# Automation testing with SELENIUM (Web GUI testing)

Selenium is:

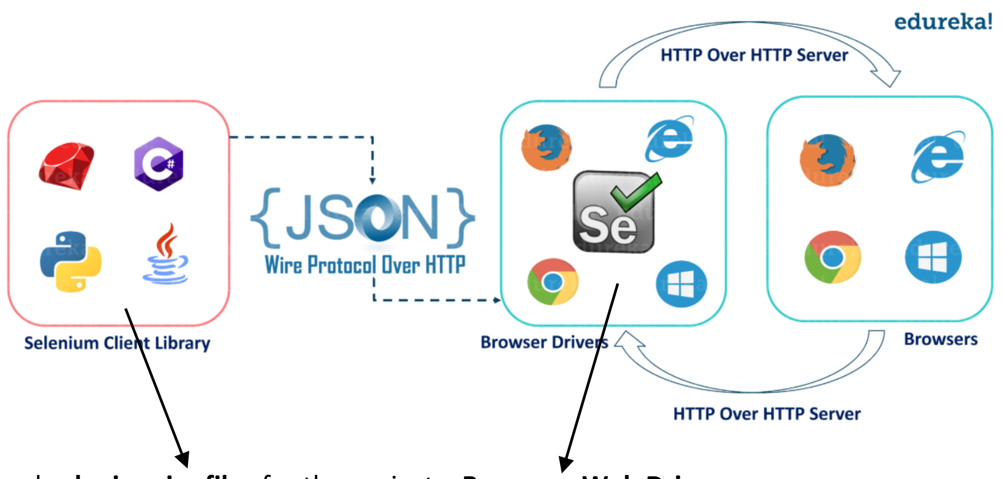
1. Selenium is an open-source automation testing tool, so that means you can just download selenium
2. It is exclusively for Web Based Applications
3. Selenium supports multiple browsers
4. Multiple platforms: Windows, Apple OS X, Linux
5. Can be coded in multiple languages Java, C#, Python, PHP, Ruby

Selenium is a robust set of different software tools each with a different approach to supporting test automation for web-based applications. It can be run in three variations:

- Selenium WebDriver
- Selenium Grid
- Selenium IDE

Official website: seleniumhq.org

Selenium WebDriver Framework Architecture Diagram:



Download **selenium jar files** for the project + **Browsers Web Driver**

https://www.selenium.dev/downloads/

Primer:

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

public class Third {

    public static void main(String[] args) {

           // System Property for Chrome Driver
           System.setProperty("webdriver.chrome.driver", "D:\\ChromeDriver\\chromedriver.exe");

          // Instantiate a ChromeDriver class.
          WebDriver driver=new ChromeDriver();

          // Launch Website
          driver.navigate().to("http://www.javatpoint.com/");

          //Maximize the browser
          driver.manage().window().maximize();

          //Scroll down the webpage by 5000 pixels
          JavascriptExecutor js = (JavascriptExecutor)driver;
          js.executeScript("scrollBy(0, 5000)");

          // Click on the Search button
          driver.findElement(By.linkText("Core Java")).click();
    }
}
```

Locators supported by WebDriver:

- Id
- ClassName
- Name
- CSS
- XPath
- LinkText

## 7.2.3 Page object model (POM)

Page object model design pattern has nowadays become a very popular way of organizing the test framework because of its easy test maintenance and reducing the duplication of code.

```
Public class HomePage extends BasePage{
   @FindBy(id="xx_container")
   private WebElement makePostDialogLocator;

   @FindBy(css="._fjdk")
   private WebElement nextButtonLocator;

   @FindBy(css="a[data-testid='left_nav_item']")
   private webElement welcomePageButtonLocator;
}
```

Figure 53: Example of POM

As presented in Figure 53, we have first identified the locators and defined it on the top after the class. In this way, we can achieve the readability of test scripts and we can easily identify locators and change them if needed at only one place.

Page Object Model approach is writing all the functionalities / reusable components of a page that we want to automate in a separate class. Say now if we consider three pages as Home page, Login page, Welcome page, etc.

For the above pages, we will create classes as *HomePage.class*, *LoginPage.class* and *WelcomePage.class*. In each class, we will identify and write reusable locators and methods, which are specific to a page.

