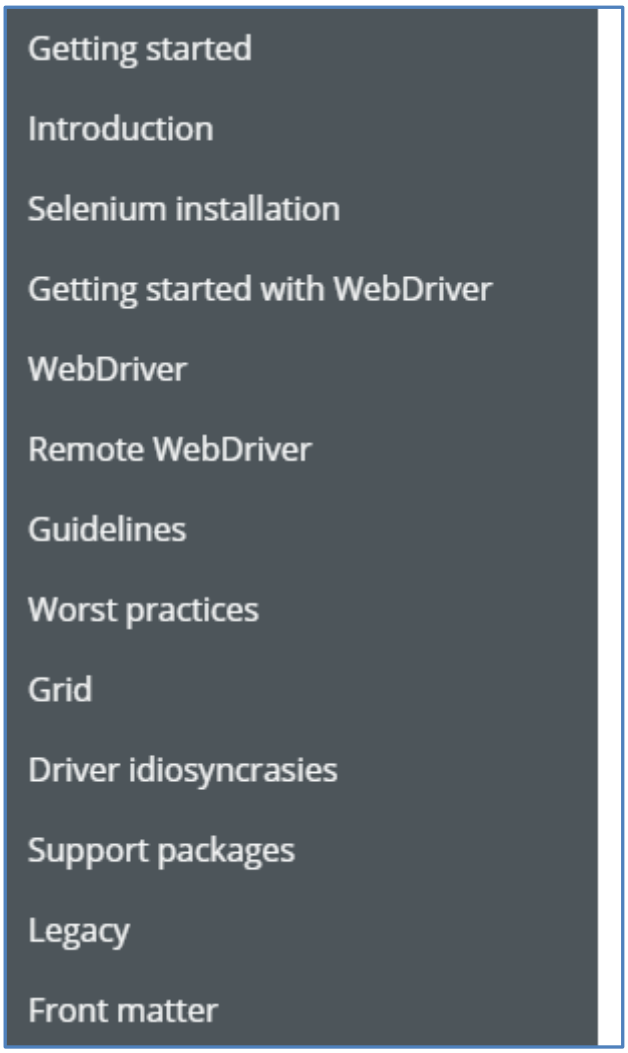
The exported java code looks as follows

```
// Generated by Selenium IDE
import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.core.IsNot.not;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.remote.RemoteWebDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.Dimension;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.Alert;
import org.openqa.selenium.Keys;
import java.util.*;
import java.net.MalformedURLException;
import java.net.URL;
public class LoginLogoutTest {
  private WebDriver driver;
  private Map<String, Object> vars;
  JavascriptExecutor js;
  @Before
  public void setUp() {
    driver = new ChromeDriver();
       js = (JavascriptExecutor) driver;
       vars = new HashMap<String, Object>();
  }
  @After
  public void tearDown() {
       driver.quit();
  }
  @Test
  public void loginLogout() {
    driver.get("https://the-internet.herokuapp.com/login");
    driver.manage().window().setSize(new Dimension(1050, 708));
    driver.findElement(By.id("username")).sendKeys("tomsmith");
    driver.findElement(By.id("password")).sendKeys("SuperSecretPassword!");
    driver.findElement(By.xpath("//form[@id='login']/button/i")).click();
    driver.findElement(By.xpath("//i[contains(.,'Logout')]")).click();
    driver.close();
  }
}
```

