**Automation Framework:**

It is a standard rules & guidelines which should be followed by all the automation engineers while automating the test cases of the application.

* We should use automation framework to have consistency.
* Initially automation framework will be designed by Architect & then it is implemented by all the automation engineers.

**Based on the design, framework can be categorized as following types.**

1. Data Driven Framework
2. Method Driven Framework
3. Keyword Driven Framework
4. Module Driven Framework
5. Hybrid Automation Framework

Many organizations will create the automation framework & they will allow others to use the same. Below are few examples

1. Cucumber Framework (<http://cucumber.io/>) 🡪 BDD(Behaviour Driven Development)
2. Protractor Framework (<http://protractortest.org>) 🡪 Angular JS
3. Robo-Agile Framework (<http://roboagile.com>) 🡪 Script less agile framework
4. Robot Framework (<http://robotframework.org>) 🡪 Keyword driven

**Specification for Designing Framework:**

1. List of required software’s & Files
2. Standard Files & Folder structure
3. Generic Methods
4. Rules & Guidelines

**Design Framework:**

**List of required software’s & Files:**

1. JDK
2. Eclipse + TestNG Plugin
3. Selenium Jar (1) + POI (13) + TestNG Jar (4)
4. Browsers
5. Driver Executables
6. AUT (Application Under Test) 🡪 Application URL
7. MS-Office
8. AutoIT

**Standard Files & Folder Structure:**

1. **Create Workspace**:
2. Create a folder in the required location

**Ex:-** E:\\Projects\AutomationProjects

1. Open ‘Eclipse’, go to ‘File’ 🡪 Switch Workspace 🡪 Other, browse & select the above created folder & click ‘Launch’. It will restart the eclipse.
2. **Create Java Project:**
3. In eclipse go to File 🡪 New 🡪 Java Project
4. Specify the project name & click finish.

**Ex:-** CLGS

1. **Configure Maven:**

* Maven is a dependency tool.
* ‘Maven’ is used to automatically update the ‘Jar’ files. Dependency means ‘jar files’.
* We cannot update ‘Driver executables’ using ‘maven’.

**Steps to configure Maven & Adding dependencies:**

1. ‘Right’ click on java project (Ex:- CLGS)
2. Go to ‘Configure’.
3. Select ‘Convert to Maven project’.
4. Click on ‘Finish’ (It will generate ‘POM.xml’ file. [project Object Module])
5. Open POM.xml and Go to ‘Dependencies’ tab & click on ‘Add’.
6. Specify ‘Group ID’, ‘Artifact ID’ & ‘Version’ of dependencies then click ‘Ok’.
7. Save ‘POM.xml’ (Ctrl+S)

**Note:**

1. Click ‘Save’(ctrl+s) after adding each dependencies separately.
2. Add another dependency only after the finishing of first dependency is completely downloaded.

**List of Maven Dependencies:**

1. **Selenium Dependency:**

|  |
| --- |
| **URL:**  **https://mvnrepository.com/**  **Group ID:-** org.seleniumhq.selenium  **Artifact ID:-** selenium-java  **Version:-** LATEST |

1. **TestNG Dependency:**

|  |
| --- |
| [**URL:-**](URL:-) <http://testng.org/doc/maven.html>  **Group ID:-** org.testng  **Artifact ID:-** testng  **Version:-** 6.8 |

1. **POI Dependencies:**

|  |
| --- |
| <URL:-> <http://mvnrepository.com/artifact/org.apache.poi>  **Group ID:-** org.apache.poi  **Artifact ID:-** poi-ooxml  **Version:-** 3.16 |

|  |
| --- |
| **Group ID:-** org.apache.poi  **Artifact ID:-** poi  Version:- 3.16 |

|  |
| --- |
| **Group ID:-** org.apache.commons  **Artifact ID:-** commons-io  **Version:-** 1.3.2 |

**Note:**

1. When we use ‘Maven’ all the downloaded files will be listed in ‘**Maven Dependencies**’ folder in the project.
2. ‘Jar’ files downloaded by ‘Maven’ will be present in folder “**C:\Users\UserName\.m2\repository**” path.
3. **Associate TestNG:**
4. Right click on ‘Java project’. (Ex:- CLGS)
5. Go to ‘Build path’.
6. Select ‘Add Libraries’.
7. Select ‘TestNG’.
8. Click ‘Next’ & click ‘Finish’.
9. **Include Driver Executables:**
10. Right click on ‘Java project’. (Ex:- CLGS)
11. Go to ‘New 🡪 Folder
12. Specify the folder name as ‘Drivers’ & click on ‘Finish’.
13. Copy the required driver executable files (Ex:- ChromeDriver, GeckoDriver) & paste that inside the above folder.
14. **Include Excel Files:**
15. Right click on ‘Java project’. (Ex:- CLGS)
16. Go to ‘New 🡪 Folder
17. Specify the folder name as ‘Input\_Data’
18. Go to the location of ‘Input\_Data’ folder & create an excel file with name ‘Inputs’.
19. Refresh the ‘Java project’ in eclipse so that excel file is visible.
20. **Create folder for storing Screenshots:**
21. Right click on ‘Java project’. (Ex:- CLGS)
22. Specify the folder name as ‘Screensshot\_Images’ & click ‘Finish’.
23. **Associate Property File:**
24. Create a new folder with name ‘Propery\_Files’ inside the ‘Java project’. (Ex:- CLGS)
25. Right click on this folder, go to ‘New’ select ‘File’.
26. Specify the file name as ‘config.properties’ & click ‘Finish’.
27. **Create package:**
28. Right click on ‘src’ folder.
29. Go to ‘New 🡪 Package
30. Specify the package name & click finish.

**Note:** Create 3-packages as below

1. generic
2. page
3. script

**Automation Folder structure in Eclipse:**

|  |
| --- |
|  |

**Generic Methods:**

1. Create a class with name ‘**Excel\_Data**’ inside the ‘**generic**’ package & develop the methods in this class such that ‘**getValue()**’, ‘**getRowCount()**’ & ‘**getColumnCount()**’.

|  |
| --- |
|  |

1. In the ‘**generic**’ package create another class with name ‘**AutoUtils**’ & develop methods to take screenshot as ‘**getPhoto()**’ & method to get value from property file as ‘**getProperty()**’.

**Using Methods, Inheritance & Annotations:**

1. In automation framework any inputs which does not change their value should be made as constants & all the constants should be stored inside ‘**Interface**’ because interface variables are by default ‘**public static final**’.

In ‘**generic**’ package create an **interface** with name ‘**IAutoConstants**’ & store all the constant values.

|  |
| --- |
| **package** generic;  public interface **IAutoConstants**  **{**    ***String* CHROME\_KEY="**webdriver.chrome.driver**";**  ***String* CHROME\_VALUE="**./Drivers/chromedriver.exe**";**  ***String* GECKO\_KEY="**webdriver.gecko.driver**";**  ***String* GECKO\_VALUE="**./Drivers/geckodriver.exe**";**  ***String* EXCEL\_PATH="**./Input\_Data/Inputs.xlsx**";**  ***String* CONFIG\_PATH**="./Propery\_Files/config.properties";  ***String* IMAGE\_PATH**="./Screensshot\_Images/";  **}** |

1. In automation for every regression test case we develop respective ‘**Test’** class.
2. Every ‘Test’ class will have some common steps such as opening the application & closing the application which will consume lot of space, time & memory.
3. To avoid repetition of these steps we develop methods to ‘Open application’ & ‘Close application’ & to access same methods in all classes directly we use ‘**Inheritance**’.

To execute these methods automatically (without calling it) we use ‘**Annotations**’ (@BeforeMethod & @AfterMethod).