```
com.matf.pages
    BasePage.java
    HomePage.java
    LoginPage.java
    WelcomePage.java
```

Figure 54: Class organization

Besides given pages, there is a *BasePage.class* — this is the parent class of all pages' classes. It contains the locators and methods that are mutual for all inheriting pages. In general, *BasePage.class* is not required to exist, and there can be more than one parent class.

On the first page - a Login page, after a user successfully logs in, it navigates to a respective page — Welcome page. Whenever we are navigating to another page, we need to return that page object.

The Page Object model provides the following advantages.

- There is a clean separation between test code and page specific code such as locators and layout.
- There is a single repository for the services or operations offered by the page rather than having these services scattered throughout the tests.

In both cases, this allows any modifications required due to UI changes to ail be made in one place. Useful information on this technique can be found on numerous blogs as this design pattern is widely used. Many have written on this design pattern and can provide useful tips beyond the scope of this script.

This way, the test is operating with the methods of page classes, making the test easily readable:

```
Public void testLoginToFb{
   # Open FB login page
   LoginPage loginPage = new LoginPage(driver).open();

   # Login
   WelcomePage welcomePage = loginPage.login(email, password);

   # Verify home page
   Assert welcomePage.isWelcomeMessageExpected();

   # Logout
   loginPage = welcomePage.logout();

   # Verify FB login page
   Assert loginPage.isLoginPageDisplayed();
}
```

Figure 55. Example of test

### 7.2.4 Selenium WebDriver commands

In the following section, we will give a few examples of basic *WebDriver* and *WebElement* methods, as well as how they are used.

- **Fetching a Page**

  ```
  driver.get(http://imi.pmf.kg.ac.rs)
  ```

- **Getting page url and title**

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

- **Defining web elements**

  The following elements can be defined in multiple ways:

  ```
  <input id="username" type="text" class="abcd" name="login" />
  <a class="issue-link" href="jira/test/2021" id="key">IMI</a>
  driver.findElement(By.id("username"));
  driver.findElement(By.name("login));
  driver.findElement(By.class("abcd"));
  driver.findElement(By.xpath("//input[@id='elementID']"));
  driver.findElement(By.cssSelector("input[id='elementID']"));
  driver.findElement(By.linkText("some text"));
  ```

- **Locating UI Elements (WebElements)**

  Locator is a command that tells Selenium IDE which GUI elements it needs to operate on. Most of the selenium webdriver commands need an element locator. There are several ways how GUI elements can be located, and which one is the best varies from element to element since it depends on multiple conditions (is the element unique, dynamic, which attributes are defined, etc.).
  - ByID
  - ByCss
  - By XPath
  - By Class Name
  - By Tag Name
  - ByName
  - By Link Text

- **Getting element -** find a webElement on a webpage

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

- **Getting element text values**

  ```
  element.getText();
  ```

- **Click on GUI element**

  ```
  element.click()
  ```

- **Type on GUI element**

  Method *sendKeys()* is used for keyboard manipulation whether is just text typing or pressing Enter, Shift etc.

  ```
  element.sendKeys(“type something”);
  element.sendKeys(Keys.ENTER);
  ```

- **Navigation: History and Location**

  ```
  driver.navigate().to(“http://example.com”);
  driver.navigate().forward();
  driver.navigate().backward();
  ```

- **Closing browser**

  ```
  driver.close();
  ```

  Method closes the web browser window that the user is currently working on (window that is being currently accessed by the WebDriver).

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

  Method closes all opened windows (opened by WebDriver).

- **Drag and Drop**

  ```
  (new Actions(driver)).dragAndDrop(element, target).perform();
  ```

### 7.2.5 Selenium Waits

It is not an unusual situation for webdriver to try to manipulate web element before it is visible on a page, or while it is disabled, not clickable, etc. As a result, some exceptions will be thrown. To avoid this situation, certain waits can be added.

Selenium webdriver provides two types of waits, **implicit wait** and **explicit wait**.

The *implicit wait* will pause webdriver for a defined amount of time before trying to continue the execution.