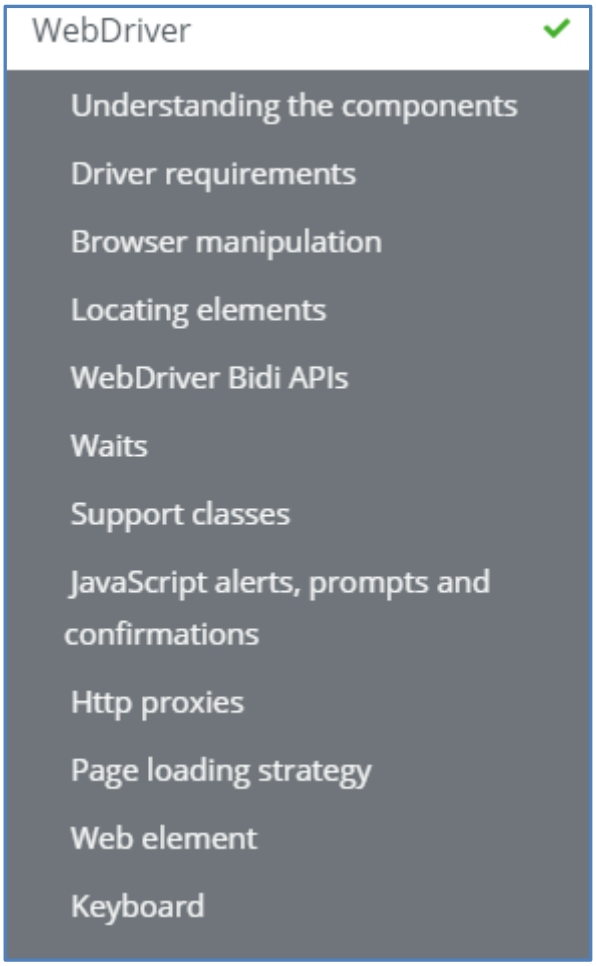
To know more about the other rich feature set of Selenium IDE you can follow the link -https://applitools.com/blog/selenium-ide-with-dave-haeffner/ and read more content in here specific to Selenium IDE.

Chapter 11: Improved Selenium Documentation

The Selenium Documentation available for the Selenium project has improved significantly. It now has a better UI with a detailed list of topics in an organized manner. The link for the Selenium documentation is here- https://www.selenium.dev/documentation/en/. The sections are segregated and look as follows:

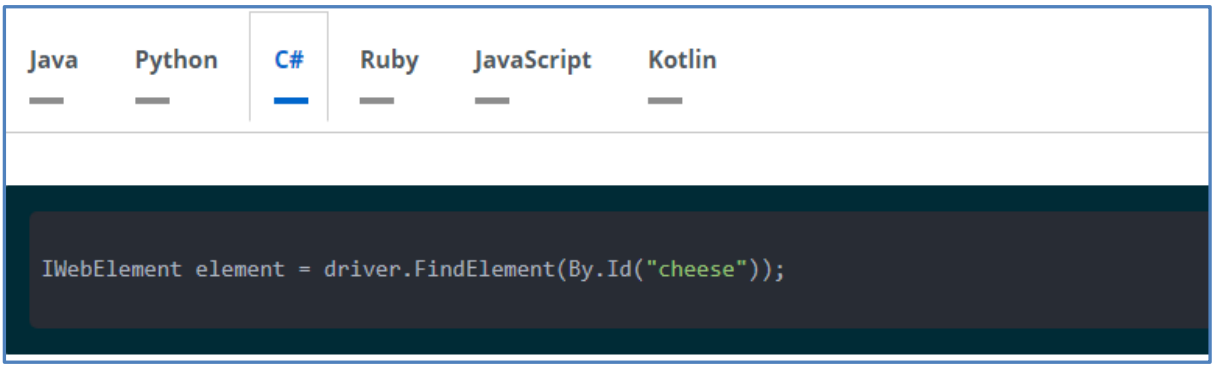


For selenium webdriver, we need to click on the WebDriver section. This contains various sections which helps us to understand the working and functioning of the webdriver component of Selenium. On clicking webdriver we see the following sections:

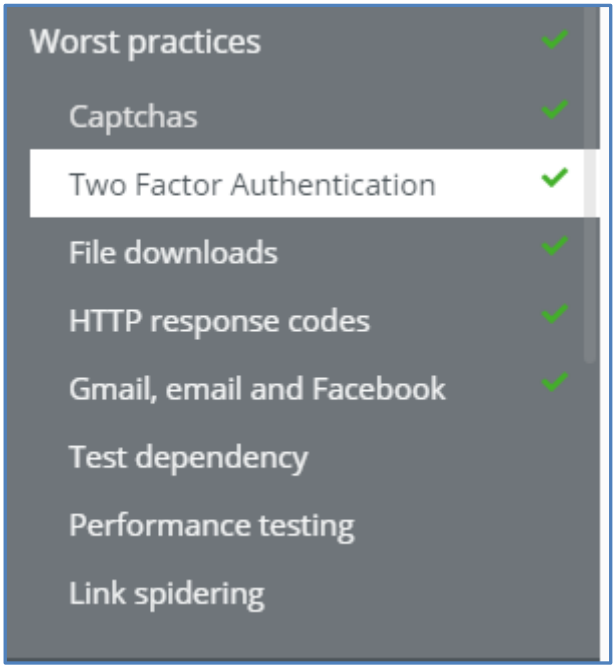


Mostly all the code examples of Selenium are covered in the supported programming language bindings. So, we will find the code example available in Java, Python, C#, JavaScript, Ruby and Kotlin. Let us look at the locating element section from above and see one of the scenarios explained as in the Selenium documentation. We pick the example of Locating one Element

We can see from the above image the Java code displayed as that tab is highlighted. If we highlight the tab for let's say C#, we will see the same code example, but this time implemented in C# programming language.



An interesting section in the documentation is the Worst Practices. It lists down the various scenarios where automation using Selenium is not a good practice. For example, using Selenium for performance testing, spidering activities is not recommended. It also lists down various scenarios about how to proceed with automation using Selenium, if we encounter them, like captcha, 2 factor authentication. The section looks as follows-



If someone wants to contribute to the Selenium documentation, the process to do it is easy and requires a GitHub account. More details on it are available here.
https://www.selenium.dev/documentation/en/contributing/
Selenium Documentation is vast and the most authentic help available on Selenium.

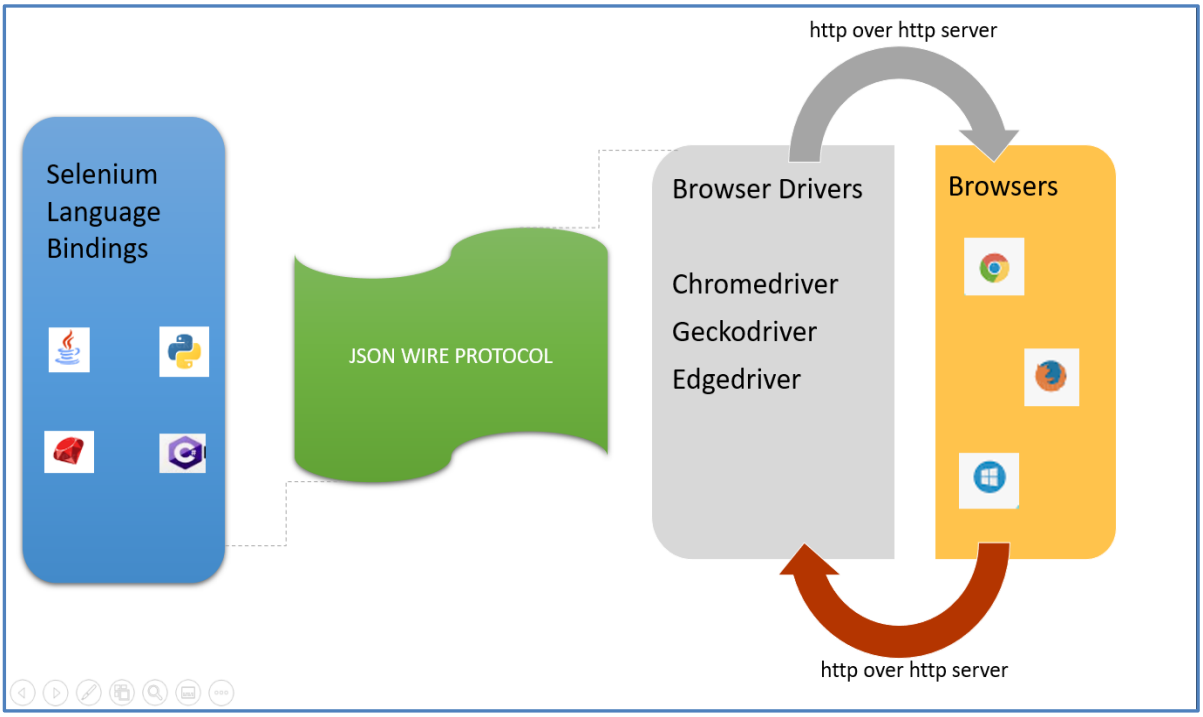
Chapter 12: Protocol Change from JSON To W3C

