Quick and Simple Selenium Automation framework setup guide

Contents:

1. Downloading the repository
2. Importing project into Eclipse
3. Framework structure overview
4. Additional downloads
5. Setting up Grid
6. Setting up CI (Jenkins)
7. Running tests

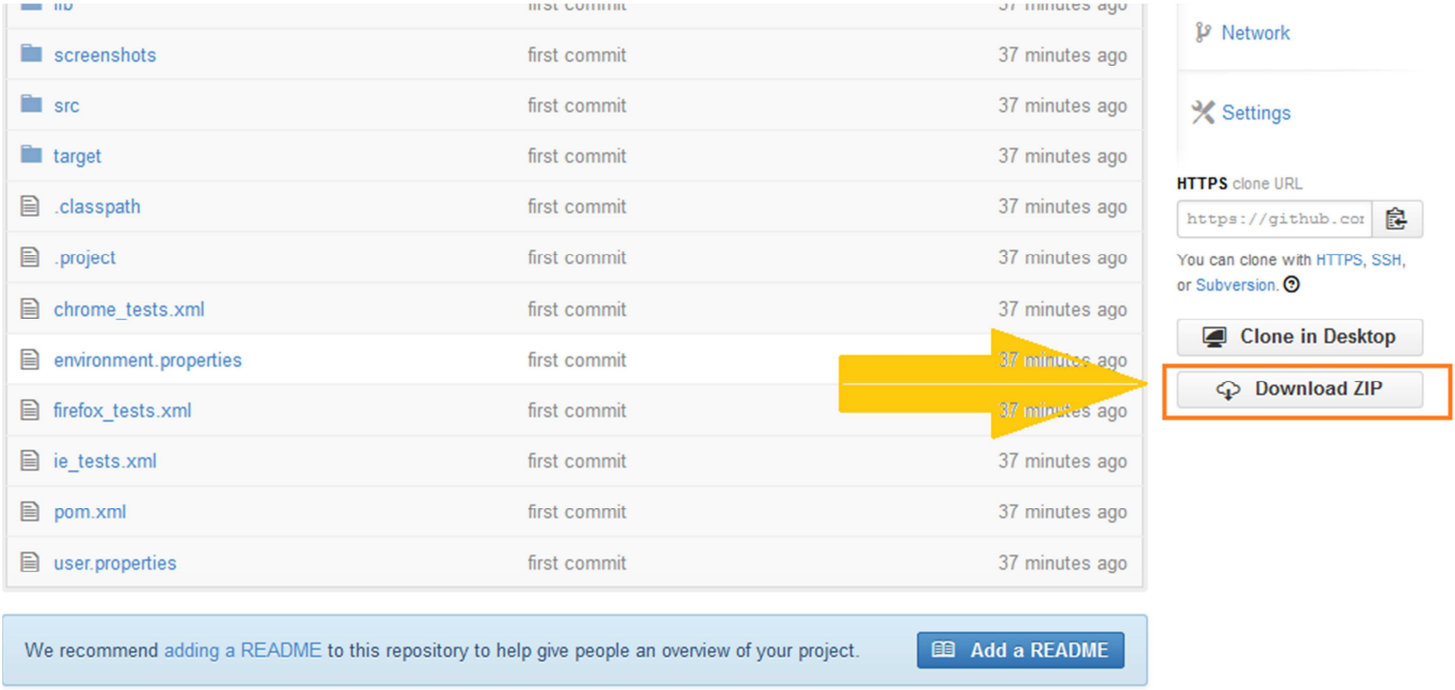
Purpose:

1. Help QA Engineers get basic idea of automation framework structure
2. Provide some useful examples for future use
3. Help understand that automation can be easy and fun

1. Downloading the framework

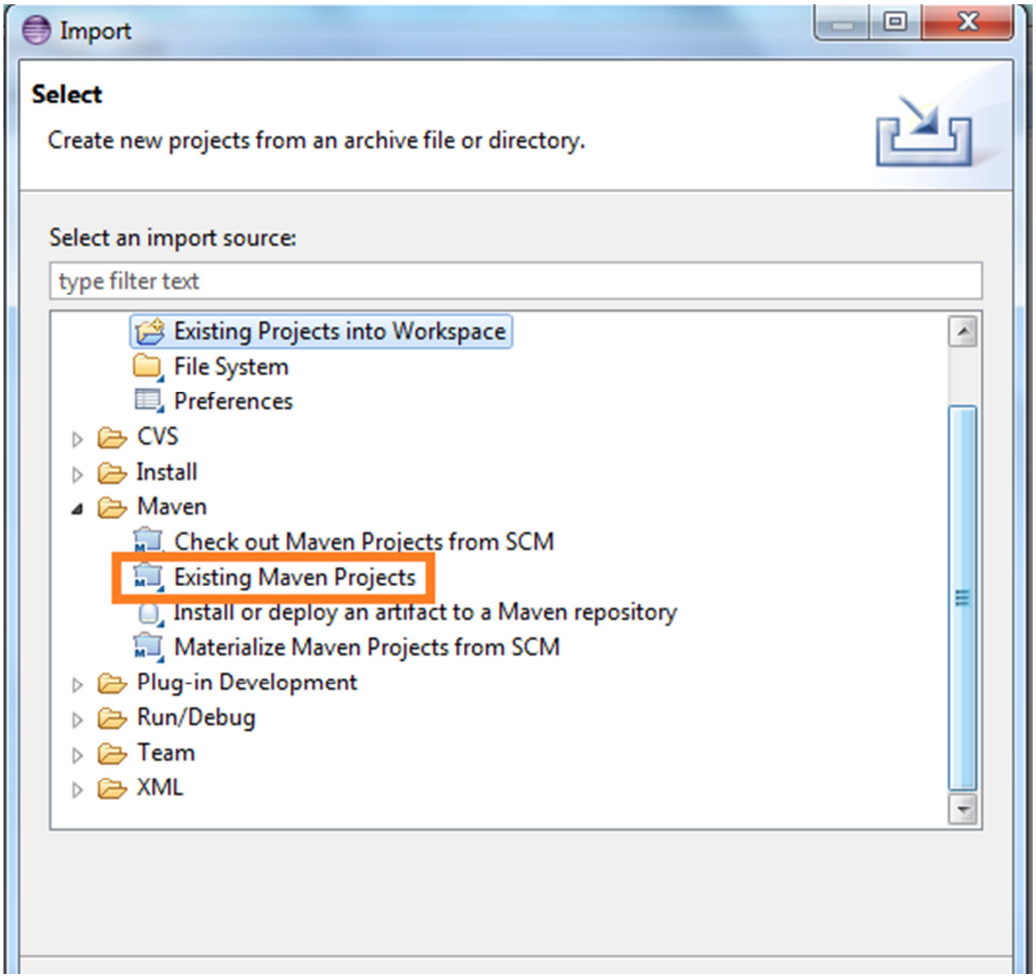
Step I:
Go to: https://github.com/bbatsalenka/automation-starter-kit

Step II:
Download ZIP as shown on the screenshot:

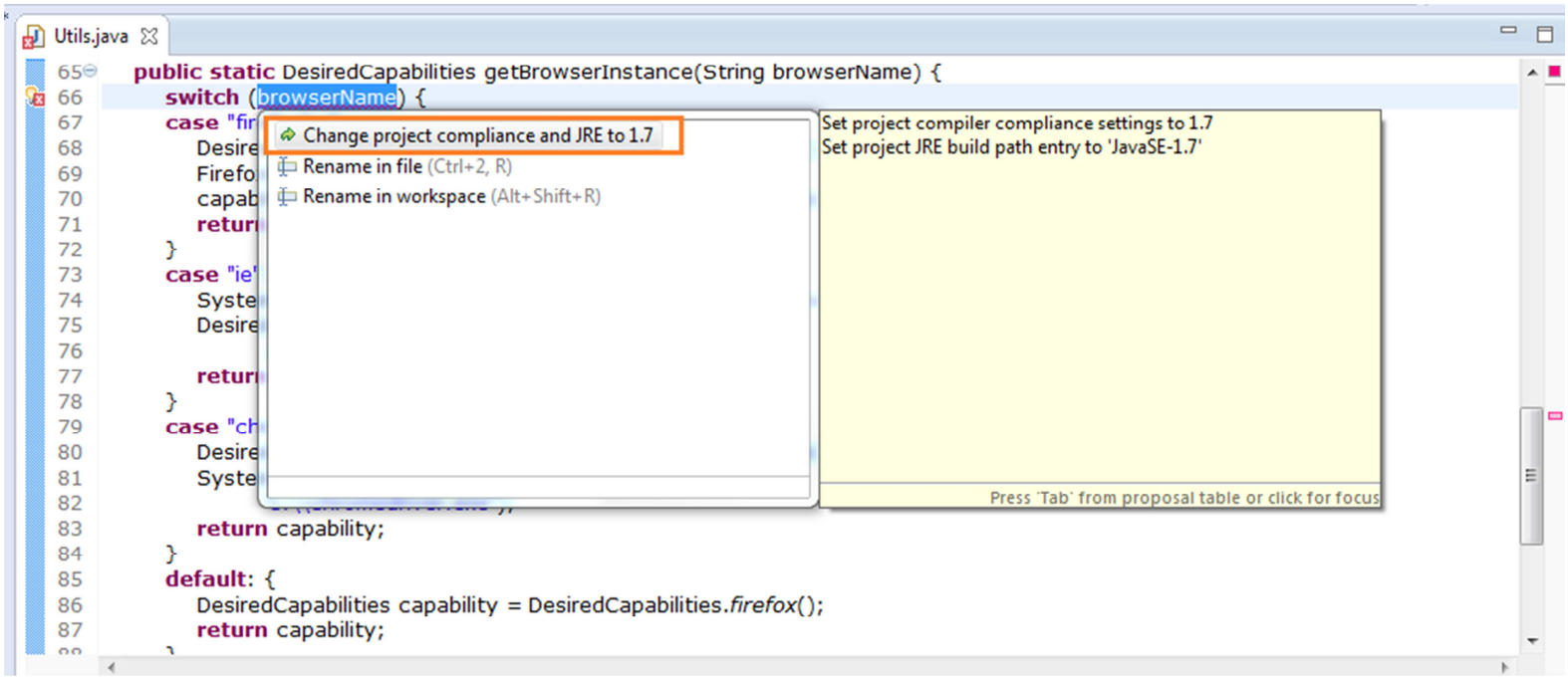


2. Importing project into Eclipse

Step I: Import downloaded framework as “Existing Maven Projects” (see screenshot) in Eclipse