```
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
```

In the case of *explicit wait* maximum wait time is defined and webdriver will wait until some defined action occurs (during that period) before continuing with the test.

In this case, webdriver will wait until the given element is visible on the page before continuing with test execution but for a maximum of 30 seconds.

```
WebDriverWait wait = new WebDriverWait(driver, 30);
wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("
      //input[@id=‘name’!]")
));
```

### 7.2.6 Selenium Exceptions

Exceptions are events due to which the program ends abruptly without giving expected output.

Selenium has its own set of exceptions. All runtime exception classes in Selenium WebDriver come under the superclass *WebDriverException*.

Though there are many Exception classes under *WebDriverException*, the ones below are commonly seen:

- **WebDriverException**: Exception comes when a code is unable to initialize WebDriver.
- **NoSuchElementException**: Accessing an element which is not available in DOM.
- **ElementNotVisibleException**: Element exists in DOM but is not visible within the page.
- **StaleElementReferenceException**: Initial reference to the element is lost.
- **NoSuchAttributeException**: Trying to get attribute value but the attribute is not available in DOM.

### 7.2.7 DB testing

Changes on the UI must reflect the state in the application's DB. Often test requirements if to check whether the value on the UI is the same as in the database. Since Selenium is a UI automation tool WebDriver alone is ineligible to perform database testing so an additional tool must be used. This can be done using Java Database Connectivity API (JDBC).

JDBC is a SQL level API that allows:

- Creating a connection with the database.
- Executing queries and update statements to extract or fetch data.
- Using and manipulating the data extracted from the database in the form of the result set. (The result set is a collection of data organized in the rows and columns).
- Disconnecting the connection.

JBDC can used to interact with different DB types MySql, Oracle, Postgres...

### 7.2.8 Selenium grid

When there is a need to execute the same set of tests on a different set of OS/browsers, with an option to be executed in parallel, the selenium grid is one of the options.

Selenium grid is a network of hub & nodes. Each node registers to the hub with a certain configuration and hub is aware of the browsers available on the node. When a request comes to the hub for a specific browser (with desired capabilities object), the hub, if found a match for the requested browser, redirects the call to that particular grid node and then a session is established bi-directionally and execution starts.

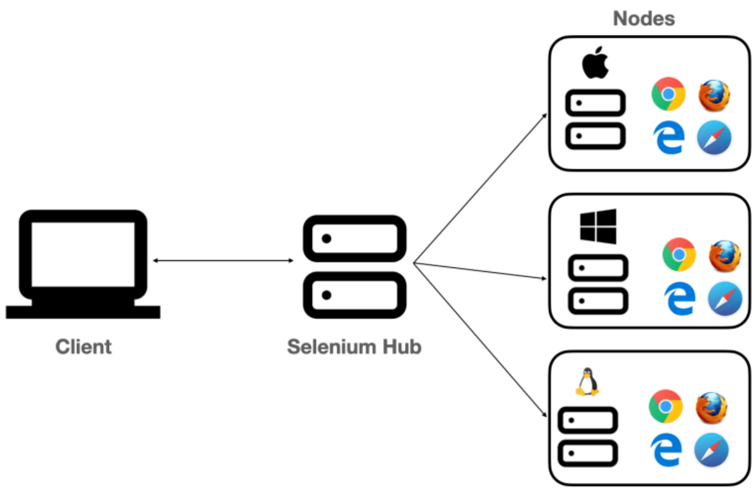


Figure 56: Selenium Grid

### 7.2.9 TestNG

TestNG is a testing framework designed to simplify a broad range of testing needs. Its main usage is to represent the entry and exit point of test automation. Using its configuration file, tests can be triggered, and test results can be generated in a report file. it also represents an integration point between Ci tools (e.g. Jenkins) and test code. More about TestNG configurations can be found on [http://testng.org/doc/documentation-main.html](http://testng.org/doc/documentation-main.html)

**Usage**

TestNG covers wide variety of testing types:

- unit testing
- integration testing
- functional testing
- end-to-end testing

**Configuration**

TestNG configuration is stored in file **testng.xml**, which has a structure like:

```
<?xml version="1.0" encodings"UTF-8"?>
<!DOCTYPE suite SYSTEM "“http://testng.org/testng-1.@.dtd">
<suite name="Suite 1" verbose="1" parallel="classes" thread-count="3">
   <test name="Nopackage” >
       <parameter name="email” value="matf.automation@gmail.com"></parameter>
       <parameter name="password"“ value="Matf123/@#"></parameter>
       <classes>
           <class name="NoPackageTest"/>
       </classes>
   </test>
   <test name="Regression suite 2" preserve-order="true">
       <classes>
           <class name="com,first.example.demoOne"/>
           <class name="com.first.example.demoTwo" />
```

```
            <class name="com.first.example.demoThree" />
        </classes>
    </test>
</suite>
```

**Annotations**

TestNG looks up for the annotations in the code and based on that defines what is to be executed and how. Some of the basic ones are:

- @Test
- @BeforeSuite/@AfterSuite
- @BeforeTest/@AfterTest
- @BeforeGroups/@AfterGroups
- @BeforeClass/@AfterClass
- @BeforeMethod/@AfterMethod
- @Parameters
- @DataProvider

@Test annotation defines which class/method is to be executed and counted as a test. @Before/After annotations define which methods are executed prior to the test (usually as test preparation) and after the test (usually the test cleanup).
@Parameters annotation defines which parameters are read from the XML file and used as local variables.
@DataProvider annotation defines a method that supplies data for the test — using data provider, the same test will be executed multiple times using different sets of data.

**Annotation attributes**

Every testing annotation has its own attributes that can define it closely. Some of the attributes are the same for multiple annotations.

Most common attributes are:

- **alwaysRun** - If set to true, this method will be run even if one or more methods invoked previously failed or was skipped. Used for @Before/After and @Test annotation
- **enabled** - Whether this class/method is enabled or not.
- **timeOut** - The maximum number of milliseconds test should take.
- **dependsOnMethods** - The list of methods this method depends on.
- **priority** - The priority for test method execution order.
- **parallel** - If set to true, tests generated using this data provider are run in parallel. Default value is false.

### Test execution order

Test methods execution can be invoked in a certain order. Tests can be ordered by:

- priority
- depends on other methods
- without any defined order

If tests are about to be executed by priority it means they have priority attribute (priority = 1, priority= 2...), Lower priorities will be scheduled first.