1. Create a class with name ‘**BaseTest**’ inside the ‘**generic**’ package & develop methods to ‘Open application’ & ‘Closing application’. Use TestNG annotations for these methods.

‘BaseTest’ class is called as ‘**Super class** (parent class)

1. The ‘Super’ class (i.e BaseTest) which has **‘@BeforeMethod’** & **‘@AfterMethod’** should be declared as ‘**abstract**’ class because it is incomplete without ‘**@Test**’ method. Incomplete means we cannot execute this class.

|  |
| --- |
| **package** generic;  **abstract** public class **BaseTest *implements* IAutoConstants**  {  **public** ***WebDriver*** driver;  **static**  **{**  **System.setProperty(***CHROME\_KEY*, *CHROME\_VALUE***);**  **System.setProperty(***GECKO\_KEY, GECKO\_VALUE***);**  }  **@BeforeMethod(alwaysRun=true)**  public void **openApp()**  **{**  *String* **appURL**=**AutoUtils.getProperty(***CONFIG\_PATH, "URL"***);**  *String* **strITO**=**AutoUtils.getProperty(***CONFIG\_PATH, "ITO"***);**  *long* **ITO** = Long.parseLong**(**strITO**);**  **driver=new ChromeDriver();**  driver.**get(***appURL***);**  driver.manage().window().maximize();  driver.manage().timeouts().**implicitlyWait(***ITO, TimeUnit.SECONDS***);**  **}**  **@AfterMethod(alwaysRun=true)**  public void **closeApp(ITestResult** testResult**)**  **{**  *String* **testName**=testResult.**getName();**  *int* **status**=testResult**.getStatus();**  **if(status==1)**  **{**  **Reporter.log(**testName*+" is* ***PASSED****", true***);**  **}**  **else**  **{**  **Reporter.log(**testName*+" is* ***FAILED****", true****);***  **AutoUtils.getPhoto(***driver, IMAGE\_PATH, testName***);**  **}**  **driver.close();**  **}**  **}** |

**GitHub:**

‘GitHub’ is an online repository which is used to store the source code. In automation we can use GitHub to store the automation framework.

After designing framework, it will be stored in GitHub repository so that all the automation team can access the same repository (Framework).

**Q) How many times we need to update Framework in GitHub?**

**🡪** Only once, that too once the framework is designed.

**Q) What is required to upload framework in GitHub?**

1. GitHub Account

2. GitHub Desktop application

**Creation of GitHub Account:**

1. Go to github.com
2. Click on ‘Sing-Up’
3. Enter Username, Email & Password
4. Click on ‘Create Account’

**GitHub Desktop Application:**

1. Go to <https://desktop.github.com/>
2. Download the software & install.

**Steps to Upload Framework:**

https://joshuadull.github.io/GitHub-Desktop/02-getting-started/index.html

1. Open ‘GitHub Desktop’ application & sign-in.
2. Go to ‘File 🡪 New Repository’
3. Specify ‘**Name**’ (It should be same as Java Project Name [Ex:- CLGS]) & ‘**Local Path**’ (It should be same as path name of workspace [Ex:- E:\JAVA\Work\_Space]).Click on ‘Create repository’ button.
4. Click on ‘Publish Repository’ button.
5. Click on ‘Publish Repository’ button in pop-up. (Before clicking this button un-check ‘Keep this for private’ checkbox.
6. After publishing it will generate URL. URL will be [https://github.com/*UserName*/*ProjectName*](https://github.com/UserName/ProjectName)

**Ex:-** https://github.com/sudeer9/Framework

**Steps to Downloading Framework from GitHub:**

1. Open the browser & enter the URL of GitHub repository. (Ex:- https://github.com/sudeer9/Automation\_POM)
2. Click on ‘Clone or download’
3. Click on ‘Download Zip’.
4. After downloading the .zip file unzip it.
5. Open ‘Eclipse’, go to ‘File’ & select ‘Import’.
6. Under the option ‘General’ select ‘Existing Project into Workspace’, click ‘Next’.
7. Browse & select the extracted zip folder & follow default instructions till finish.

