Created by: Binyamin refaelov

Maven project creation

Eclipse -> File -> New -> Other -> Maven -> create a simple project (Checkmark) -> Next->

Give name to project:

Group Id: example.ea

Artifact Id: testSkeleton

Please, now press alt+enter on the newly created project

Copy the location of the project, we will use it shortly.

How do we start with github:

https://www.youtube.com/watch?v=LPT7v69guVY

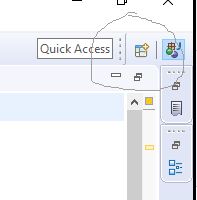
Please enter your github credentials

Create new project there:

Give a name at your will.

Now you got a name for your project, what you do with it?

Please press the following link in eclipse:



This is the perspective window, please press the “Git repositories” option.

You can also search for it, this will add a secondary screen on the bottom left of the screen:

Press on: CLONE GIT REPOSITORY

Add the project git link you received from github, for example:

<https://github.com/thedarkman123/autotemplate.git>

Please add all the authentication details you see further down the page, and press next and finish.

Now please press the right click on the project, go to

TEAM -< SHARE PROJECT

Choose The current project repository and press finish.

Do your first commit by going to your “team” option.

Next go to your pom.xml file, add some dependencies, for example:

Please be aware, we will explain the use of each one when It comes to that, but meanwhile, please check for the version of each one to be updated.

Now a bit about the selenium infrastructure:

Main thing we need is this few:

* Selenium Web Driver API Wrapper
* Selenium Grid
* Selenium Node

What is selenium grid:

* Central server for selenium nodes - allows us to connect many nodes to one server if we want to run 100-1000’s of tests concurrently
* Receive calls and send them to nodes
* Default port is 4444
* Can be configured using a json file for complex configurations
* Extensions are available

Where to download:

What we need is to download the selenium standalone server from:

<https://www.seleniumhq.org/download/>

Now, copy the file that you downloaded to a more convenient place and open cmd there with the following command:

* java -jar selenium-server-standalone-3.6.0.jar -role hub -port 4444
* It is possible to run the hub as a windows service
* Access the Grid console:
* http://%hostname%:4444/grid/console

Please remember you can create a .bat file with this command written behind it.

Please look at the newly created grid to understand where you should attach your nodes.

After, we should add new nodes to it, right now should be empty.

* java -jar selenium-server-standalone-2.53.1.jar -role node -hub http://localhost:4444/grid/register Dwebdriver.chrome.driver=chromedriver.exe

More options:

-port - set the  node port

-MaxSessions - maximal number of browsers sessions (all browsers on node

-Max instances - maximal number of a specific  browser isntances per node

One important note:

Where you selenium server is, so should your drivers for it should be:

Chrome driver, firefox driver, ie driver and so on.

Example can be found here:

https://github.com/thedarkman123/automation/tree/master/src/test/resources/executables

Instead of all this, docker can be a good solution/alternative.

Now it’s time to build the framework,

This is the structure:

Under src/test/java create the following packages:

pageobjects

testcases

utilties

listeners

Under src/test/resources create the following packages:

Excel

Runner

Logs

Properties

Executables

Create 2 files under logs package:

Application.log

Selenium.log

Create 2 properties files:

OR. properties

Config.properties

The first file we are going to add to our project will be a WebDriverWrapper, it should be under our utils folder, the more extensive it is, the better, it should give us a nice interface to communicate with selenium webdriver actions:

A good example for one is found here:

<https://github.com/thedarkman123/automation/blob/master/src/test/java/utilities/WebDriverWrapper.java>

Every bit from selenium should be added here for further use.

Now we should create our basetest class, and out pageobjects class, and we will make our first test to see all is working.

In testcases, we will make a BaseTest Class Which will be our initialization

Of many of out working components, one of them is our wrapper, example:

https://github.com/thedarkman123/automation/blob/master/src/test/java/testcases/BaseTest.java

next is a generic page object referenced here, should be for tests that aren’t related to specific page:

<https://github.com/thedarkman123/automation/blob/master/src/test/java/pageobjects/GenericPageObject.java>

under utilities we will have a small wrapper for properties file too, like this example right here:

<https://github.com/thedarkman123/automation/blob/master/src/test/java/utilities/PropertiesWrapper.java>

properties are written like this:

key=somevalue

examples are everywhere

Next is to create a runner file:

Before that we need to add a new software via eclipse, only this one is related to testNG,we do it as the following:

On eclipse press - > Help ->install new software->

Copy this to the url: <http://dl.bintray.com/testng-team/testng-eclipse-release/> ->

If nothing happens press ADD -> after, we should add the testNG bit, so our eclipse will recognize it, and add this to our project.