Last but not the least, one of the major changes which will be implemented fully from Selenium 4, is the change of the communication protocol from JSON to W3C in Selenium.

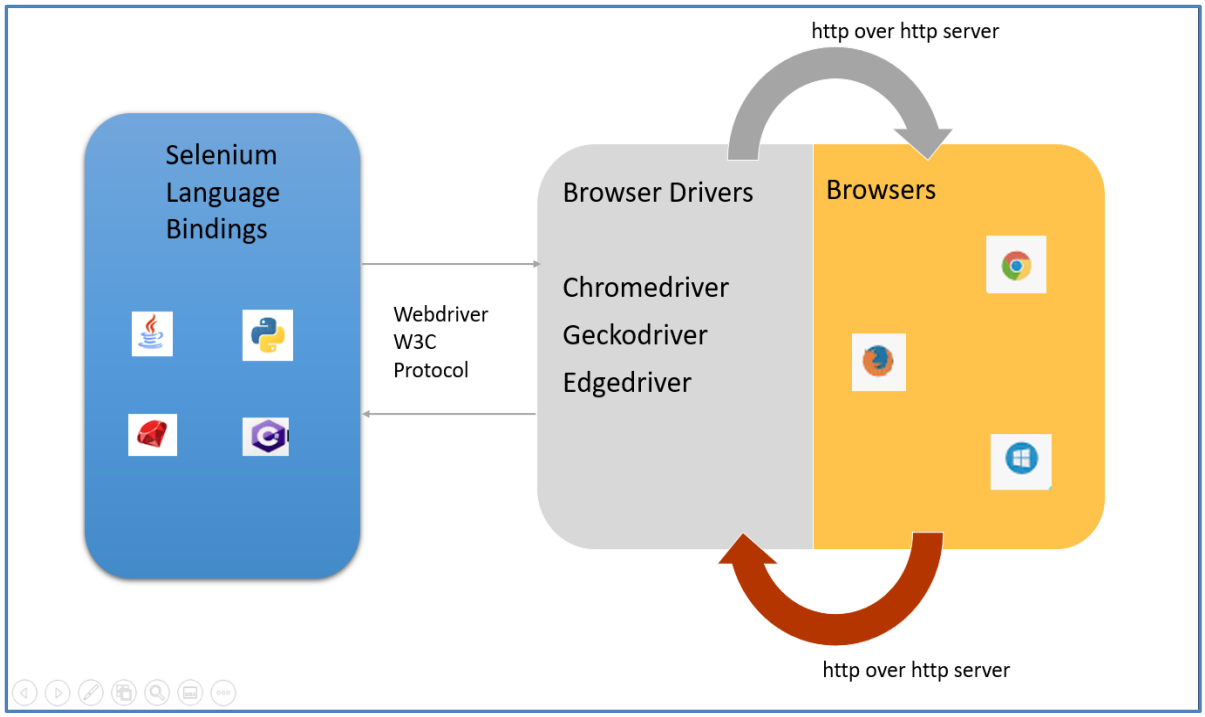
The earlier version of Selenium used the JSON Wire Protocol as the communication protocol between the selenium programming language bindings and the browser driver executables. In the latest version of Selenium 4, the protocol for communication will be WebDriver W3C. The creation of this protocol started a few years ago and is now complete and an acceptable standard by the W3C consortium. More details are available on the below URL.

https://w3c.GitHub.io/webdriver/

When we execute a script which uses Selenium to automate the browser, the browser and Selenium must talk to each other. This communication between the selenium language bindings, the browser driver executable, and the browser itself is to be governed by a set of rules. This set of rules which govern the communication among different interfaces is known as a protocol. In the earlier version of selenium, the communication between the selenium language implementation, and the browser driver was governed by the Json Wire Protocol. This means that API encoding, and decoding was involved in the communication between the language binding and the browser river using the Json Wire Protocol. The following image shows this -



But with the selenium 4, we no longer need the encoding and decoding of the API, as selenium 4 has moved the communication protocol to WebDriver W3C. So, the now the mechanism looks as per image below.