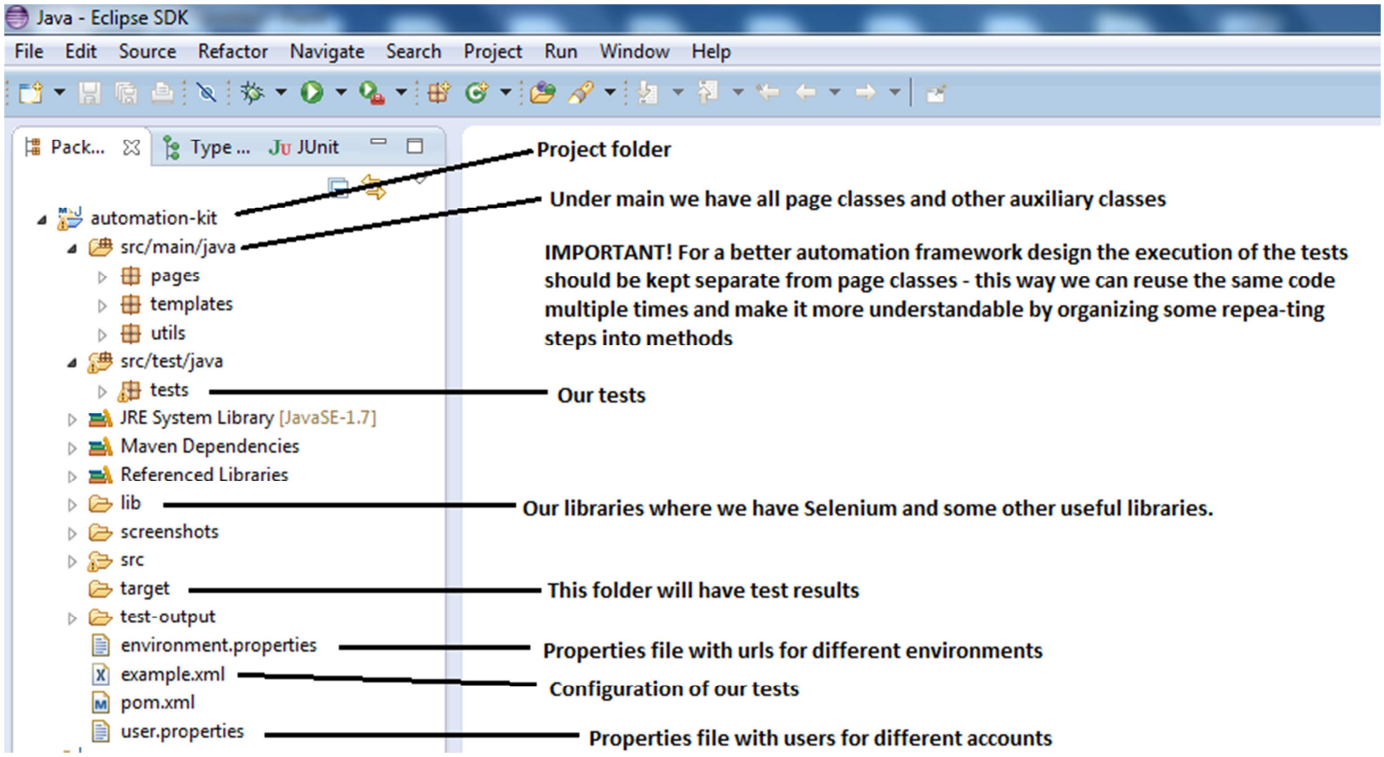


Step II: if there are issues in the project related to java 1.7 (as on the screenshot), use the solution provided by Eclipse – Change project

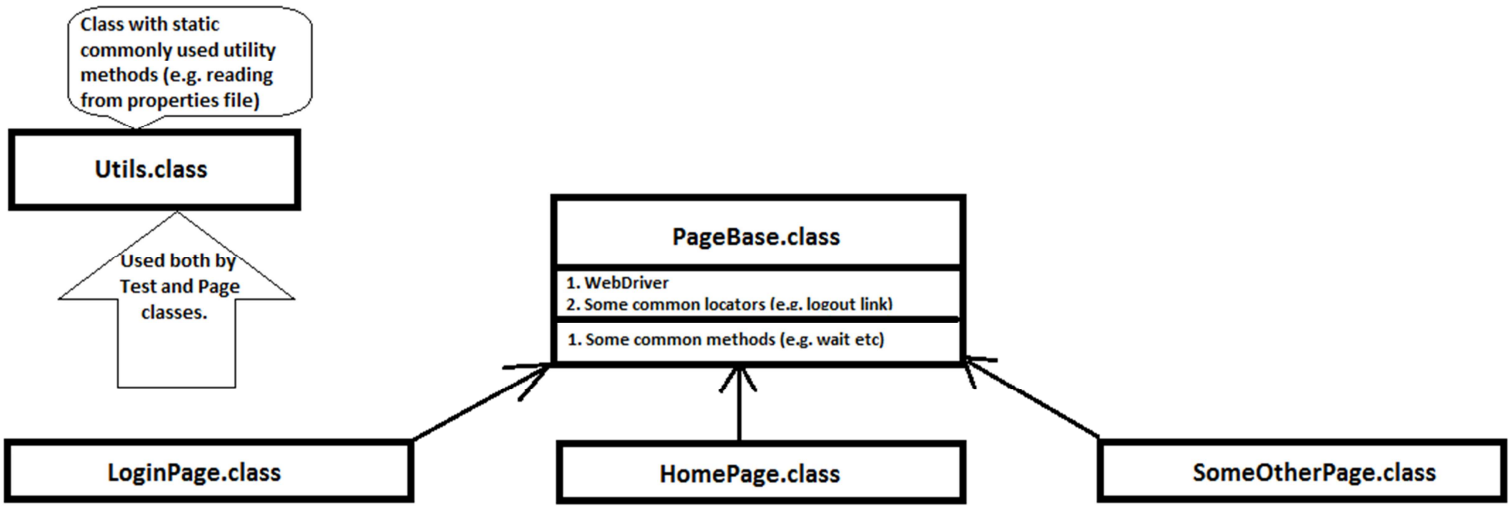


compliance and JRE to 1.7

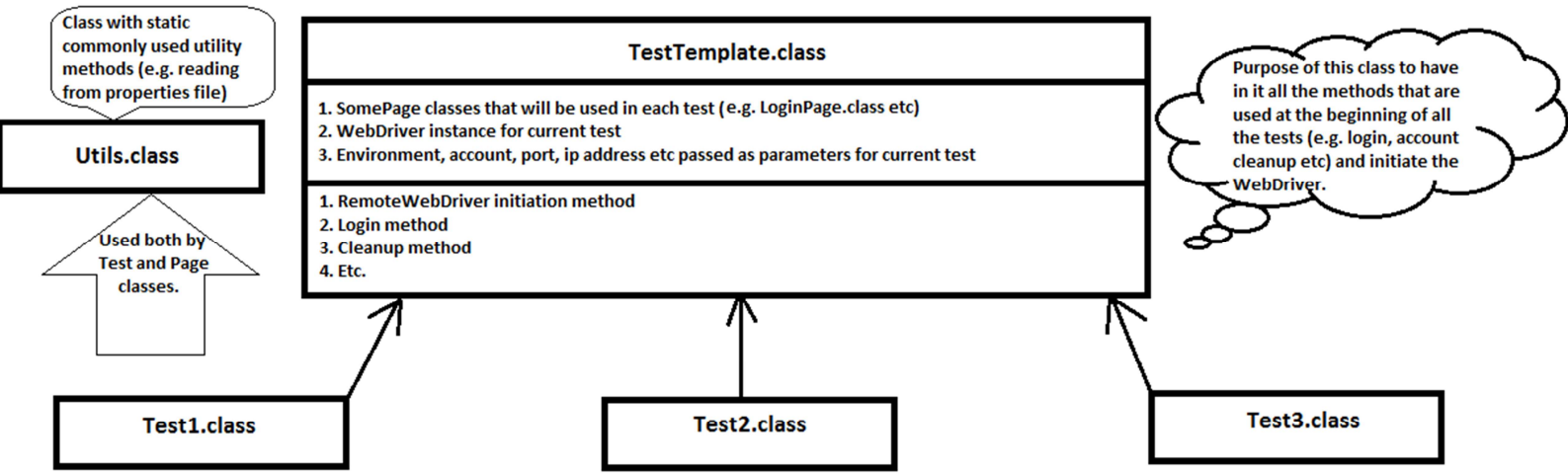
3. Framework structure overview



Below is inheritance diagram of the framework Page classes.



Here is the diagram of the Test classes hierarchy:



Each page extends the PageBase.class that has some common locators and methods in it (e.g. logout or terms of use link, waitForPageLoaded() method etc.).

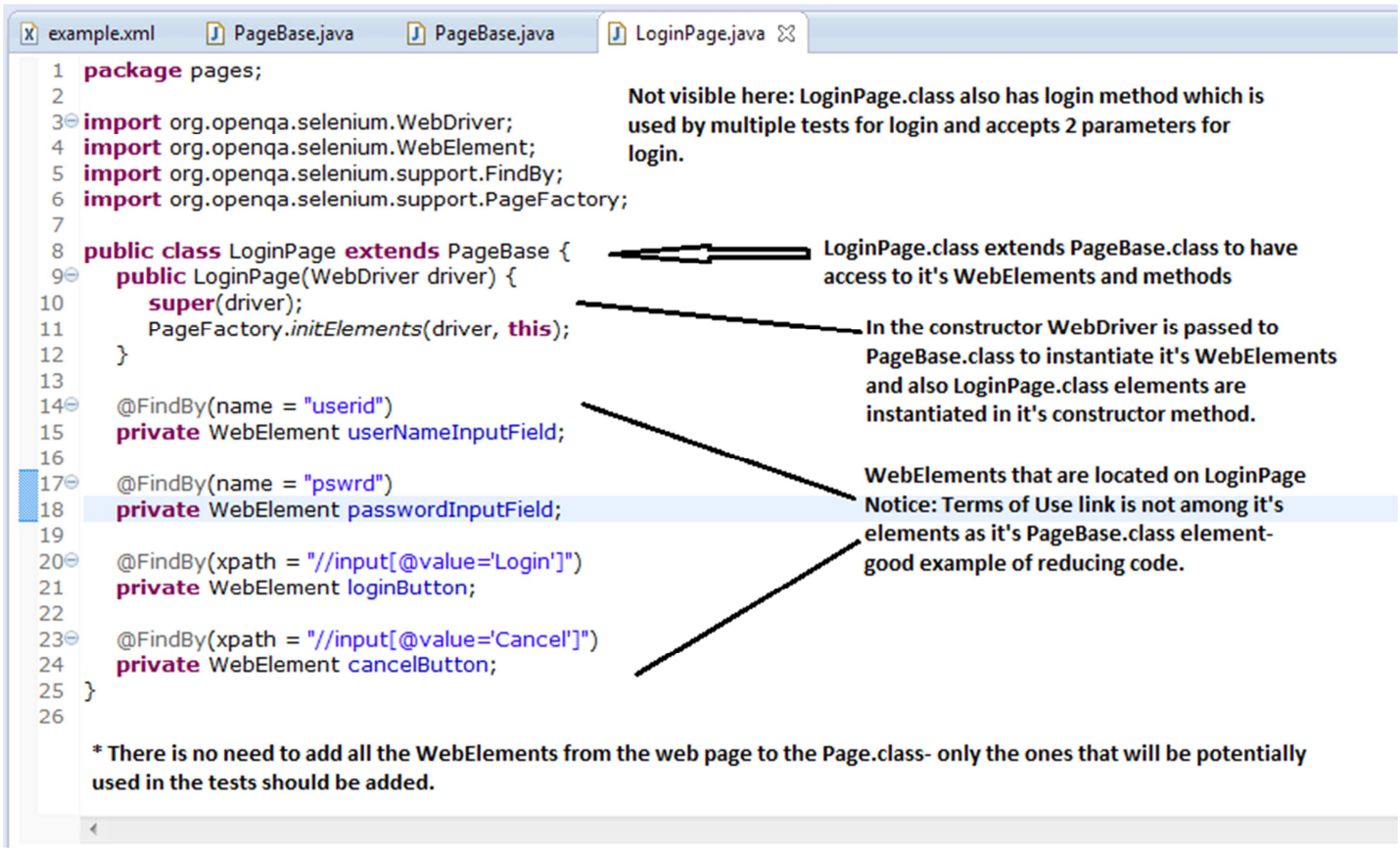
example.xml | PageBase.java | PageBase.java

```
1  package pages;
2
3  import org.openqa.selenium.WebDriver;
4  import org.openqa.selenium.WebElement;
5  import org.openqa.selenium.support.FindBy;
6  import org.openqa.selenium.support.PageFactory;
7
8  // each page extends this class that has some common methods and WebElements
9  public abstract class PageBase {
10
11     protected WebDriver driver;
12     protected static final int DEFAULT_WAIT_4_ELEMENT = 25;
13     protected static final int DEFAULT_WAIT_4_PAGE = 30;
14
15     // PageBase constructor that initiates WebElement
16     public PageBase(WebDriver driver) {
17        this.driver = driver;
18        PageFactory.initElements(driver, this);
19     }
20
```

PageBase.class that all other pages will extend

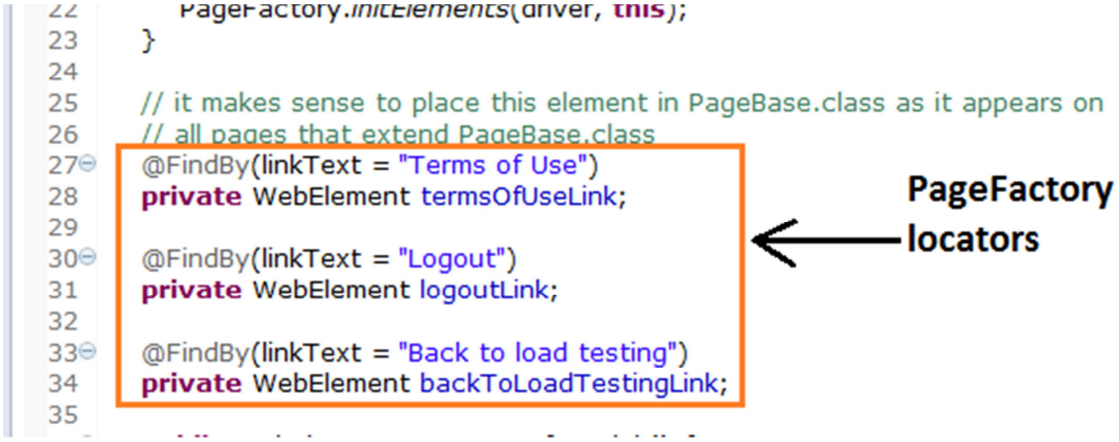
Use PageFactory to instantiate and access it's elements

Here is an example of LoginPage.class that extends PageBase.class:




```
1  package pages;
2
3  import org.openqa.selenium.WebDriver;
4  import org.openqa.selenium.WebElement;
5  import org.openqa.selenium.support.FindBy;
6  import org.openqa.selenium.support.PageFactory;
7
8  public class LoginPage extends PageBase {
9      public LoginPage(WebDriver driver) {
10         super(driver);
11         PageFactory.initElements(driver, this);
12     }
13
14     @FindBy(name = "userid")
15     private WebElement userNameInputField;
16
17     @FindBy(name = "pswrd")
18     private WebElement passwordInputField;
19
20     @FindBy(xpath = "//input[@value='Login']")
21     private WebElement loginButton;
22
23     @FindBy(xpath = "//input[@value='Cancel']")
24     private WebElement cancelButton;
25 }
26
```

To store WebElements as class fields we use PageFactory (the elements are usually initialized in the constructor as is shown on the screenshot above)



To structure each test and supply some parameters to it, like: environment, user credentials, browser parameters, parameters for the Grid we are using .xml format TestNG file:

example.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Firefox-Tests" parallel="tests" preserve-order="true"
   thread-count="4" verbose="5">
   <listeners>
      <listener class-name="packageName.ScreenshotOnFailure" />
   </listeners>
   <test name="Example">
      <parameters>
         <parameter name="browser" value="firefox" />
         <parameter name="environment" value="stage" />
         <parameter name="user" value="stage_11" />
         <parameter name="port" value="5555" />
         <parameter name="ipAddress" value="localhost" />
      </parameters>
      <classes>
         <class name="packageName.ExampleTest" />
      </classes>
   </test>
</suite> <!-- Suite -->
```

Configuring tests for running in parallel, can change the thread-count for some other number and verbose.

Listener class added just once per suite for taking screenshots on failure

Test name

Browser parameters

Some other parameters as testing environment, users, etc.

Node parameters for Selenium Grid (localhost is used if the local machine both a hub and a node, otherwise remote machine ip should be used)

Class name with specific tests

We add each class that has specific tests in it to .xml tests configuration file as a separate <test>. Each .xml configuration file has a ScreenshotOnFailure listener in it that takes screenshots when a test failure occurs.

Note: if structure of the framework is followed the ScreenshotOnFailure listener works perfectly fine for tests running in parallel and takes screenshot for the right browser when a failure occurs.

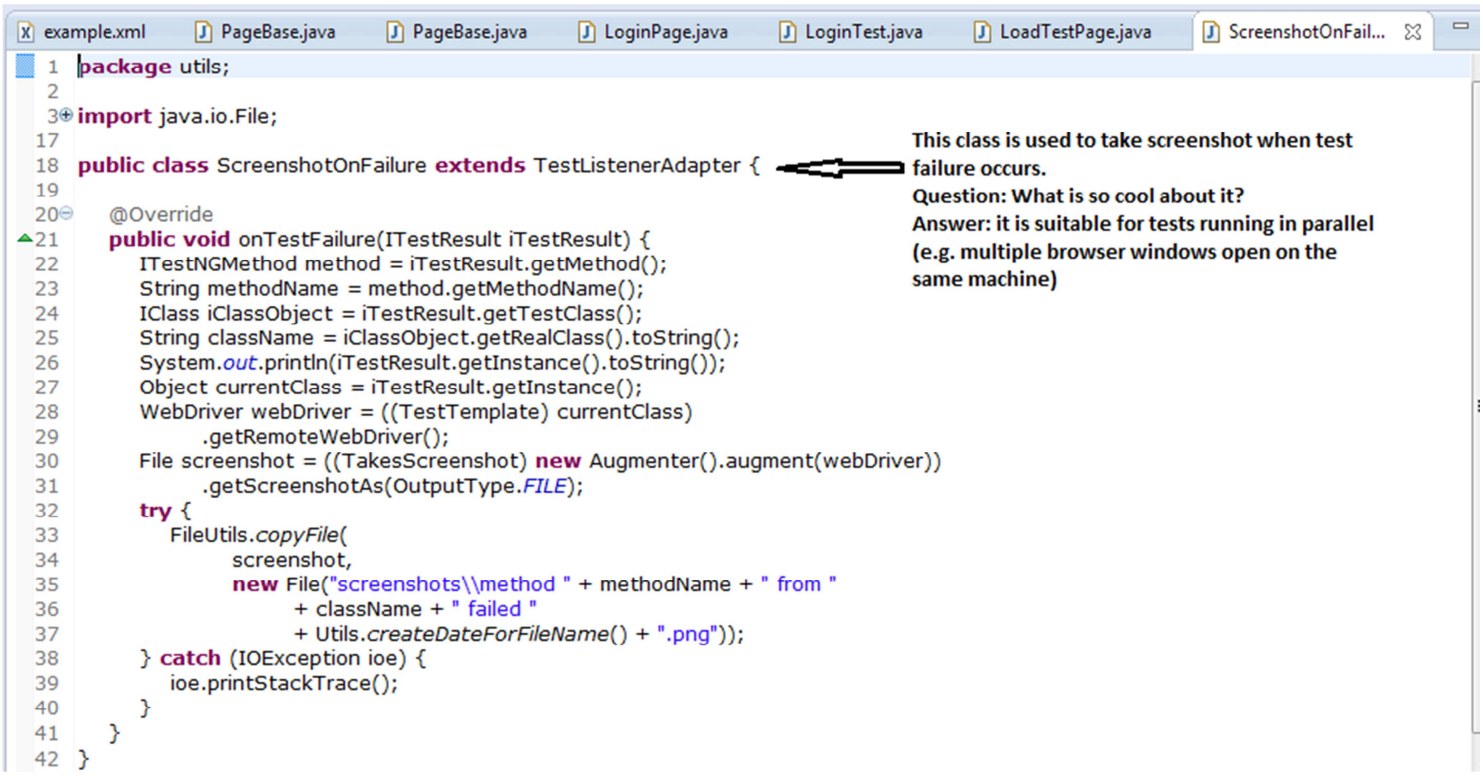
example.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Firefox-Tests" parallel="tests" preserve-order="true"
   thread-count="4" verbose="5">
   <listeners>
      <listener class-name="packageName.ScreenshotOnFailure" />
   </listeners>
   <test name="Test I">
      <parameters>
         <parameter name="someParameter" value="someParameter" />
      </parameters>
      <classes>
         <class name="packageName.ExampleTestI" />
      </classes>
   </test>
   <test name="Test II">
      <parameters>
         <parameter name="someParameter" value="someParameter" />
      </parameters>
      <classes>
         <class name="packageName.ExampleTestII" />
      </classes>
   </test>
</suite> <!-- Suite -->
```