After, we should press right click on the project and we will see another Testng sub-menu:

Convert to testNG:

Here we will see a stub for a runner:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Suite">

<test thread-count="5" name="Test">

<classes>

<class name="testcases.testTest"/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

After comes the definitions of classes and the tests:

Suite name is not that important buy we should change it according to our need/desire, next comes the test declaration, in the test tag, in name = we should write the name of the class (the tests class, not the base one)

For example:

<test name="My first Tests">

Next should be the declaration of the specific class,which in it are the test cases we want to initiate.

For example: (name of package,name of class)

<class name="testcases.TestTest"/>

Now we can run the maven file by right click on it

(Beware: if any issues arise, we should update the maven and clean the project, after, things will work again)

Next: logs!

<https://www.guru99.com/tutorial-on-log4j-and-logexpert-with-selenium.html>

Let’s start with configuration

Create a file for a log4j.properties in the properties folder, we need to create 2 separate files (which we already previously configured)

Selenium logger:

log4j.rootLogger=DEBUG,file

log4j.appender.file=org.apache.log4j.RollingFileAppender

log4j.appender.file.File=D:\\Guru99\\src\\Selenium.logs

log4j.appender.file.maxFileSize=900KB

log4j.appender.file.maxBackupIndex=5

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c<strong>**{1}**</strong>:%L - %m%n

log4j.appender.file.Append=false

Application logger:

og4j.logger.devpinoyLogger=DEBUG, dest1

log4j.appender.dest1=org.apache.log4j.RollingFileAppender

log4j.appender.dest1.maxFileSize=900KB

log4j.appender.dest1.maxBackupIndex=6

log4j.appender.dest1.layout=org.apache.log4j.PatternLayout

log4j.appender.dest1.layout.ConversionPattern=%d{dd/MM/yyyy HH:mm:ss} %c %m%n

log4j.appender.dest1.File=D:\\ Guru99\\src\\Manual.logs

log4j.appender.dest1.Append=false

Now we need to configure a relative path, so the file will write itself in the correct relative location

Example:

.\\src\\test\\resources\\logs\\Application.log

We only need to configure the initialization in the basetest file and we can use the logger to write our debugs using the method debug()

That is it.

From this moments, we can add assertions as we possibly want, no problems.

Some great info about xpath:

<https://devhints.io/xpath>

The way to check xpath in web console is:

$x("//div[@class='pinwrapper']");

Some great info about testng as a whole:

Including functionality, assertions

<https://www.swtestacademy.com/testng-tutorial/>

Some important note:

We are using the page object scheme, which means that each test step is written in the page object it belong to as follows:

Example for a pageobject:

**package** pageobjects;

**public** **class** PageObject {

**public** **static** **class** ExamplePage1 **extends** GenericPageObject{

**public** **static** **void** goToPage() {

System.***out***.println("all the needed info");

}

**public** **static** **void** doSomeStuffOnThePage() {

System.***out***.println("all the needed info");

}

}

**public** **static** **class** ExamplePage2 **extends** GenericPageObject{

**public** **static** **void** goToPage() {

System.***out***.println("The next page stuff here");

}

**public** **static** **void** doSomeStuffOnThePage() {

System.***out***.println("The great stuff here");

}

}

}

And in the test cases, we need to use them and do assertions there, not in the pageObject.

//test site https://btc-gain.com/

A great method to work with to check if xhr ended it’s thing is to check the readystate, read here:

<https://stackoverflow.com/questions/33348600/selenium-wait-for-ajax-content-to-load-universal-approach>

This is specific for a case where all the site is reloading after a certain action from the user which doesn’t go to another page but stays the same.

Now, dataproviders,on this site for sake of example:

<https://btc-gain.com/?a=signup>

Dataproviders are great for a page where we should enter many values to fields, one after another.

The anotation is @dataProvider(name=”nameforit”)

And after the syntax is as follows:

public static **Object**[][] loginData () {

*//Userdata is formatted as username and password*

**return** **new** **Object**[][] { { "user1", "pass1" }, { "user2", "pass2" }};

}

When I call a test with the dataprovider, the test will happen as much time as the data provided for it

We will take info not from an object, but from an excel:

First create a folder and a file for excel which we will call it testdata (it really doesn't matter)

We added an excel utility:

<https://github.com/thedarkman123/automation/blob/master/src/test/java/utilities/ExcelReader.java>

next look at the dataProvider class:

<https://github.com/thedarkman123/automation/tree/master/src/test/java/dataprov>

And the use:

For example:

@Test(dataProvider="excelData",dataProviderClass=Dataproviders.**class**)

**public** **void** firstTest(String firstName,String lastName) {

System.***out***.println("firstname: "+firstName);

System.***out***.println("lastname: "+lastName);

}

Now let's add some listeners for reportng:

We need to add to the testng.xml the following:

<listeners>

<listener class-name=*"org.uncommons.reportng.HTMLReporter"*></listener>

<listener class-name=*"org.uncommons.reportng.JUnitXMLReporter"*></listener>

</listeners>

Now when we run the tests, we will have another output of report in the html->index.html on the folder; test-output

Now add custom listeners: