**DAY 1 – Cucumber HTML Report and Extent Report**

**Topics Covered**

1. Cucumber HTML Report and Extent Report
2. Selenium Waits
3. Page Object  Model
4. TestNG framework questions clarification
5. Cucumber framework question clarification
6. Java questions clarification
7. Selenium questions clarification.

**Create HTML reports in Cucumber**

Imagine that you have to share the test reports with your client and senior management; in that case you will need a shareable HTML report which you can share after executing your tests.

You can achieve this by following some very simple steps.

**Create an HTML report by adding a plugin to testrunner.java class**

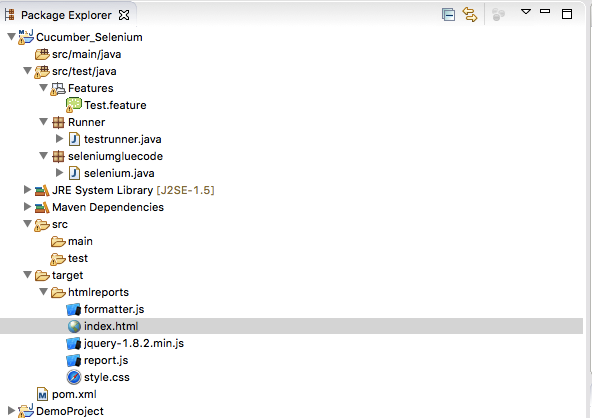
**Step 1:** In your testrunner.java class, add a plugin inside @CucumberOptions to format your test results into the HTML format.

**plugin = { "pretty", "html:target/htmlreports" }**

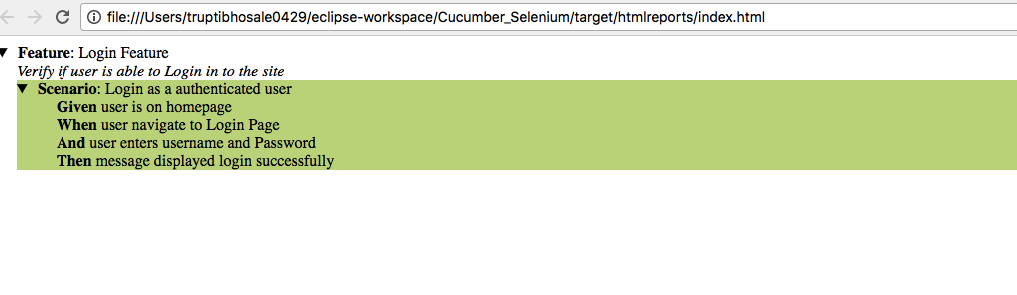
In order to set the path for the reports, we have to give a path in the project. To make this easier, the path is target/htmlreports.

**Step 2:** Now save the testrunner.java class and execute it. On execution, you will see that the folder htmlreports is created inside the target folder.

**Step 3:** Access the folder and look for the index.html file; that is the file which contains the test results in HTML format.



**Step 4:** Open the index.html to view the report. The report created would be similar to the image below.



**Create HTML report by using extent-reports**

We have already seen how to create an HTML test report, but with the help of extent reports we can create more well-organized and detailed reports.

**Step 1:** To implement extent report, we need to add two dependencies to the pom.xml and update the project after adding the dependency.

*Cucumber-extentsreport*  
*extentreports*

The dependencies for the above would be like this:

<dependency>

<groupId>com.vimalselvam</groupId>

<artifactId>cucumber-extentsreport</artifactId>

<version>3.0.2</version>

</dependency>

<dependency>

<groupId>com.aventstack</groupId>

<artifactId>extentreports</artifactId>

<version>3.1.2</version>

</dependency>

**Step 2:** Add a new folder to the project. Eg. “***config***” by *right clicking the project folder → New → Folder → Config*. Now we have to add an XML file to this folder. This XML file states the theme of the report, title, etc. The report.xml file would be like this:

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

<extentreports>

<configuration>

<!-- report theme --> <!-- standard, dark -->

<theme>standard</theme>

<!-- document encoding --> <!-- defaults to UTF-8 -->

<encoding>UTF-8</encoding>

<!-- protocol for script and stylesheets --> <!-- defaults to https -->

<protocol>https</protocol>

<!-- title of the document -->

<documentTitle>Selenium Cucumber Framework</documentTitle>

<!-- report name - displayed at top-nav -->

<reportName>Functional Testing report</reportName>

<!-- global date format override --> <!-- defaults to yyyy-MM-dd -->

<dateFormat>yyyy-MM-dd</dateFormat>

<!-- global time format override --> <!-- defaults to HH:mm:ss -->

<timeFormat>HH:mm:ss</timeFormat>

<!-- custom javascript -->

<scripts>

<![CDATA[

$(document).ready(function() {

});

]]>

</scripts>

<!-- custom styles -->

<styles>

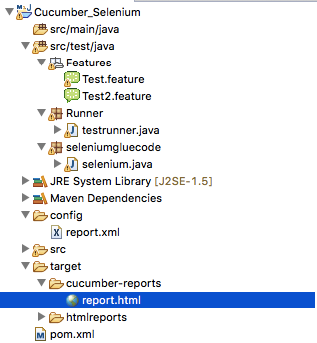
<![CDATA[

]]>

</styles>

</configuration>

</extentreports>



**Step 3:** Now we are almost ready with the setup required for the report, but in order to fetch the report for every test, we need to add a plugin in testrunner.java and add an @AfterClass. In the plugin, we will mention the Extent formatter and the location where we want the report to be saved, and in the after class, we will write a function to load the report.xml. The final testrunner.java class would be like this:

**package** Runner;

**import** **java.io.File**;

**import** **org.junit.AfterClass**;

**import** **org.junit.runner.RunWith**;

**import** **com.cucumber.listener.ExtentCucumberFormatter**;

**import** **com.cucumber.listener.Reporter**;

**import** **cucumber.api.CucumberOptions**;

**import** **cucumber.api.junit.Cucumber**;

**@RunWith**(Cucumber.class)

**@CucumberOptions**(

features ="src/test/java/features"

,glue= "seleniumgluecode",

plugin = { "com.cucumber.listener.ExtentCucumberFormatter:target/cucumber-reports/report.html"},

monochrome = **true**

)

**public** **class** **testrunner** {

**@AfterClass**

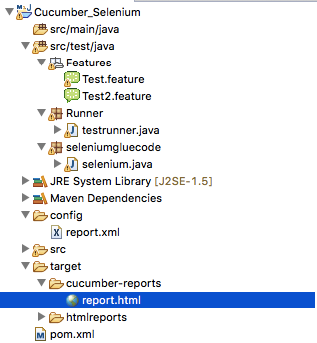
**public** **static** **void** **writeExtentReport**() {

Reporter.loadXMLConfig(**new** File("config/report.xml"));