Because of the above change there would be some major advantages in the way selenium scripts communicate with the browsers

a. Reduce flakiness, as all are going to talk using the same rules, the scripts will be extremely stable.
b. Improved action class support, we will now be able to use more advance keyboard and mouse actions and have better control over the browsers
c. Speed, the script execution will be faster, as for each command there won't be any encoding and decoding happening using a separate protocol.

To see the protocol used during script execution, we created a simple script, where we launched a web application, and closed it. The following is the code for it

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

import io.GitHub.bonigarcia.wdm.WebDriverManager;
```

```
public class demoW3C {

	WebDriver driver;


  @BeforeMethod
  public void beforeMethod() {
		WebDriverManager.chromedriver().setup();
		    driver = new ChromeDriver();
  }

  @Test
  public void openApp() throws Exception{

driver.get("https://seleniumsummit21.agiletestingalliance.org/");
		Thread.sleep(1000);
  }

  @AfterMethod
  public void afterMethod() {
		driver.close();
  }

}
```

And we can see in the console output, the dialect information as W3C.

```
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
Jun 07, 2021 2:18:26 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
Jun 07, 2021 2:18:26 PM org.openqa.selenium.devtools.CdpVersionFinder findNearestMatch
INFO: Found exact CDP implementation for version 91

===============================================
```

Various resources were used to generate the above chapter, following are the links

a. http://www.selenium.dev
b. https://w3c.GitHub.io/webdriver/
c. https://medium.com/@juanba48/selenium-4-is-now-w3c-compliant-what-does-this-mean-ceb44de2d29b
d. https://saucelabs.com/selenium-4

Concluding remarks

We are aware that Selenium has and is undergoing massive changes. The book is an attempt to cover changes as per the latest beta-3 release of Selenium 4.

Please note that even before we completed the book, we have a new release beta-4 already in place.

We will try our best to keep the code relevant as Selenium moves ahead, we would need lot of support for this though. Hence, we would like to welcome everyone in the community to please let us know if the code breaks or something is amiss in the releases later than beta3. If you want to contribute to keep this book relevant, you are most welcome to do so. Please connect with us on LinkedIn showing your interest. We would be extremely glad to help take the conversation ahead. Any contributions made to this book or to the code will be given due credit.

We wish that the learning never stops.

About ATA

ATA is a global alliance of Technologists, QA, Agile and DevOps practitioners, professionals, and experts. ATA is driving community driven shared and collaborative learnings across multiple chapters in 24 countries.

ATA is a training and certification organization providing globally recognized Certification in the areas of Agile Testing, Kanban, DevOps, and technology/tools. For more details, please refer the following URL

https://www.agiletestingalliance.org/

Apart from this, Agile Testing Alliance is pioneering selenium learnings and certification exam called as **CP-SAT**. (CP-SAT is Globally recognized #1 certification program in Selenium).

CP-SAT stands for **Certified Professional - Selenium Automation Testing**. It is not only about Selenium but a host of other Selenium ecosystem technology stack like TestNG, JUnit, Json, Excel file reading, data driven testing etc. and everything that would be required for practically driving successful Selenium implementation. It is also the only certification exam which can be taken in any of the technology streams like Java, C# or Python. To know more about the same please visit the following URL.

https://cpsat.agiletestingalliance.org/

License Information

This book has been released as per the creatives common license below.

Attribution-Noncommercial 4.0 International (CC BY-NC 4.0)

Details can be found on the below URL
https://creativecommons.org/licenses/by-nc/4.0/

Reference screenshot from the above URL is shown below.