* Uploading initial designed Framework into ‘GitHub Repository’ which is called as **‘Publish’**.
* All the automation team will ‘**Import**’ the framework present in the ‘GitHub Repository’ into their respective local systems.
* After writing script or modifying the script or deleting the script it will be updated in ‘GitHub’ repository which is called as ‘**Push**’.
* In order to get the latest updates from online repository into local system, automation engineers should perform an action called ‘**Pull**’.

**Steps to Import Framework:**

1. Open ‘Eclipse’, go to ‘File’ & select ‘Import’.
2. Expand ‘Git’.
3. Select ‘Project from Git’.
4. Click ‘Next’. Then select “Clone URI” and click on next
5. Specify URI (Uniform Resource Identifier) [Ex:- [https://github.com/sudeer9/Automation\_POM.git](https://github.com/vijaykmutalik1987/CLGS.git)]
6. It will automatically display ‘Host’ & ‘Repository Path’ details.
7. Enter UserName & Password.
8. Click ‘Next’
9. In the next window,Browse and choose the newly created workspace directory.
10. Click next-next-Finish.

**Push:**

Whenever we create new File **(?)**, modify existing file **(>)** or deleting file **(x)** then we should push it to repository.

1. In ‘Eclipse’, Right click on ‘File’ or ‘Folder’.
2. Go to ‘Team’, select ‘Commit’.
3. Drag & Drop the files from ‘Un-staged’ into ‘Staged’.
4. Specify the comment then click ‘Commit & Push’.
5. Click ‘Ok’.

This will update any changes done in the local system into Online-repository.

**Pull:**

To get the latest changes from online repository to local system we should ‘Pull’ the code from repository.

1. Right click on ‘Java Project’ (Ex:- CLGS), go to ‘Team’.
2. Select the option ‘Pull’, click on ‘Close’.

**Framework Implementation:**

Converting manual test cases into automation script using the framework is called as ‘Framework Implementation’.

* In automation we automate only regression test cases given by manual testing team.
* While automating the test cases we develop 2-types of classes.

1. **POM Class**
2. **Test Class**

**Guidelines for POM class:**

1. Execute the given test cases manually so that we get more clarity on the steps which are to be automated.
2. While executing the test cases note down the **page elements** present on the page & **actions to be performed** on that page with respect to test case steps.
3. After executing all the test cases develop respective ‘Page’ class with elements & methods.

**Rules to develop POM-Class:**

1. Number of POM classes should be same as number of webpages present in the application & POM class should be created under ‘page’ package.
2. Name of the class should be same as respective webpage name & class name should end with ‘Page’ word. **Ex:-** LoginPage.
3. In every POM class elements should be declared using **‘@FindBy**’ annotation.
4. Elements should be initialized using ‘**PageFacrory.initElements()**’ inside the Constructor.
5. Develop ‘**public**’ method for all the actions which are to be performed on the page.

**Sample Regression Test cases:**

1. **Valid Login**

Enter valid username

Enter valid password

Click on ‘Login’ button

Verify ‘Home page’ is displayed or not

1. **Invalid Login**

Enter valid Username

Enter invalid password

Click on ‘Login’ button.

Verify validation message is displayed or not.

1. **Check Product Version**

Enter valid username.

Enter valid password.

Click on ‘Login’ button.

Click on ‘Help’.

Click on ‘About Acti-time’.

Verify the version.

**List of Page Elements & actions for above test cases:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl. No** | **Page Name** | **Elements** | **Actions** |
| 1 | LoginPage | UN text field, PW text field, Login button, Error message text | 1. Enter username 2. Enter password 3. Click login 4. Verify the text of the error message |
| 2 | HomePage | Help button, About actiTime button, Product Version text. | 1. Verify title. 2. Click Help. 3. Click About actiTime 4. Verify text of product version. |

**POM class for LoginPage:**

|  |
| --- |
| public class **LoginPage**  {  @FindBy(id="username")  private WebElement unTB;  @FindBy(name="pwd")  private WebElement pwTB;  @FindBy(xpath="//div[.='Login ']")  private WebElement loginBTN;  @FindBy(xpath="//span[contains(.,'invalid')]")  private WebElement errorMSG;  public **LoginPage(WebDriver driver)**  **{**  **PageFactory.initElements(driver, this);**  **}**  *//Methods*  public void setuserName(String un)  {  unTB.sendKeys(un);  }  public void setPassword(String pw)  {  pwTB.sendKeys(pw);  }  public void clickLogin()  {  loginBTN.click();  }  public void verifyErrorMSG(String expectedMSG)  {  String actualMSG = errorMSG.getText();  Assert.assertEquals(actualMSG, expectedMSG);  }  } |

**POM class for HomePage:**

|  |
| --- |
| public class **EnterPage**  {  @FindBy(xpath="//div[contains(text(),'Help')]")  private WebElement help;  @FindBy(linkText="About your actiTIME")  private WebElement aboutAT;  @FindBy(xpath="//span[@class='productVersion']")  private WebElement version;    public **EnterPage(WebDriver driver)**  **{**  **PageFactory.initElements(driver, this);**  **}**  *//Methods*  public void clickHelp()  {  help.click();  }  public void clickAboutActiTIME()  {  aboutAT.click();  }  public void verifyProductVersion(String ExpectedVersion)  {  String ActualVersion = version.getText();  Assert.assertEquals(ActualVersion, ExpectedVersion);  }  public void verifyHomePageDisplayed(WebDriver driver, String eTitle)  {  String aTitle = driver.getTitle();  Assert.assertEquals(aTitle, eTitle);  Reporter.log("Actual tiltle is matching with the Expected tiltle",true);  }} |

***Test Scripts:***

**Valid Login:**

|  |
| --- |
| public class **ValidLogin\_Test** *extends* **BaseTest**  {  **@Test(priority=1,groups= {"login","smoke"})**  public void **testValidLogin()** throws InterruptedException  {  String un=Excel\_Data.getValue(EXCEL\_PATH, "Valid Login", 1, 0);  String pw=Excel\_Data.getValue(EXCEL\_PATH, "Valid Login", 1, 1);  String eTitle=Excel\_Data.getValue(EXCEL\_PATH, "Valid Login", 1, 2);  LoginPage l=new LoginPage(driver);  l.setuserName(un);  Thread.sleep(2000);  l.setPassword(pw);  Thread.sleep(2000);  l.clickLogin();  EnterPage e=new EnterPage(driver);  e.verifyHomePageDisplayed(driver, eTitle);    }  } |

**Invalid Login:**

|  |
| --- |
| public class **InvalidLogin\_Test** *extends* **BaseTest**  {  **@Test(priority=2,groups= {"login"})**  public void **testInvalidLogin()** throws InterruptedException  {  int rc=Excel\_Data.getRowCount(EXCEL\_PATH, "Invalid Login");  for(int i=1;i<=rc;i++)  {  String un=Excel\_Data.getValue(EXCEL\_PATH, "Invalid Login", i, 0);  String pw=Excel\_Data.getValue(EXCEL\_PATH, "Invalid Login", i, 1);  String expectedMsg=Excel\_Data.getValue(EXCEL\_PATH, "Invalid Login", i, 2);  LoginPage l=new LoginPage(driver);  //Enter User Name  l.setuserName(un);  l.setPassword(pw);  l.clickLogin();  Thread.sleep(2000);  l.verifyErrorMSG(expectedMsg);  Thread.sleep(2000);  }  }  } |