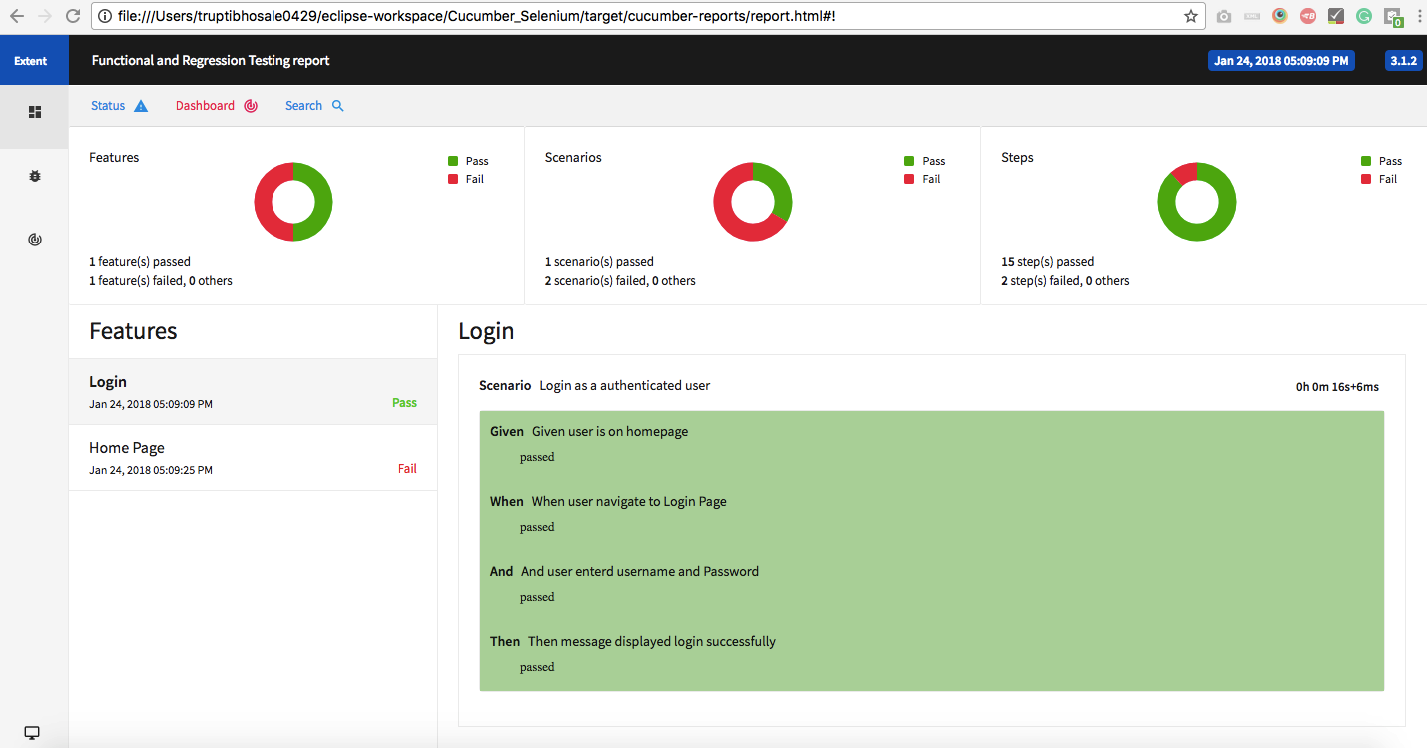
}

}

**Step 4:** On executing the tests, a new folder will be created at the path mentioned in the plugin. Open the folder and open the report.html.



The report created will have a heading, graph of the test results, detailed results for all features executed, and you can also filter the test results with the status Pass/Fail by clicking the Status menu.



## Implicit, Explicit and Fluent Waits in Selenium

Most of the web applications are developed using Ajax and Javascript. When a page is loaded by the browser the elements which we want to interact with may load at different time intervals.

Not only it makes this difficult to identify the element but also if the element is not located it will throw an "**ElementNotVisibleException**" exception. Using Waits, we can resolve this problem.

Let's consider a scenario where we have to use both implicit and explicit waits in our test. Assume that implicit wait time is set to 20 seconds and explicit wait time is set to 10 seconds.

Suppose we are trying to find an element which has some **"ExpectedConditions** "(Explicit Wait), If the element is not located within the time frame defined by the Explicit wait(10 Seconds), It will use the time frame defined by implicit wait(20 seconds) before throwing an "**ElementNotVisibleException**".

**Selenium Web Driver Waits**

1. Implicit Wait
2. Explicit Wait

## Implicit Wait

Selenium Web Driver has borrowed the idea of implicit waits from Watir.

The implicit wait will tell to the web driver to wait for certain amount of time before it throws a "No Such Element Exception". The default setting is 0. Once we set the time, web driver will wait for that time before throwing an exception.

In the below example we have declared an implicit wait with the time frame of 10 seconds. It means that if the element is not located on the web page within that time frame, it will throw an exception.

