Automation Suite Manual

Author: Ratheesh C

Table of Contents

1. Introduction

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

1.1. Selenium controls web browsers

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

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

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

1.2. One interface to rule them all

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

2. Selenium installation

Selenium setup is quite different from the setup of other commercial tools. To use Selenium in an automation project you need to install the language bindings libraries for your language of choice. In addition, you will need WebDriver binaries for the browsers you need to automate and run the test.

2.1. Selenium bindings

First you need to install the Selenium bindings for your automation project. The installation process for libraries depends on the language you choose to use.

Four basic steps of Selenium WebDriver installation process

1. Download and Install Java 8 or higher version
2. Download and configure Eclipse or any Java IDE
3. Download Selenium WebDriver Java Client
4. Configure Selenium WebDriver

After installing the Java Environment Kit (JDK), a ***JAVA_HOME environment variable*** must be set to point to the JDK installation directory. So it is needed to set JAVA_HOME variable and PATH variable to the installed JDK bin directory.

2.2. Adding Executables to PATH

Most of the drivers require an extra executable for Selenium to communicate with the browser. It can be manually specified where the executable driver is located before starting WebDriver, but this can make your tests less portable as the executables will need to be in the same place on every machine, or include the executable within your test code repository.

By adding a folder containing WebDriver's binaries to your system's path, Selenium will be able to locate the additional binaries without requiring your test code to locate the exact location of the driver.

Note: Download the WebDriver binary supported by your browser and place it in the System PATH. To drive Chrome or Chromium, download chrome driver and put it in a folder that is on the system's path.

Edge is Microsoft's latest browser, which is included with Windows 10 and Server 2016. Updates to the Edge are bundled with major Windows updates, it is needed to download a binary which matches the build number of your currently installed build of Windows. The Edge Developer site contains links to all the available binaries.

2.3. Installation of selenium using maven

Installation of Selenium libraries for Java can be done using Maven also. Add the selenium-java dependency in your project pom.xml:

```
<dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>3.12.0</version>
</dependency>
```

The selenium-java dependency supports running automation project with all Selenium supported browsers. If needed to run tests only in a specific browser, the dependency can be added to the dependency for that browser in the pom.xml file. If you want to run tests only in Chrome, you should add the following dependency in pom.xml:

2.4. Installing Standalone server

If you plan to use Grid then you should download the selenium-server-standalone JAR file. All the components are available via selenium-server. The standalone JAR contains everything, including the remote Selenium server and the client-side bindings. This means that if you use the selenium-server-standalone jar in your project, you do not have to add selenium-java or a browser specific jar.

2.5. Driver requirements

WebDriver drives the browser using the browser's built-in support for automation, although not all browsers have official support for remote control. WebDriver's aim is to emulate a real user's interaction with the browser as closely as possible. Even though all the drivers share a single user-facing interface for controlling the browser, they have slightly different ways of setting up browser sessions. Since many of the driver implementations are provided by third parties, they are not included in the standard Selenium distribution.

3. Getting started with selenium

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

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

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

4. WebDriver

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

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

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

5. Test Suite

To test the behaviour of our software or project, we need to run multiple tests all at once. Moreover, running them manually one by one is not the way. This process of running multiple tests at once is called a test suite. Building a test suite using WebDriver will require you to understand and effectively use several different components. As with everything in software, different people use different terms for the same idea. Below is a breakdown of how terms are used in this description.

Terminologies:

- API(Application Programming Interface): This is the set of "commands" used to manipulate WebDriver.
- Library: A code module which contains the APIs and the code necessary to implement them. Libraries are specific to each language binding, eg .jar files for Java, .dll files for .NET, etc.

- Driver: Responsible for controlling the actual browser. Most drivers are created by the browser vendors themselves. Drivers are generally executable modules that run on the system with the browser itself, not on the system executing the test suite. (Although those may be the same system.) Drivers are also refered to as proxies.
- Framework: An additional library used as a support for WebDriver suites. Here we are using a framework called TestNG. The test framework is responsible for running and executing your WebDriver and related steps in your tests.

6. TestNG

TestNG is a Java-based open-source test automation framework. It covers a broader range of test categories: unit, functional, end-to-end, integration, etc. This framework is quite popular among developers and testers for test creation due to its useful features like grouping, dependence, prioritization, ease of using multiple annotations, etc. Another reason for its popularity is that it helps them organize tests in a structured way and enhances the scripts' maintainability and readability.

Features Of TestNG

- It provides a powerful and wide variety of annotations to support your test cases.
- It helps to perform parallel testing, dependent method testing.
- It offers the flexibility to run tests through multiple data sets through the TestNG.xml file or via the data-provider concept.
- Test cases can be grouped and prioritised as per needs.
- Provides access to HTML reports and can be customised through various plugins.
- It can be easily integrated with Eclipse, Maven etc.

7. Page object models

Page Object is a design pattern which has become popular in test automation for enhancing test maintenance and reducing code duplication. A page object is an object-oriented class that serves as an interface to a page of your AUT. The tests then use the methods of this page object class whenever they need to interact with the UI of that page. The benefit is that if the UI changes for the page, the tests themselves don't need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are in one place.

Advantages of the Page Object Design Pattern

- There is a clean separation between test code and page specific code such as locators (or their use if you're using a UI Map) and layout.
- There is a single repository for the services or operations offered by the page rather than having these services scattered throughout the tests.

In both the cases this allows any modifications required due to UI changes to all be made in one place.

8. Reporting

When we run this test, there are two separate sections in the Eclipse where these reports are visible.

8.1. Console Reports In TestNG

Console reports in TestNG are short and simple, which just denote the overall summary of the test.

8.2. TestNG Report Section In Eclipse

Alongside the console tab, the reports tab lies in Eclipse that generates a more in-depth view than what we had in the console.

8.2.1. Generate Emailable Report In TestNG

Emailable reports are generated in TestNG to let the user send their test reports to other team members. Emailable-reports do not require any extra work from the tester, and they are a part of overall test execution. To generate emailable reports, first, run the TestNG test class if you have not already.

Once we have run the test case, a new folder generates in the same directory with the name *test-output***.**

8.2.2. Generate automation report File In TestNG

Automation reports are a type of summary reports that one can transfer to other people in the team through any medium. Index reports, on the other hand, contains the index-like structure of different parts of the report, such as failed tests, test file, passed test, etc.

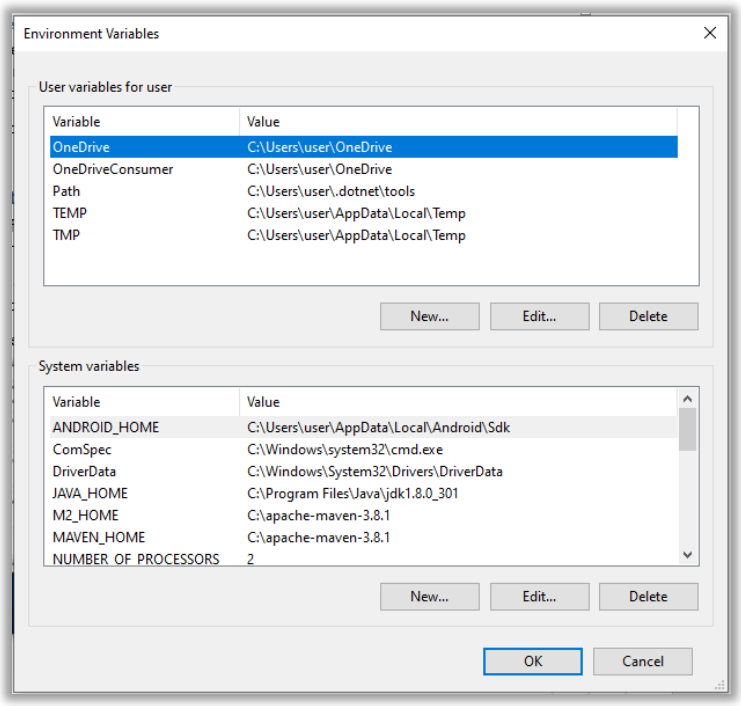
To open the *AutomationTestReport.html* file, locate it in the same test-output folder.