**Verify Product version:**

|  |
| --- |
| public class **VerifyProductVersion\_Test** *extends* **BaseTest**  {  **@Test(priority=3,groups= {"smoke"})**  public void **testProductVerson()** throws InterruptedException  {  int rc=Excel\_Data.getRowCount(EXCEL\_PATH, "Verify Product Version");  for(int i=1;i<=rc;i++)  {  String un=Excel\_Data.getValue(EXCEL\_PATH, "Verify Product Version", i, 0);  String pw=Excel\_Data.getValue(EXCEL\_PATH, "Verify Product Version", i, 1);  String version=Excel\_Data.getValue(EXCEL\_PATH, "Verify Product Version", i, 2);  LoginPage l=new LoginPage(driver);  l.setuserName(un);  l.setPassword(pw);  l.clickLogin();  Thread.sleep(2000);  EnterPage e=new EnterPage(driver);  e.clickHelp();  Thread.sleep(2000);  e.clickAboutActiTIME();  Thread.sleep(2000);  e.verifyProductVersion(version);  }  }  } |

**Q) Explain your ‘Automation Framework’?**

1. In our automation framework for every manual test case we have developed ‘**Test class**’ using **TestNG.**
2. Every ‘**Test class**’ extends ‘**BaseTest**’ class which is an ‘**abstract**’ class.
3. ‘**BaseTest**’ class contains **i) Static block, ii) Before Method iii) After Method** and also implements ‘**IAutoConstants**’ Interface.
4. ‘**IAutoConstants**’ interface contains all the constants such as **Keys, Values & Path of excel, config-file &image path.**

|  |
| --- |
| ***String* CHROME\_PATH=”**webdriver.chrome.driver**”;** |

1. During run time first ’**BaseTest**’ class will execute the ‘**Static block**’ which will set the ‘**path of driver executables**’.

|  |
| --- |
| **static**  **{**  **System.setProperty(***CHROME\_KEY*, *CHROME\_VALUE***);**  } |

1. After setting the path it will execute the **‘@BeforeMethod**’. It will get the URL & Timeouts from ‘Property file’.

|  |
| --- |
| *String* **appURL**=**AutoUtils.getProperty(***CONFIG\_PATH, "URL"***);** |

1. Then it will open the browser.

|  |
| --- |
| driver=**new ChromeDriver();** |

1. It will enter the URL

|  |
| --- |
| *String* **appURL**=**AutoUtils.getProperty(***CONFIG\_PATH, "URL"***);**  driver.**get(***appURL***);** |

1. It will set the timeout.

|  |
| --- |
| *String* **strITO**=**AutoUtils.getProperty(***CONFIG\_PATH, "ITO"***);**  *long* **ITO** = Long.parseLong**(**strITO**);**  driver.manage().timeouts().**implicitlyWait(***ITO, TimeUnit.SECONDS***);** |

1. After executing **‘@BeforeMethod’** it will start execution of ‘**Test method**’ present in the ‘Test classes.
2. In ‘Test method’ it will take the data from excel sheet.

|  |
| --- |
| *String* **un**=**Excel\_Data.getValue(**EXCEL\_PATH, "Valid Login", 1, 0**);** |

1. It will call the method present in ‘**POM class**’ which will perform action on the application.

|  |
| --- |
| **LoginPage l=new LoginPage(*driver*);**  l.**setuserName(*un*);** |

1. After executing all the steps of ‘Test method’ it will go to **‘@AfterMethod**’.
2. ‘@AfterMethod’ will check the status of the test & it will **take screenshot** if test case fails.

|  |
| --- |
| **If(ITestResult.getStatus==2)**  **{**  **AutoUtils.getPhoto(***driver, IMAGE\_PATH, testName***);**  **}** |

1. At the end it will close the browser.

|  |
| --- |
| driver**.close();**  Or  driver**.quit();** |

1. After executing all the scripts it will generate ‘**html-report**’ (emailable-report.html) inside the ‘**test-output**’ folder of the framework.