Listener is one per suite

Insert each test class with specific tests as is shown

Below is the ScreenshotOnFailure.class declaration. When a screenshot is taken it has it's name as the method in which the failure occurred with date and time.




```
1  package utils;
2
3  import java.io.File;
17
18 public class ScreenshotOnFailure extends TestListenerAdapter {
19
20    @Override
21    public void onTestFailure(ITestResult iTestResult) {
22       ITestNGMethod method = iTestResult.getMethod();
23       String methodName = method.getMethodName();
24       IClass iClassObject = iTestResult.getTestClass();
25       String className = iClassObject.getRealClass().toString();
26       System.out.println(iTestResult.getInstance().toString());
27       Object currentClass = iTestResult.getInstance();
28       WebDriver webDriver = ((TestTemplate) currentClass)
29             .getRemoteWebDriver();
30       File screenshot = ((TakesScreenshot) new Augmenter().augment(webDriver))
31             .getScreenshotAs(OutputType.FILE);
32       try {
33          FileUtils.copyFile(
34                screenshot,
35                new File("screenshots\\method " + methodName + " from "
36                      + className + " failed "
37                      + Utils.createDateForFileName() + ".png"));
38       } catch (IOException ioe) {
39          ioe.printStackTrace();
40       }
41    }
42 }
```

To run the tests from CI tool like Jenkins we will need to provide Maven commands to it, like:

-Dtests=ie_tests.xml -Dreceiver=youremail@domain.com test

ie_tests.xml – is the .xml file with specific tests that you want to run.

For this we need to configure our pom.xml file (it's already configured and here is just a brief overview):

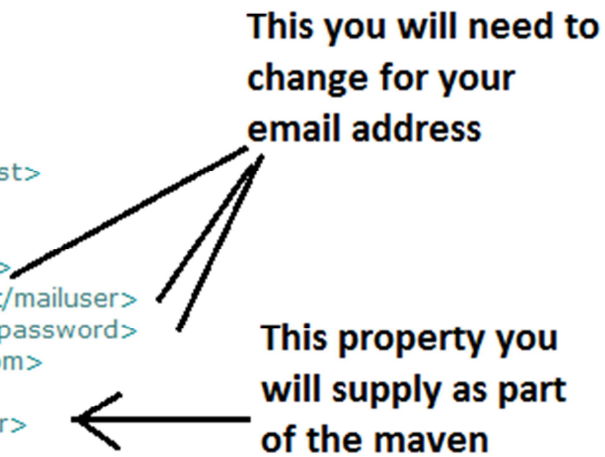
We supply .xml file names to maven-surefire-plugin to run all the tests contained in this .xml file

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.15</version>
    <configuration>
        <testFailureIgnore>true</testFailureIgnore>
        <suiteXmlFiles>
            <suiteXmlFile>${tests}</suiteXmlFile>
        </suiteXmlFiles>
    </configuration>
</plugin>
```

We are using maven-postman-plugin to send tests-run report and screenshots (if any) to the specified email address.

Note: you will need to change the “From” property to the email address from which you want to receive emails.

```
<plugin>
    <groupId>ch.fortysix</groupId>
    <artifactId>maven-postman-plugin</artifactId>
    <version>0.1.6</version>
    <executions>
        <execution>
            <id>send_an_email</id>
            <phase>test</phase>
            <goals>
                <goal>send-mail</goal>
            </goals>
            <configuration>
                <mailhost>smtp.gmail.com</mailhost>
                <mailport>465</mailport>
                <mailssl>true</mailssl>
                <mailAltConfig>true</mailAltConfig>
                <mailuser>bagditester@gmail.com</mailuser>
                <mailpassword>[redacted]</mailpassword>
                <from>bagditester@gmail.com</from>
                <receivers>
                    <receiver>${receiver}</receiver>
                </receivers>
                <fileSets>
                    <fileSet>
                        <directory>${basedir}/target/surefire-reports</directory>
                        <includes>
                            <include>emailable-report.html</include>
```



```
                <includes>
                    <include>emailable-report.html</include>
                </includes>
            </fileSet>
            <fileSet>
                <directory>${basedir}/screenshots</directory>
                <includes>
                    <include>**/*.png</include>
                </includes>
            </fileSet>
        </fileSets>
        <subject>Important subject</subject>
        <failonerror>true</failonerror>
        <htmlMessage>
<![CDATA[
<p>Attached is tests run report.</p>
<br>
<p>Have a nice day.</p>
]]>
        </htmlMessage>
    </configuration>
</execution>
</executions>
</plugin>
```

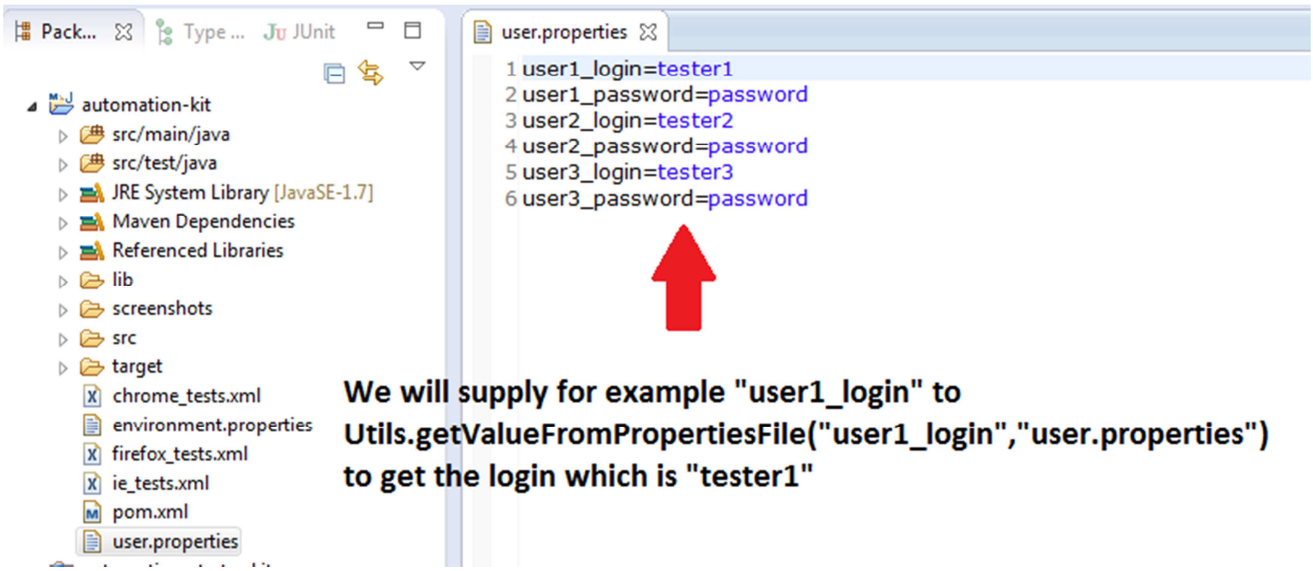
Name of the report

Screenshots location

Email subject

Email body

In our tests we also use .properties files to store parameters for our tests (e.g. “environment”=”stage”, “user”=”testuser”, “password”=”password”) – the advantage of it is just to supply a value to the method Utils.getValueFromPropertiesFile(String variable, String filename) and the method will return the value of requested variable.



4. Additional downloads

   Additionally you will need to download the following:

   i. Selenium-standalone-server
   ii. Chromedriver.exe
   iii. IEDriverServer.exe
   iv. Jenkins CI tool

5. Setting up Grid:

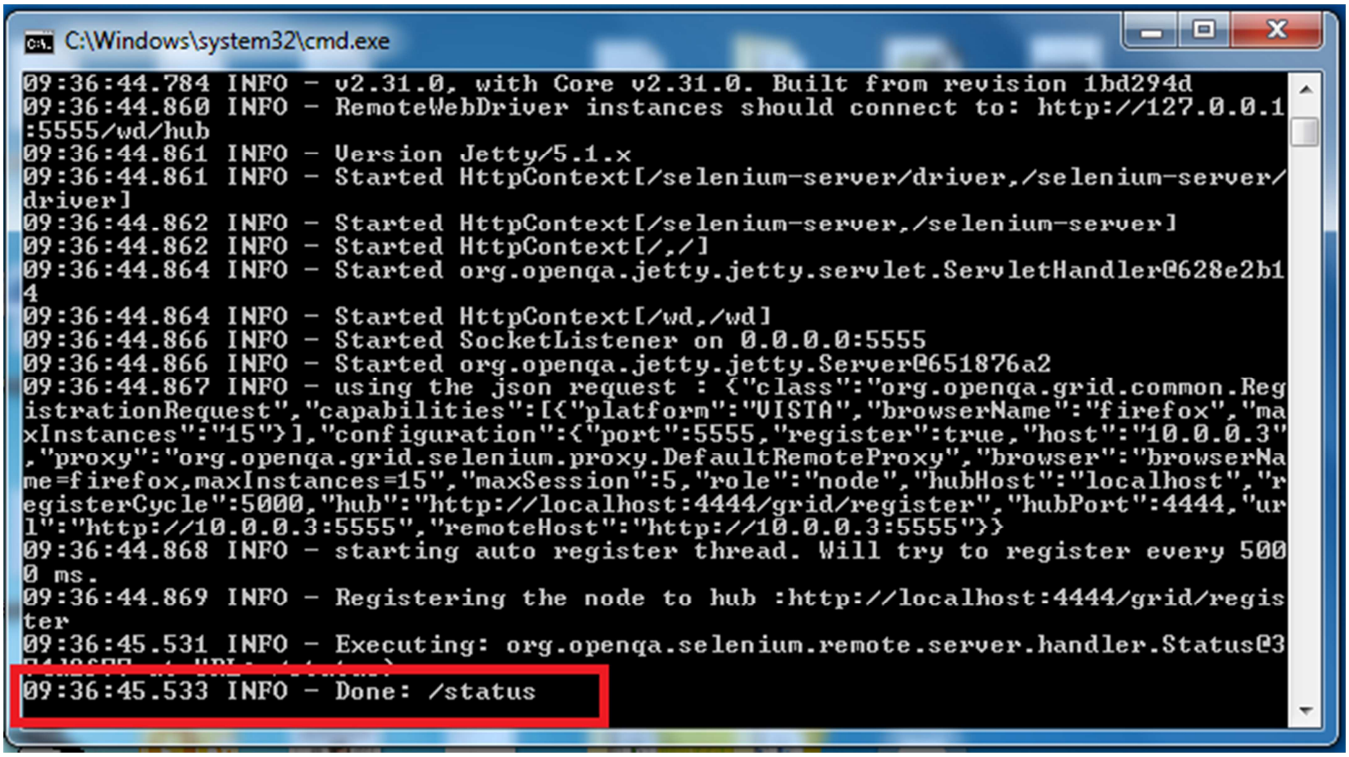
   Since you will have the hub of your Grid integrated with your Jenkins CI you will need only to start the nodes either on your local machine or on remote ones so that you could run tests on them.

   To configure the node you will need to create a .cmd file with the following command in it:

   cd to the location of selenium-server-standalone-version.jar
   java -jar selenium-server-standalone-2.31.0.jar -role node -hub http://localhost:4444/grid/register
   \ -browser browserName=firefox,platform=WINDOWS
   \ -browser browserName=internet explorer,version=9,platform=WINDOWS
   \ -browser browserName=chrome,platform=WINDOWS

   If you are not using any nodes, then the ip address or hostname should be the one of local machine (you do not need to start the hub as it's a part of Selenium Grid plugin on Jenkins – the setup will be explained further)

After you start the node, make sure your node is connected to the hub, look for the following line:



You will need to place the previously downloaded .exe and .jar files (Selenium-standalone-server, Chromedriver.exe, IEDriverServer.exe) on the C:\ drives of your nodes or some other location, then you will need to change it in the getBrowserInstance() method. Also you will need to have java on your nodes:

```
public static DesiredCapabilities getBrowserInstance(String browserName) {
   switch (browserName) {
   case "firefox": {
      DesiredCapabilities capability = DesiredCapabilities.firefox();
      FirefoxProfile firefoxProfile = new FirefoxProfile();
      capability.setCapability(FirefoxDriver.PROFILE, firefoxProfile);
      return capability;
   }
   case "ie": {
      System.setProperty("webdriver.ie.driver", "C:\\IEDriverServer.exe");
      DesiredCapabilities capability = DesiredCapabilities
            .internetExplorer();
      return capability;
   }
   case "chrome": {
      DesiredCapabilities capability = DesiredCapabilities.chrome();
      System.setProperty("webdriver.chrome.driver",
            "c:\\chromedriver.exe");
      return capability;
   }
   default: {
      DesiredCapabilities capability = DesiredCapabilities.firefox();
      return capability;
   }
   }
}
```

If you place the drivers for browsers somewhere else - you will need to change the location in this method

6. Setting up Jenkins:

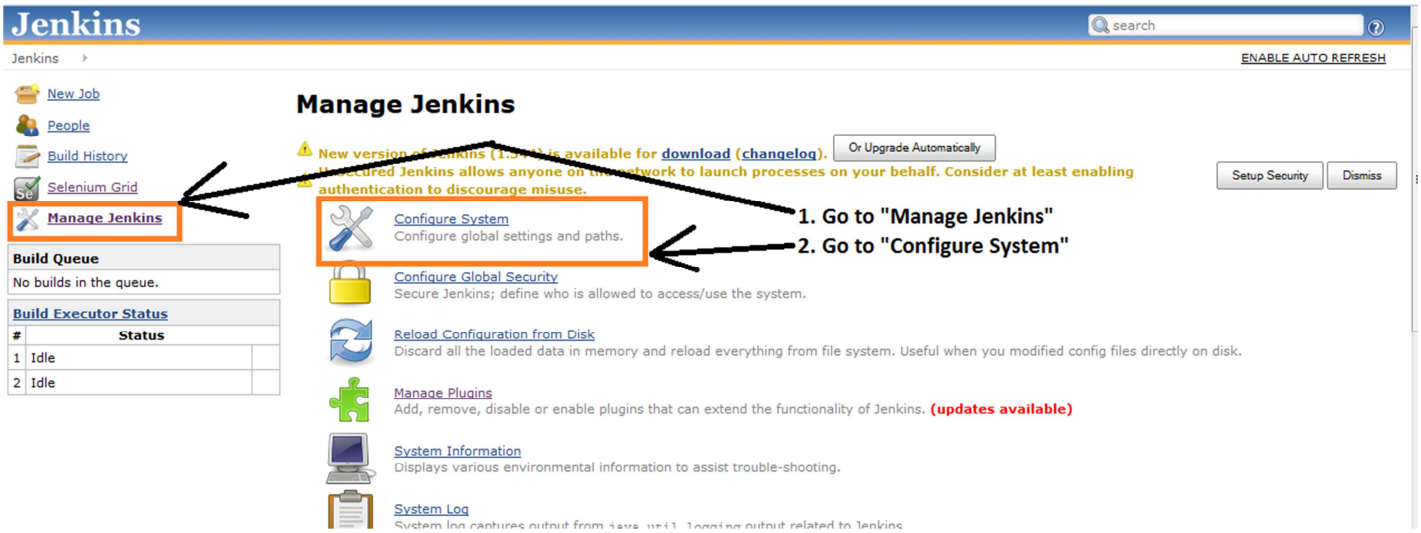
   You will need to download Jenkins CI and install it on your hub machine.

   After it's installed you should type in you browser: localhost:8080 and if the installation is successful you will see the Jenkins dashboard.