9. Steps followed to implement Automation Suite

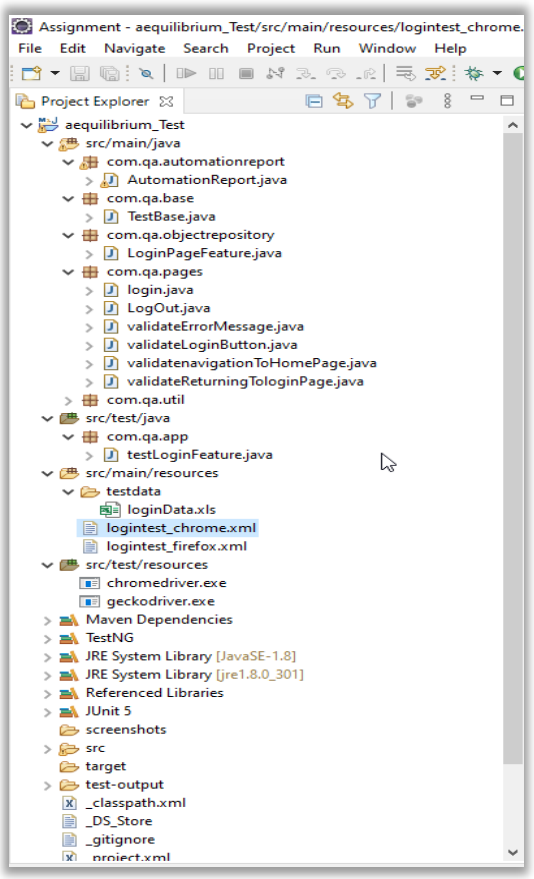
The Automation suite for this web application is implemented using Selenium WebDriver with TestNG framework Maven build automation tool.

Here in this context the jdk-1.8 is installed, IDE eclipse-jee-2020-06-R-win32-x86_64 and Selenium WebDriver 3.12.0 are configured.

JAVA_HOME variable and PATH variable to the installed JDK bin directory has been set.



The structure of framework is as follows:



The Maven build automation tool is used here.

This generate the following folders under the Project:

- src/main/java,
- src/test/java,
- src/main/resources,
- src/test/resources.

The folder src/main/java consists of various packages to include respective Java codes like base package, utility package, object repository package, automation report package and test case packages.

The folder src/test/java consists of package to include the respective testing code.

The test cases are arranged in the respective packages for easy identification and to run the specific test cases if needed.

Resources are kept in the /src/main/resources and the testing resources are placed in the /src/test/resources.

The testing code is created using the TestNg class where all the test cases are arranged.

As per the TestNG feature we can prioritize the test cases or run specific test cases.

Testing codes are imported with specific java codes. So, when a test case is executed, it will execute the exact java code and gives the result accordingly.

The driver can be included in the frame itself by adding into /src/test/resources.

The /src/main/resources consists of the test data file-loginData.xlx, and the important *testng.xml* file. Here there are two testng.xml files:

- logintest_chrome.xml and
- logintest_firefox.xml

These two xml files can be executed to run the test cases in respective browser.

TestNG.xml file is a configuration file that helps in organizing the tests. It allows testers to create and handle multiple test classes, define test suites and tests. It makes a tester's job easier by controlling the execution of tests by putting all the test cases together and run it under a single XML file.

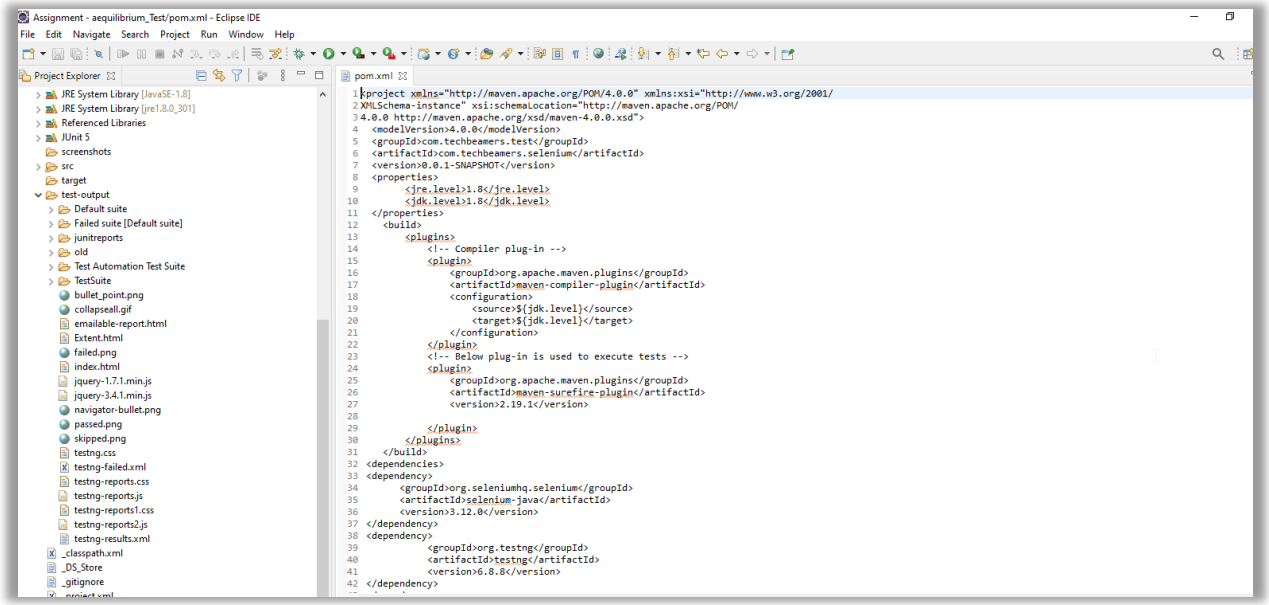
The *testng.xml* can be named based on the test and can include multiple testing xml file in a test suite. The *testng.xml* file must clearly specify the test class from the testing code.

POM stands for Project Object Model. It is the fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as *pom.xml*.

The POM contains information about the project and various configuration details used by the Maven to build the project(s).

POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal. Some of the configuration that can be specified in the POM are as follows:

- Project dependencies
- Plugins
- Goals
- Build profiles
- Project version
- Developers
- Mailing list
- It should be noted that there must be only a single POM file for a project.