One test can be dependent on the other test(s). Test annotation attribute *dependsOnMethods = "some_test"* will mark that test dependent on test named *"some_test"*. In that case it will be executed only after the "some_test" is successfully completed. Otherwise, it will be marked as skipped.

If not priority or dependency is defined, then tests will be executed in alphabetical order.

### Parallelism

TestNG can be instructed to run test methods in separate threads (in parallel). It can be achieved in several ways:

- From **suite.xml file** by setting attributed **thread-count** and **parallel** for suite tag. Thread-count is several threads to be used for tests execution and parallel defines what should be execute in parallel (methods, classes, or tests).
- Adding parallel=True attribute to @DataProvider annotation.
- By defining @Test annotation attributes *threadPoolSize* and *invocationCount*.

### Assertions

Assertions are used for validating a test case. The assertion is met if the actual test result matches with that of the expected result. There are two types of assertions.

- **Hard** - usually throw an Assertion Error whenever an assertion condition has not been met. The test case will be immediately marked as Failed when a hard assertion condition fails.
- **Soft** - continues with the next step of the test execution even if the assertion condition is not met.

Exceptions are not thrown unless it is asked for.

Some of the assertions:

- **assertEquals**(object1, object2, "Message on failure")
- **assertNotEquals**(objecti, object2, "Message on failure")
- **assertTrue**(boolean, "Message on failure")
- **assertFalse**(boolean, "Message on failure")
- **fail**("Message on failure") — purposely fails test case

## 7.2.10 Test automation framework diagram

Besides Test Suite Driver (in e.g., Java), Selenium and TestNG, typical test automation framework for testing web applications contains Function Library (helper and utility methods) and Test Data Files (different sets of values with which tests are called).

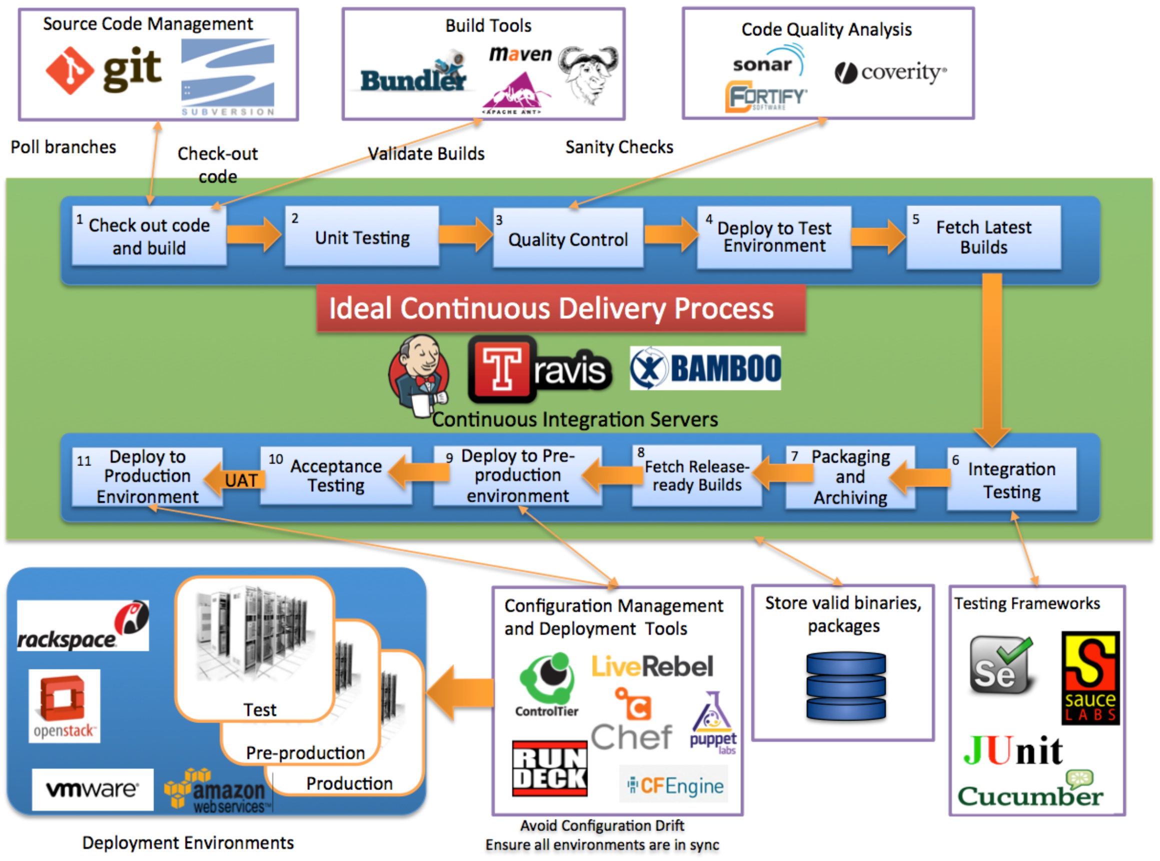

Source Code Management
Build Tools
Code Quality Analysis
Poll branches
Check-out code
Validate Builds
Sanity Checks
1 Check out code and build
2 Unit Testing
3 Quality Control
4 Deploy to Test Environment
5 Fetch Latest Builds
Ideal Continuous Delivery Process
Continuous Integration Servers
11 Deploy to Production Environment
UAT
10 Acceptance Testing
9 Deploy to Pre-production environment
8 Fetch Release-ready Builds
7 Packaging and Archiving
6 Integration Testing
Test
Pre-production
Production
Deployment Environments
Configuration Management and Deployment Tools
Avoid Configuration Drift
Ensure all environments are in sync
Store valid binaries, packages
Testing Frameworks

Figure 57: Test automation framework diagram