=====================================================================

**Q) What type of Framework you used in the project?**

**🡪** Hybrid framework which is the combination of 2 or more frameworks such as Data driven, Method driven, Module driven, TestNG & POM.

**Q) When do you run the framework?**

**🡪** Whenever we get new build we run the framework.

**Q) What is Build?**

**🡪** It is compiled & compressed copy of the code(software).

**Jenkins:**

* We write the automation script in our local system.
* We store all the script in the GitHub server.
* Execution of the script will be done in ‘Node’ systems.

**Testing Environment:**

1. Application is installed in build server.
2. Builds are created using build tools such as

\* ANT

\* Jenkins

\* Hudson

1. Build refers to compiled & compressed copy of the software.
2. To create build developers uses ‘Build-tool’.
3. Build is created whenever there is an addition, modification or removal of the feature.
4. We execute the framework whenever we get new build.
5. Instead of executing the framework manually we can execute automatically by integrating the framework with build tool.

**Executing framework by integrating with Build-tool:**

1. We write our automation script in local system & we store it in ‘GitHub repository’.
2. Developers uses Build tool to create the build.
3. After creating build, it is installed in ‘Build server’.
4. After installing the build, it will take the framework present in the ‘GitHub’ repository & it will execute the framework.
5. Scripts are executed in ‘Node’ system & results will be stored in the framework.

* There will be multiple ‘local systems’ depends on the number of automation engineers.
* There will be multiple ‘Node systems’ depends on testing environment.
* There are many ‘Build tools’ are available out of that very frequently use ‘Jenkins’.

**Jenkins:**

Jenkin is continuous integration tool. Because whenever any changes done by the developers to the source code of the application, Jenkins will automatically detects it & creates new build.

Jenkins can be download from:- <http://jenkins-ci.org/> Or <https://jenkins.io/>

**Steps to install Jenkins:**

1. Download the Jenkins software & unzip it, which will generate Jenkins.msi (Microsoft installer).
2. Double click on **‘Jenkins.msi’** file & follow the default instructions till finish. Go to services from start menu and start the jenkins service.
3. Open the browser, enter the URL of the Jenkins. (Ex:- [**http://localhost:8080/**](http://localhost:8080/) )
4. It will ask for the password. Copy the password from ***initialAdminPassword*** file and paste it. It will be in “C:\Program Files (x86)\Jenkins\secrets\initialAdminPassword”
5. Click on **‘Install Suggested Plugins‘**—It will install the plugins
6. For the first time it will ask for Username, Password & Confirm Password. Enter valid details & click **Save and Continue**.
7. Next time onwards use the same login details.

* In our automation framework we have used ‘GitHub’ for storing the framework & ‘Maven’ for managing the ‘Jar’ files. Hence we need ‘GitHub’ & ‘Maven’ plugins in ‘Jenkins’. This will be installed by ‘Jenkins Administrator’.

**Steps to Install Plugins:**

1. Login to Jenkins.
2. Click on ‘Manage Jenkins’.
3. Click on ‘Manage Plugins’.
4. Click on second tab 🡪 ‘Available’.
5. Search for ‘GitHub’.
6. Select **‘GitHub Authentication’** check-box.
7. Select **‘GitHub Integration’** check-box.
8. Click on ‘Install without Restart’ button.
9. Similarly search for ‘Maven’ & select **‘Maven Integration’** check-box, click on ‘Install without Restart’ button.
10. Once ‘Maven’ plugin installed then go to home page of Jenkins, click on ‘New Item’. It should display **‘Maven Project’** (if Maven plugin is installed successfully).

**Surefire Plugin:**

To execute ‘TestNG.xml’ from ‘POM.xml’ we need Surefire plugin.

* The plugin content is available in URL ‘ <http://maven.apache.org/surefire/maven-surefire-plugin/examples/testng.xml>’
* Copy the plugin content from above website. Go to ‘POM.xml’ & add this plugin content as new plugin block.

|  |
| --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>3.0.0-M5</version> |

**Steps to Integrate Framework with Jenkins:**

**Required Information’s:**

1. URL of the Jenkins (Ex:- <http://10.238.2.321:8082/> [Jenkins Server])
2. Username & password of Jenkins.
3. Development project name.

Ex:- Automation\_POM

1. GitHub URL (Ex:-<https://github.com/sudeer9/Automation_POM.git>)

**Steps:**

1. Login to Jenkins.
2. Click on ‘New Item’.
3. Specify the ‘Project Name’.
4. Click on ‘Maven Project’.
5. Click ‘Ok’.
6. Select “GitHub project” checkbox
7. Specify ‘Project url’
8. Select ‘Git’ radio-button.
9. Specify ‘URL’ of the GitHub.-save the configuration
10. Go to Home page of Jenkins
11. Click on “Manage Jenkins”
12. Click on “Global Tool Configuration”
13. Click on “JDK installations” and provide the name and path of JDK and uncheck “Install Automatically” check box
14. Click on “Maven installations” and provide the name and path of MAVEN and uncheck “Install Automatically” check box . If Maven is not available then select “Install Automatically” check box. It will get installed.
15. Save the configuration
16. Now onwards whenever developer creates a build, it will automatically executes framework & results will be stored in the Framework**(C:\Program Files (x86)\Jenkins\workspace\Automation\_POM\target\surefire-reports)**

**Executing Framework directly from Jenkins:**

1. Login to ‘Jenkins’.
2. Click on ‘Automation project’.
3. Click on ‘Build now’.

**Steps performed by ‘Jenkins’ when we click on ‘Build now’:**

1. Downloads the framework from GitHub to Jenkins system.
2. Executes ‘POM.xml’.
3. ‘POM.xml’ will update the Jar-files.
4. ‘POM.xml’ compiles the automation scripts.
5. ‘POM.xml’ executes ‘TestNG.xml’ (Surefire plugin)
6. ‘TestNG.xml’ will creates multiple threads, one for each ‘test block’.
7. Every thread will connect to respective m=’Node’ & executes the script.
8. After the execution results will be stored in ‘Jenkins’.

**Steps to see the Result in ‘Jenkins’:**

1. Login to ‘Jenkins’.
2. Click on ‘Automation project’
3. In the right side of the page we can see ‘Graphical’ result.
4. To view the details, click on link present under ‘Build history’ then click on ‘Test Result’. In the same page you can also see ‘Console output’.
5. In order to see older results, click on old links present under ‘Build history’.

**Q) How do you schedule execution of Framework?**

1. Login to ‘Jenkins’.
2. Click on ‘Automation project’.
3. Click on ‘Configure’. Select check-box ‘Build periodically’. Specify the schedule & click ‘Save’.

**Syntax of the Schedule:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MINUTE** | **HOUR** | **DOM** | **MONTH** | **DOW** |

* MINUTE 🡪 minutes within the hour (0-59)
* HOUR 🡪 The hour of the day (0-23)
* DOM 🡪 The day of the month (1-31)
* MONTH 🡪 The month (1-12)
* DOW 🡪 The day of the week (0-7) where 0 & 7 are ‘Sunday’.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 30 | 10 | \* | \* | \* | 🡪 every day @ 10:30 AM |
| 00 | 18 | \* | 3 | 5 | 🡪 March all Fridays @ 6PM |
| 00 | 18 | 23 | 3 | 5 | March 23 Friday @ 6PM |

**Parallel Execution:**

**Refer attached programs :**

1. SuperClass
2. Test1
3. Test2

**Selenium Grid:**

Starting HUB machine: java -jar seleniumJarFile.jar -role hub

Starting NODE machine and registering with HUB machine: java -jar seleniumJarFile.jar -role node -hub http://192.168.0.74:4444/grid/register