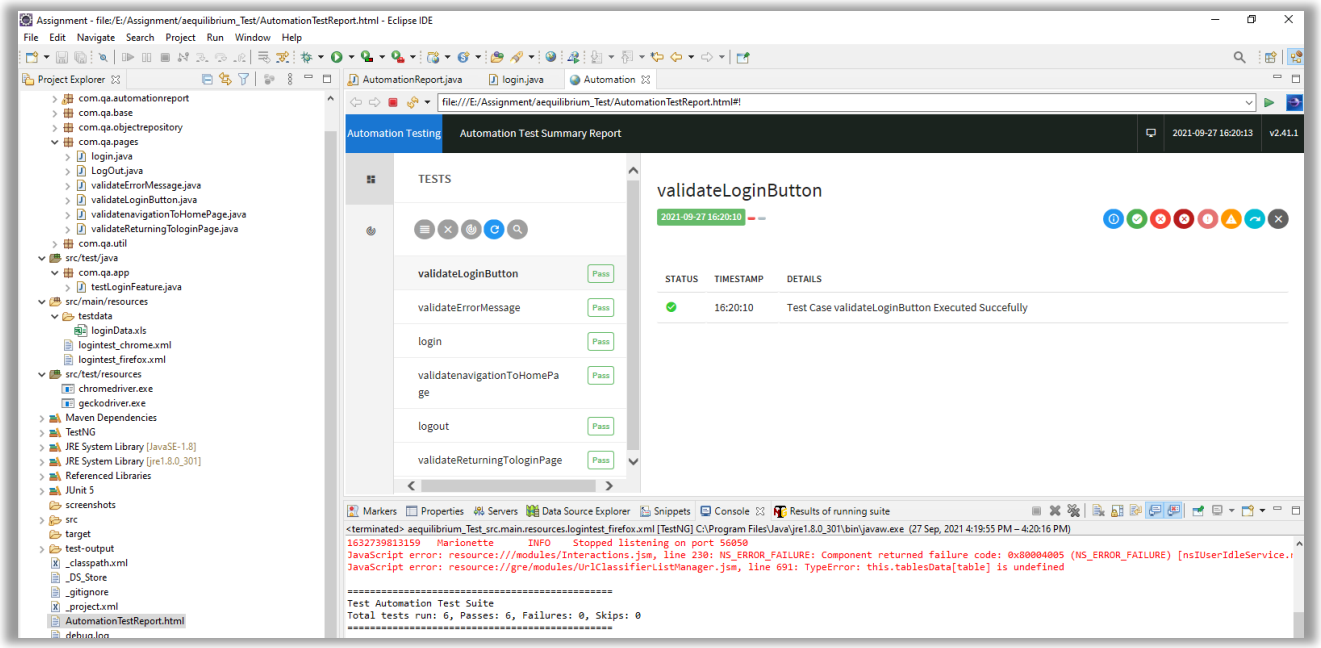


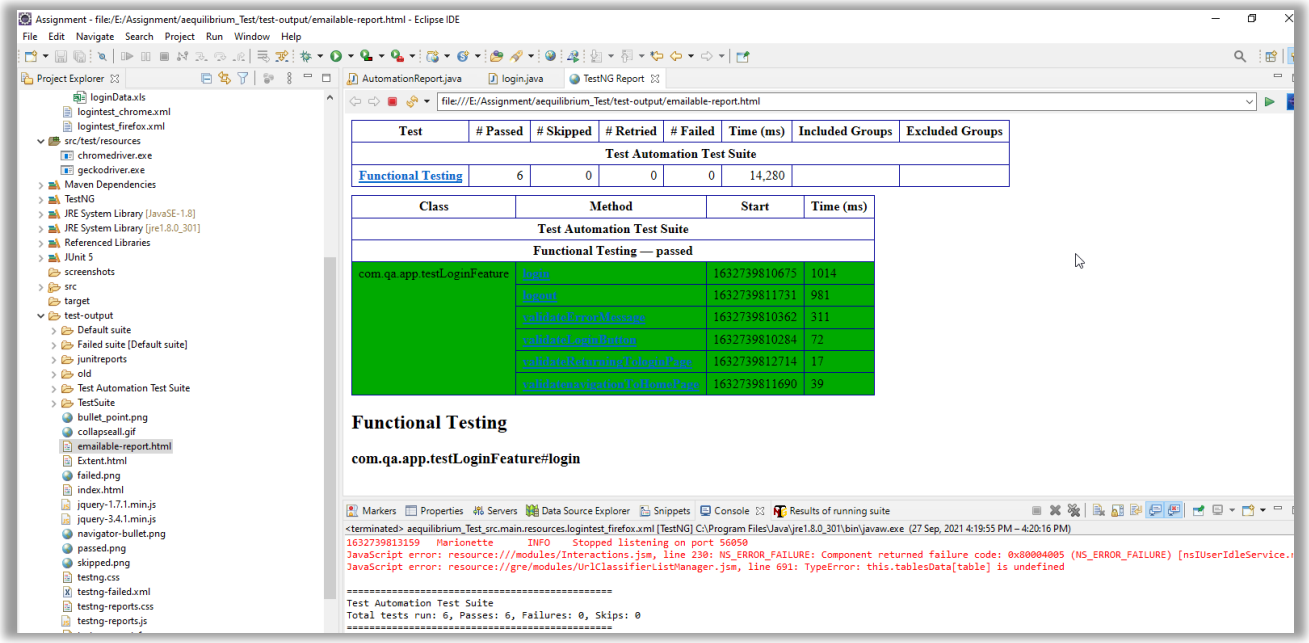
In the above *pom.xml*, the default project source folder structure, output directory, plug-ins required, repositories, reporting directory, which Maven will be using while executing the desired goals can be seen.

Maven *pom.xml* is also not required to be written manually. Maven will provide numerous archetype plugins to create projects and it will create the project structure and the *pom.xml.*

9.1. Automation test report

The java code for automation report generation is included in the *com.qa.automationreport* as *AutomationReport.java*. The various types of report generation classes are included there.





9.2. Screeshots

The screenshots of the failed test cases can be captured and saved either in any location or inside the framework itself. Here, the screenshots are saved inside the the folder ***screenshots*** whch is available in the framework. The screenshot function is written in the utility class. This function is used in the automation report generation code in such a way that the screenshot function will be executed when the test fails.

***