To declare implicit wait:

**Syntax**:

driver.manage().timeouts().implicitlyWait(TimeOut, TimeUnit.SECONDS);

package guru.test99;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import org.testng.annotations.Test;

public class AppTest {

protected WebDriver driver;

@Test

public void guru99tutorials() throws InterruptedException

{

System.setProperty ("webdriver.chrome.driver",".\\chromedriver.exe" );

driver = new ChromeDriver();

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

String eTitle = "Demo Guru99 Page";

String aTitle = "" ;

// launch Chrome and redirect it to the Base URL

driver.get("http://demo.guru99.com/test/guru99home/" );

//Maximizes the browser window

driver.manage().window().maximize() ;

//get the actual value of the title

aTitle = driver.getTitle();

//compare the actual title with the expected title

if (aTitle.equals(eTitle))

{

System.out.println( "Test Passed") ;

}

else {

System.out.println( "Test Failed" );

}

//close browser

driver.close();

}

}

**Explanation of Code**

In the above example,

**Consider Following Code:**

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

Implicit wait will accept 2 parameters, the first parameter will accept the time as an integer value and the second parameter will accept the time measurement in terms of SECONDS, MINUTES, MILISECOND, MICROSECONDS, NANOSECONDS, DAYS, HOURS, etc.

## Explicit Wait

The explicit wait is used to tell the Web Driver to wait for certain conditions (**Expected Conditions**) or the maximum time exceeded before throwing an "**ElementNotVisibleException**" exception.

The explicit wait is an intelligent kind of wait, but it can be applied only for specified elements. Explicit wait gives better options than that of an implicit wait as it will wait for dynamically loaded Ajax elements.

Once we declare explicit wait we have to use "**ExpectedCondtions**" or we can configure how frequently we want to check the condition using **Fluent Wait**. These days while implementing we are using **Thread.Sleep()**generally it is not recommended to use

In the below example, we are creating reference wait for "**WebDriverWait**" class and instantiating using "**WebDriver**" reference, and we are giving a maximum time frame of 20 seconds.

**Syntax:**

WebDriverWait wait = new WebDriverWait(WebDriverRefrence,TimeOut);

package guru.test99;

import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.support.ui.ExpectedConditions;

import org.openqa.selenium.support.ui.WebDriverWait;

import org.testng.annotations.Test;

public class AppTest2 {

protected WebDriver driver;

@Test

public void guru99tutorials() throws InterruptedException

{

System.setProperty ("webdriver.chrome.driver",".\\chromedriver.exe" );

driver = new ChromeDriver();

WebDriverWait wait=new WebDriverWait(driver, 20);

String eTitle = "Demo Guru99 Page";

String aTitle = "" ;

// launch Chrome and redirect it to the Base URL

driver.get("http://demo.guru99.com/test/guru99home/" );

//Maximizes the browser window

driver.manage().window().maximize() ;

//get the actual value of the title

aTitle = driver.getTitle();

//compare the actual title with the expected title

if (aTitle.contentEquals(eTitle))

{

System.out.println( "Test Passed") ;

}

else {

System.out.println( "Test Failed" );

}

WebElement guru99seleniumlink;

guru99seleniumlink= wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath( "/html/body/div[1]/section/div[2]/div/div[1]/div/div[1]/div/div/div/div[2]/div[2]/div/div/div/div/div[1]/div/div/a/i")));

guru99seleniumlink.click();

}

}

**Explanation of Code**

**Consider Following Code:**

WebElement guru99seleniumlink;

guru99seleniumlink = wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("/html/body/div[1]/section/div[2]/div/div[1]/div/div[1]/div/div/div/div[2]/div[2]/div/div/div/div/div[1]/div/div/a/i")));

guru99seleniumlink.click();

In the above example, wait for the amount of time defined in the "**WebDriverWait**" class or the "**ExpectedConditions**" to occur whichever occurs first.

The above[Java](https://www.guru99.com/java-tutorial.html)code states that we are waiting for an element for the time frame of 20 seconds as defined in the "**WebDriverWait**" class on the webpage until the "**ExpectedConditions**" are met and the condition is "**visibilityofElementLocated**".

The following are the Expected Conditions that can be used in Explicit Wait

1. alertIsPresent()
2. elementSelectionStateToBe()
3. elementToBeClickable()
4. elementToBeSelected()
5. frameToBeAvaliableAndSwitchToIt()
6. invisibilityOfTheElementLocated()
7. invisibilityOfElementWithText()
8. presenceOfAllElementsLocatedBy()
9. presenceOfElementLocated()
10. textToBePresentInElement()
11. textToBePresentInElementLocated()
12. textToBePresentInElementValue()
13. titleIs()
14. titleContains()
15. visibilityOf()
16. visibilityOfAllElements()
17. visibilityOfAllElementsLocatedBy()
18. visibilityOfElementLocated()

## Fluent Wait

The fluent wait is used to tell the web driver to wait for a condition, as well as the **frequency**with which we want to check the condition before throwing an "ElementNotVisibleException" exception.

**Frequency:**Setting up a repeat cycle with the time frame to verify/check the condition at the regular interval of time

Let's consider a scenario where an element is loaded at different intervals of time. The element might load within 10 seconds, 20 seconds or even more then that if we declare an explicit wait of 20 seconds. It will wait till the specified time before throwing an exception. In such scenarios, the fluent wait is the ideal wait to use as this will try to find the element at different frequency until it finds it or the final timer runs out.

**Syntax:**

Wait wait = new FluentWait(WebDriver reference)

.withTimeout(timeout, SECONDS)

.pollingEvery(timeout, SECONDS)

.ignoring(Exception.class);

Above code is deprecated in Selenium v3.11 and above. You need to use

Wait wait = new FluentWait(WebDriver reference)

.withTimeout(Duration.ofSeconds(SECONDS))

.pollingEvery(Duration.ofSeconds(SECONDS))

.ignoring(Exception.class);

package guru.test99;

import org.testng.annotations.Test;

import java.util.NoSuchElementException;

import java.util.concurrent.TimeUnit;

import java.util.function.Function;

import org.openqa.selenium.By;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.support.ui.ExpectedConditions;

import org.openqa.selenium.support.ui.FluentWait;

import org.openqa.selenium.support.ui.Wait;

import org.openqa.selenium.support.ui.WebDriverWait;

import org.testng.annotations.Test;

public class AppTest3 {

protected WebDriver driver;

@Test

public void guru99tutorials() throws InterruptedException

{

System.setProperty ("webdriver.chrome.driver",".\\chromedriver.exe" );

String eTitle = "Demo Guru99 Page";

String aTitle = "" ;

driver = new ChromeDriver();

// launch Chrome and redirect it to the Base URL

driver.get("http://demo.guru99.com/test/guru99home/" );

//Maximizes the browser window

driver.manage().window().maximize() ;

//get the actual value of the title

aTitle = driver.getTitle();

//compare the actual title with the expected title

if (aTitle.contentEquals(eTitle))

{

System.out.println( "Test Passed") ;

}

else {

System.out.println( "Test Failed" );

}

Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)

.withTimeout(30, TimeUnit.SECONDS)

.pollingEvery(5, TimeUnit.SECONDS)

.ignoring(NoSuchElementException.class);

WebElement clickseleniumlink = wait.until(new Function<WebDriver, WebElement>(){

public WebElement apply(WebDriver driver ) {

return driver.findElement(By.xpath("/html/body/div[1]/section/div[2]/div/div[1]/div/div[1]/div/div/div/div[2]/div[2]/div/div/div/div/div[1]/div/div/a/i"));

}

});

//click on the selenium link

clickseleniumlink.click();

//close~ browser

driver.close() ;

}

}

**Explanation of Code**

**Consider Following Code:**

Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)

.withTimeout(30, TimeUnit.SECONDS)

.pollingEvery(5, TimeUnit.SECONDS)

.ignoring(NoSuchElementException.class);

In the above example, we are declaring a fluent wait with the timeout of 30 seconds and the frequency is set to 5 seconds by ignoring "**NoSuchElementException**"

**Consider Following Code:**

public WebElement apply(WebDriver driver) {

return driver.findElement(By.xpath("/html/body/div[1]/section/div[2]/div/div[1]/div/div[1]/div/div/div/div[2]/div[2]/div/div/div/div/div[1]/div/div/a/i"));

We have created a new function to identify the Web Element on the page. (Ex: Here Web Element is nothing but the selenium link on the webpage).

Frequency is set to 5 seconds and the maximum time is set to 30 seconds. Thus this means that it will check for the element on the web page at every 5 seconds for the maximum time of 30 seconds. If the element is located within this time frame it will perform the operations else it will throw an"**ElementNotVisibleException**"

**References –**

1. Maven Selenium Cucumber setup - <https://www.axelerant.com/resources/team-blog/setup-for-selenium-with-cucumber-using-maven>