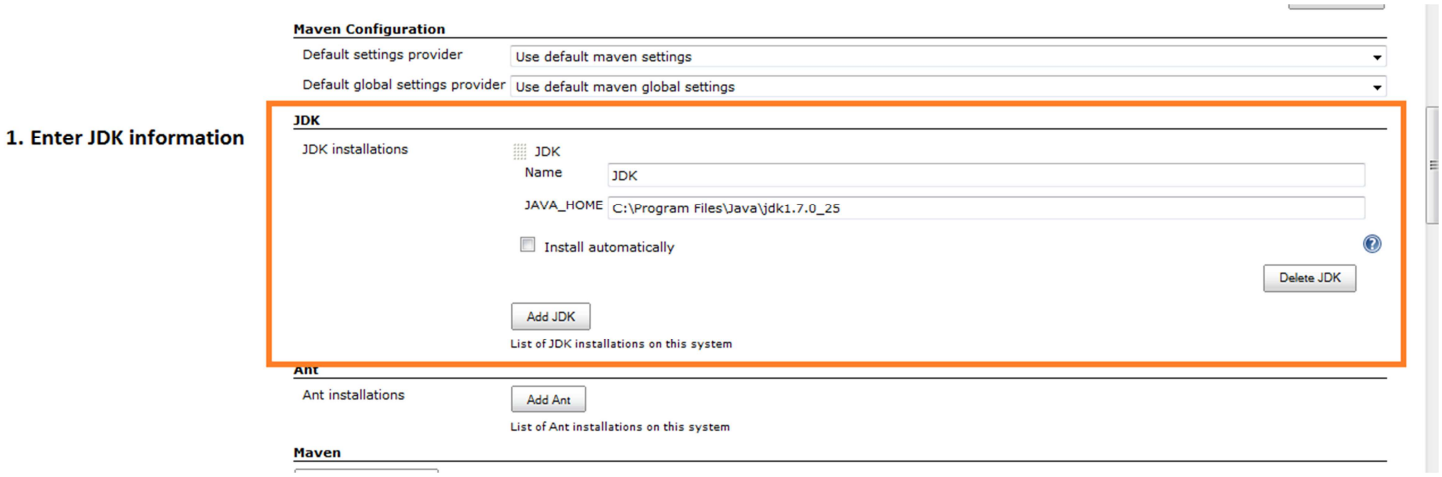
   As a prerequisite for further Jenkins setup you will need to have Maven and Java installed and set environmental variables for them.

   i. Configuring Jenkins:

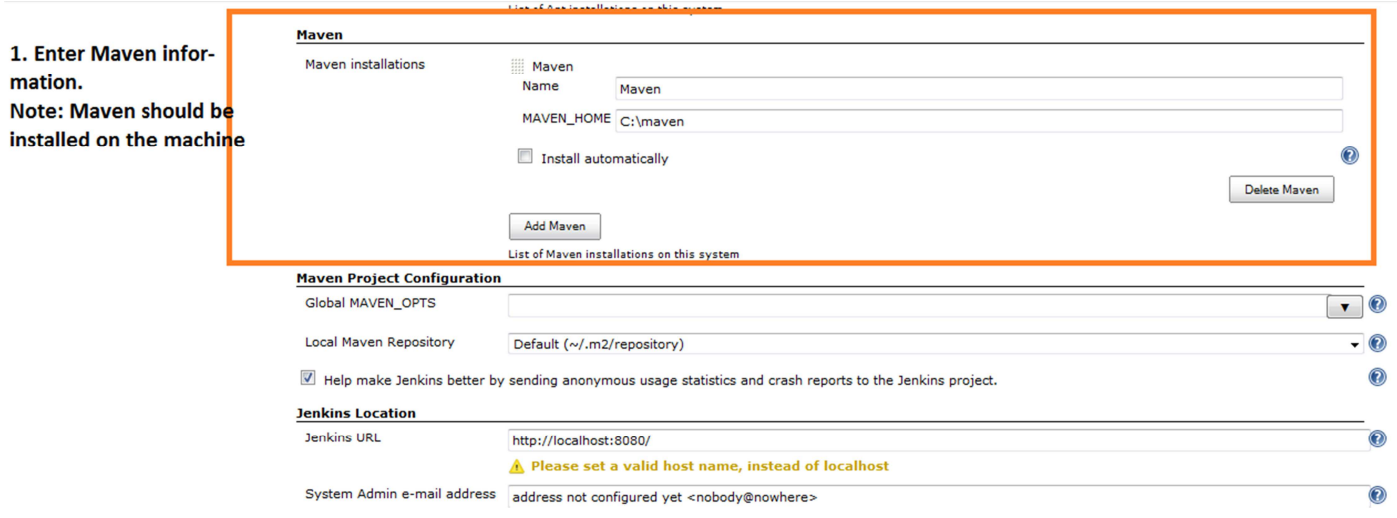
      a. Go to Manage Jenkins -> Configure System



b. Enter JDK information:



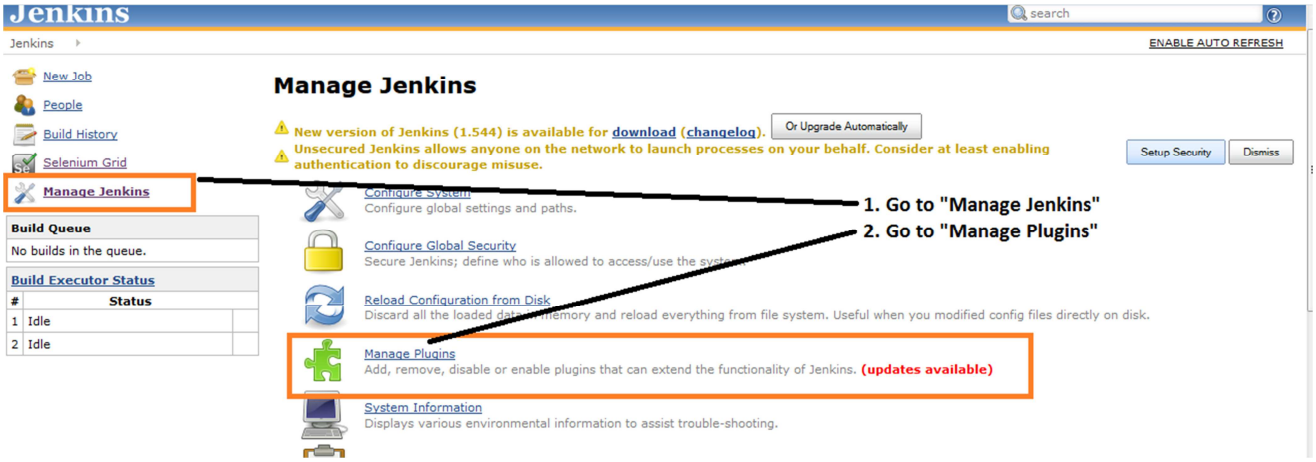
c. Enter Maven information:



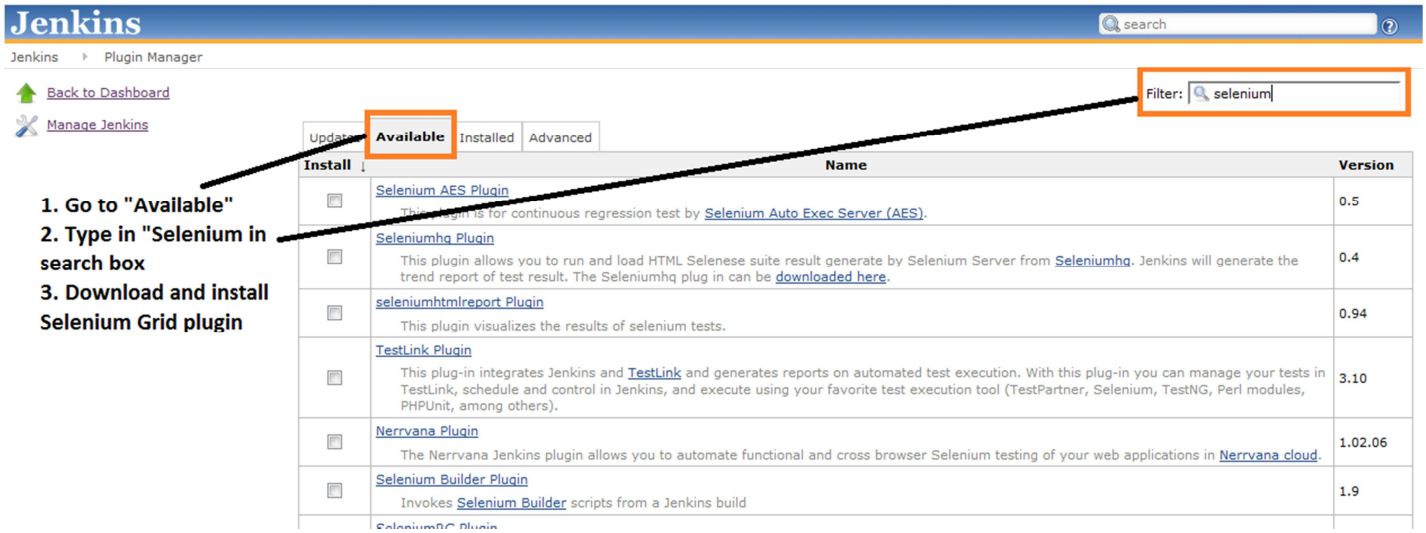
d. Click “Save” at the bottom after the above steps are done.

ii. Install Selenium Grid plugin – it’s very convenient as it runs as hub and you do not need to start the hub separately you will also be able to see nodes registered to the hub.

a. Go to “Manage Jenkins” -> “Manage Plugins”

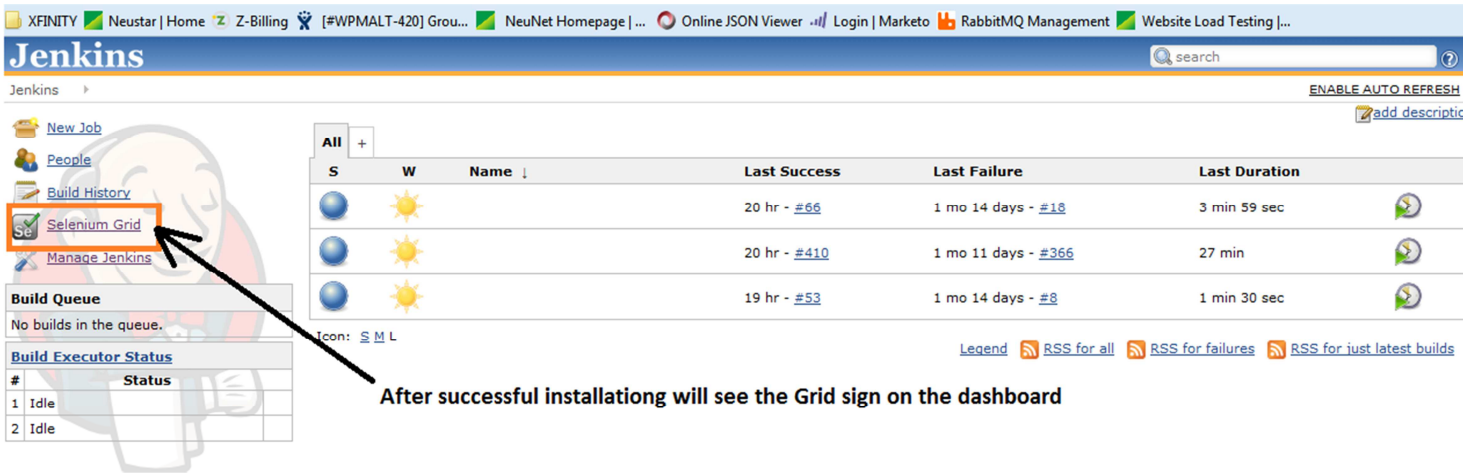


b. Click “Available” tab and type “Selenium” into search box



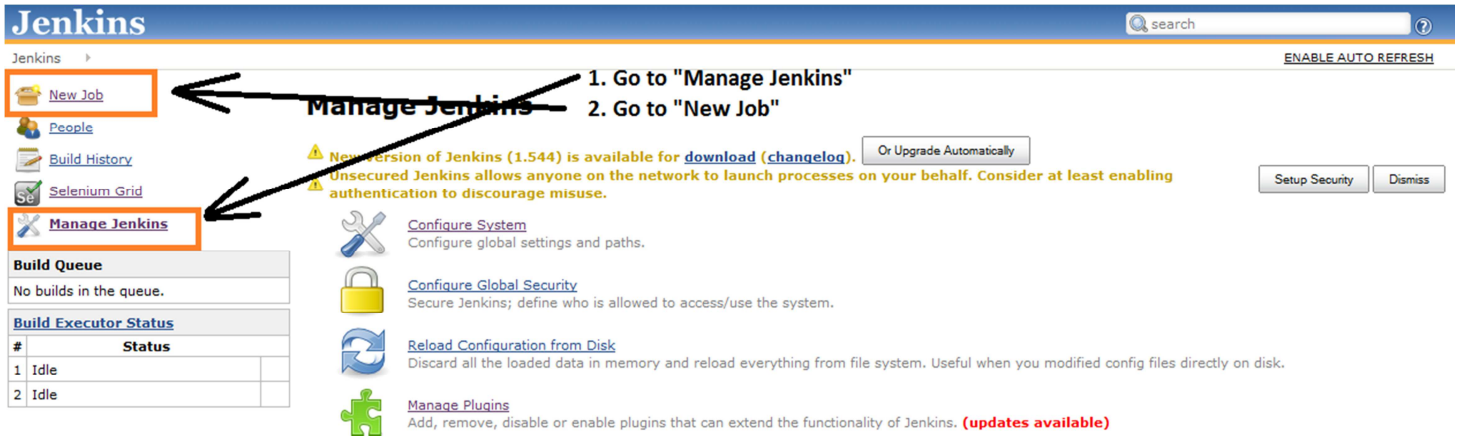
c. Download and install the plugin

d. After the plugin is installed you will see a “Se” sign on the dashboard:

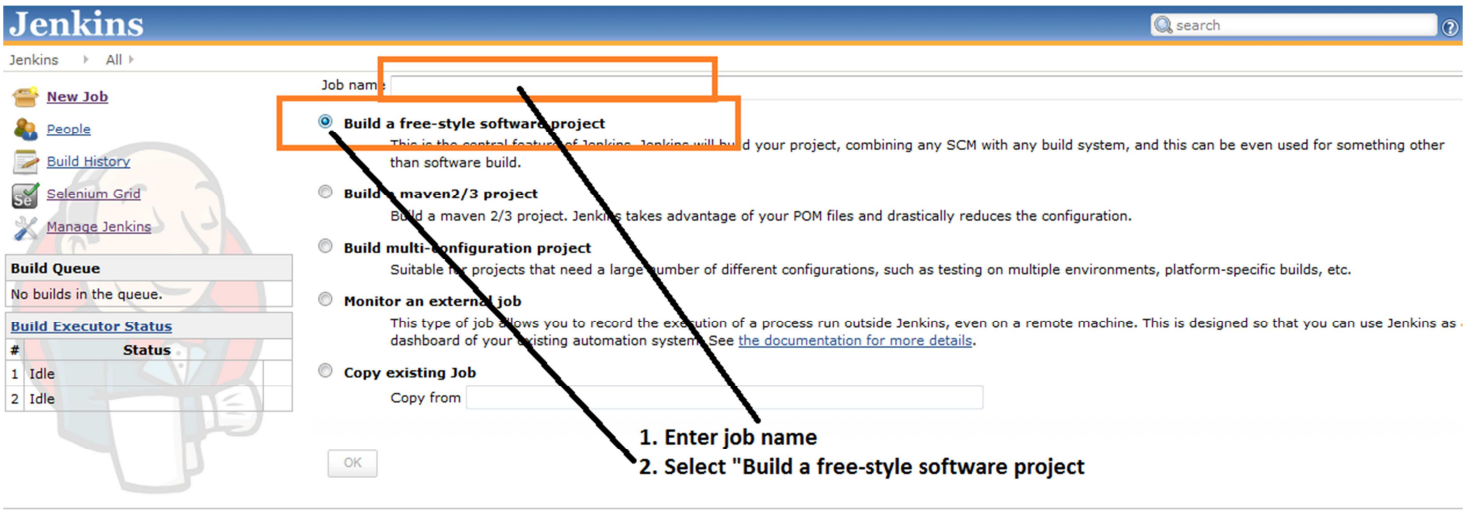


III. Create a new Job

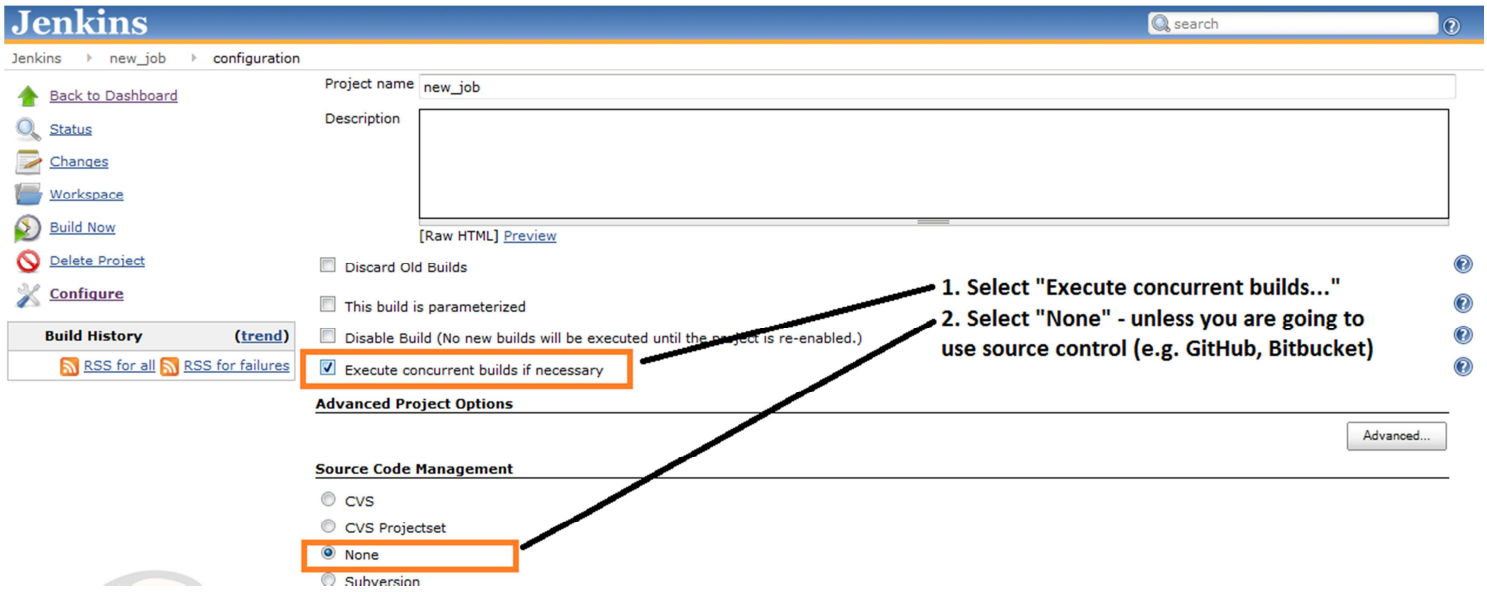
a. Go to “Manage Jenkins” -> “New job”



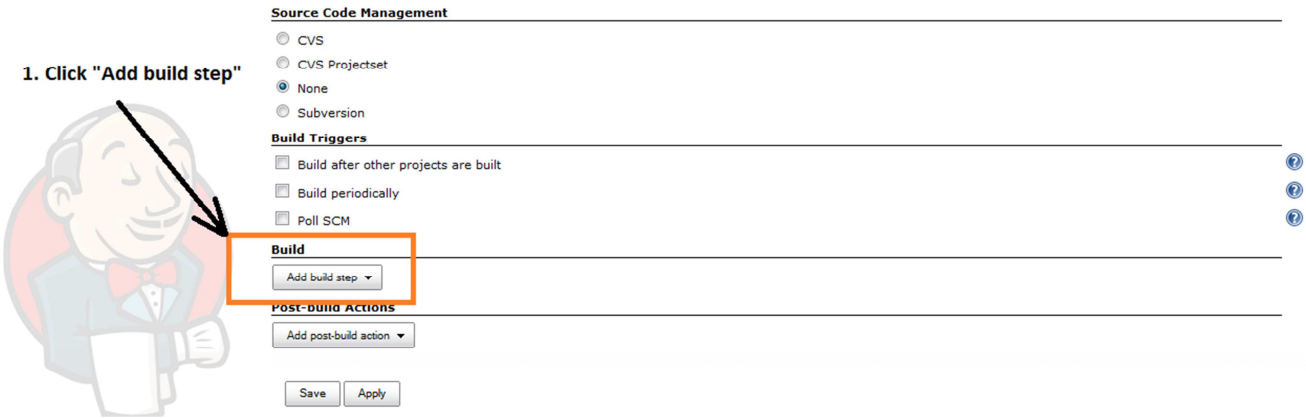
b. Select “Build a free-style software project”



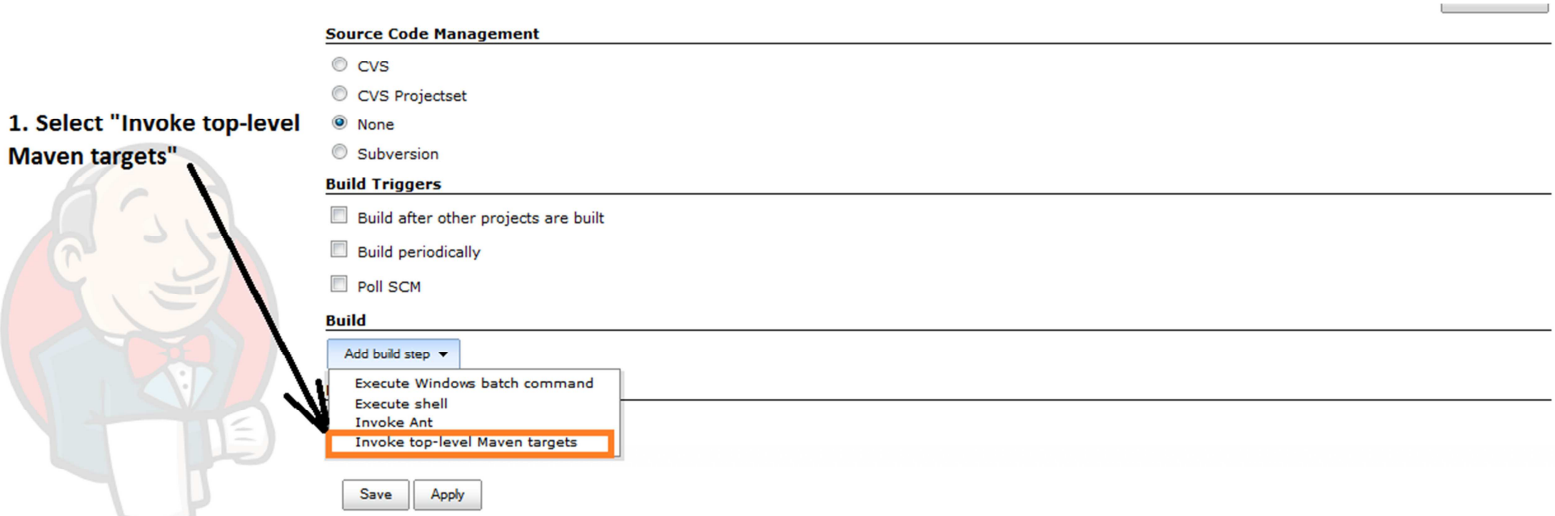
c. Click “Execute concurrent builds if necessary” and select “None” for version control (unless you want to use a repository for your tests)



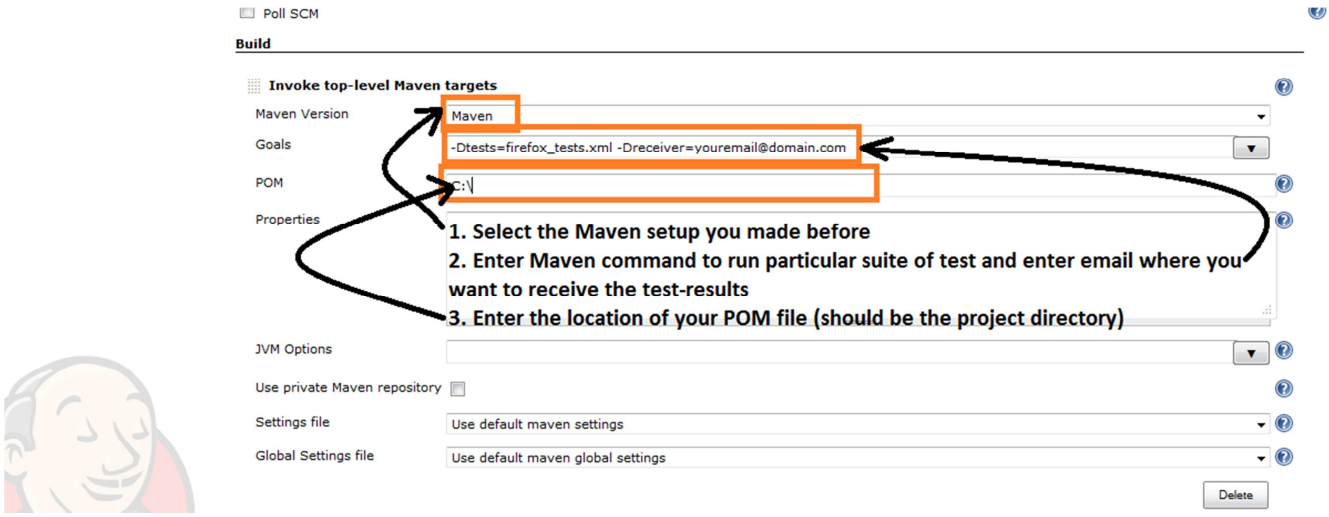
d. Click “Add build step”



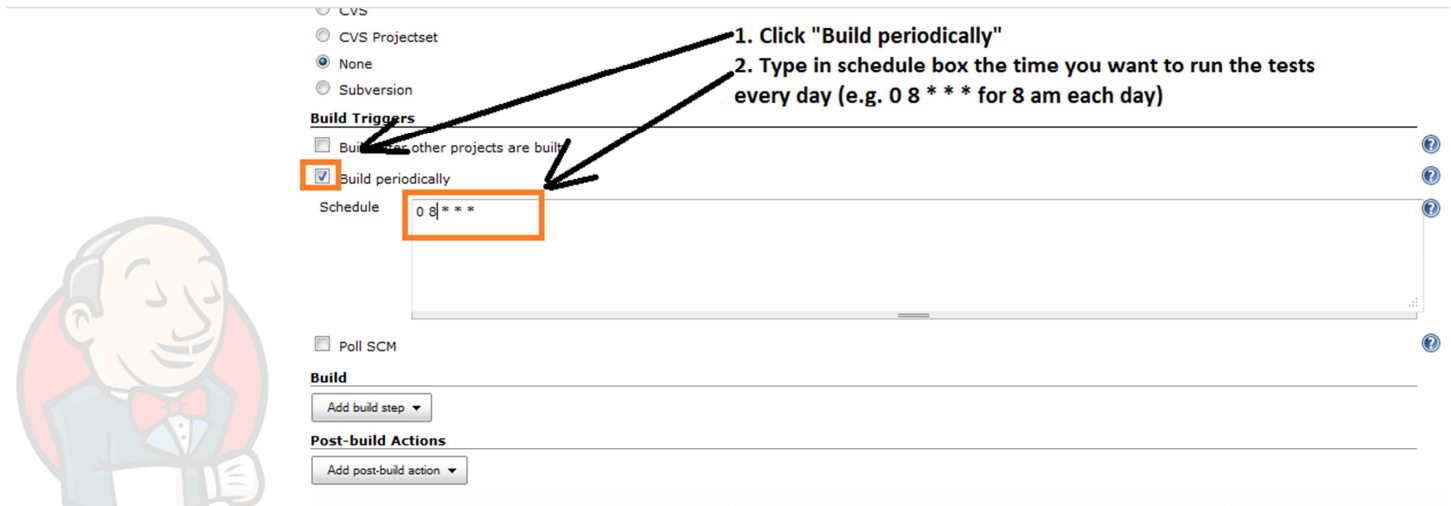
e. Select “Invoke top-level Maven targets”



f. Follow the steps from the screenshot below:



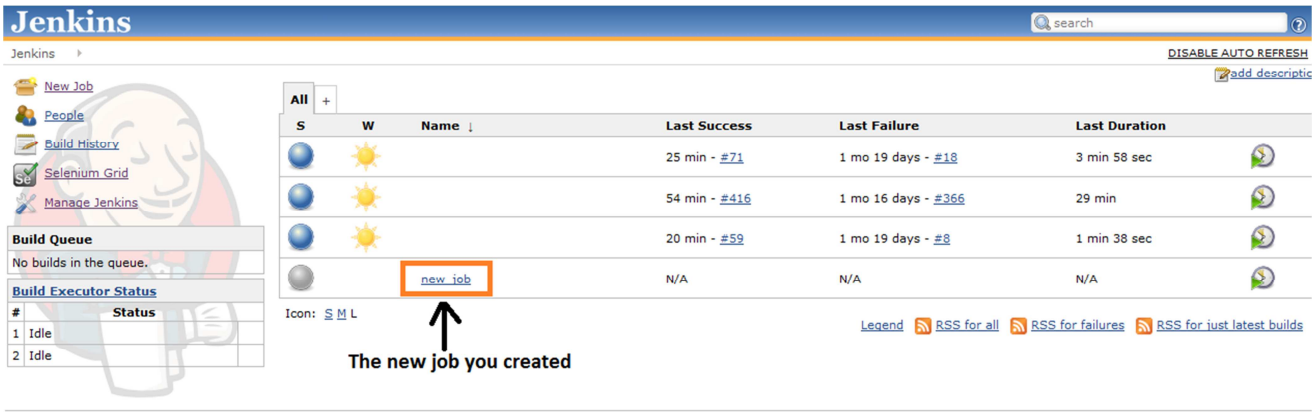
g. Follow the steps from screenshot below:



h. Click “Save” at the bottom

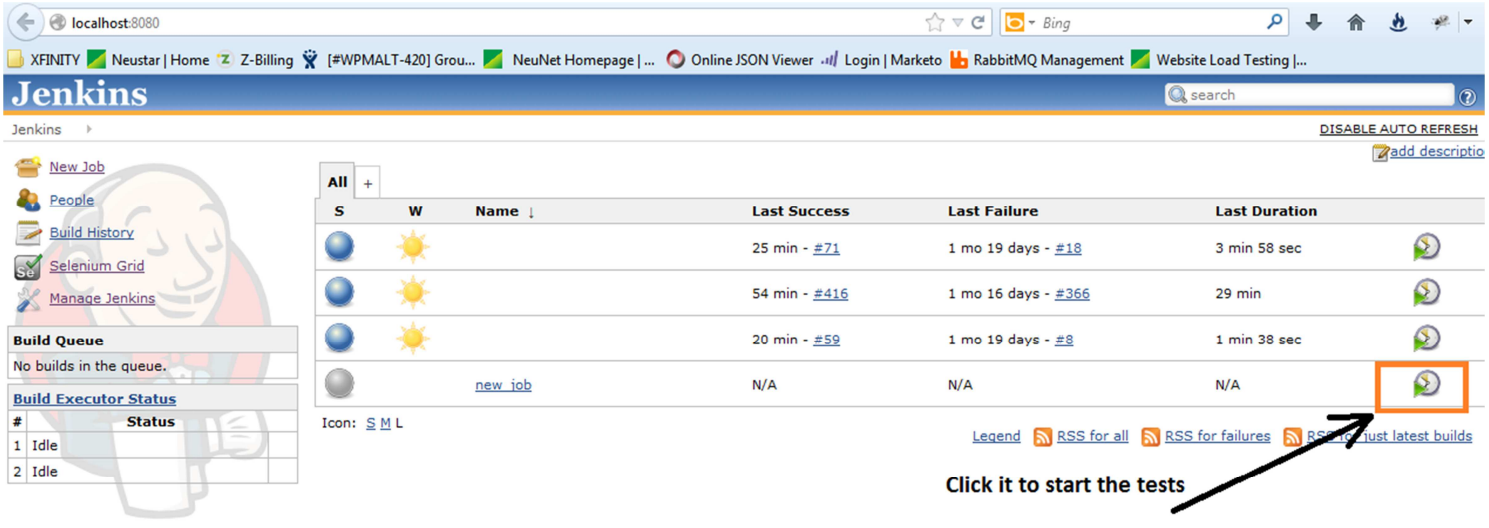
7. Running tests

If you followed all the steps from above you will see a new job created on Jenkins dashboard:



Start the node on your local machine or on your node by clicking thc .cmd file you created before and make sure node is connected.

The job is setup to run at a certain time but you can also start it by clicking the “triangle” sign in the right corner:



After the tests run, you will get a report with screenshots (if there are any failures- the tests in the starter kit will have some failures on purpose) to the email you've provided to the Maven command in Jenkins.

Test	Methods Passed	Scenarios Passed	# skipped	# failed	Total Time	Included Groups	Excluded Groups
Test III	2	2	0	1	18.1 seconds		
Test II	2	2	0	1	21.2 seconds		
Test I	1	1	0	0	22.5 seconds		
Total	**5**	**5**	**0**	**2**	**22.5 seconds**		

Class	Method	# of Scenarios	Start	Time (ms)
Test III — failed				
tests.VerifyPagesLoadFine	verifyLoadTestingToolsPageLoadsFine	1	1387905896087	1307
Test III — passed				
tests.VerifyPagesLoadFine	veifyLoadTestPageLoadsFine	1	1387905894039	1101
	verifyPerformancePageLoadsFine	1	1387905895144	942
Test II — failed				

All the tests in the starter kit have thorough comments of all the steps performed and you can basically start creating your automation framework by modifying the example tests- this way you will follow the structure of current framework.

*Disclaimer: all the tests in the repository for this Guide were configured using the following:

selenium-server-standalone-2.31.0.jar, firefox 22, ie 9, chrome 31.0.1650.63 m

And finally you will ask why automation is easy and fun? Well, first of all once the structure of your framework is created you will need just to add new tests and reuse most of the code you have in your page classes, the fun part is that while your tests are running and the job is being done – you can have a cup of coffee meanwhile ☺.