**Debugging Selenium Scripts with Logs**

Now we are moving towards the end of our most comprehensive[**Free Modules for Selenium Testing tool**](http://www.softwaretestinghelp.com/selenium-tutorial-1/). The modules we are posting now are the part of advance Selenium training.

In the previous module, we kept our focus on the [Continuous Integration tool named as Hudson](http://www.softwaretestinghelp.com/hudson-continuous-integration-tool-selenium-tutorial-25/). It’s a free tool and has a lot more capabilities to build the testing project, execute the test classes remotely and send out a notification email across the stakeholders informing them about the application status with respect to passed and failed test cases.

In the current module, we would motion towards some advance concepts that would directly or indirectly help in **optimizing the Automation framework** and brings more visibility to the users.

Thus, in the current module, we would discuss about the **logging feature, its potential, debugging capabilities**and much more.

Sometimes logging is considered to be an overhead upon the existing script creation mechanism but experts considers it to be one of the best practices if used in the accurate proportion because of the following advantages:

**Advantages of Logging in Selenium Scripts:**

* Grants a complete understanding of test suites execution
* Log messages can be stored in external files for post execution scrutiny
* Logs are an exceptional assistant in debugging the program execution issues and failures
* Logs can also be reviewed to ascertain the application’s health by the stakeholders

**Log4j – A Java based Logging API**

Moving on to the technical details about logging, let us discuss the origin of the API that we would be using throughout the **log4j module** to generate logs. Log4j was a result of collaborative efforts of people at Secure Electronic Marketplace for Europe to develop a utility that would help us generating logs and hence the log4j came into limelight in the year 1996. Log4j is an open source tool and licensed under IBM Public License.

There are three main components that constitute the implementation of log4j. These components represent the details about the log level, formats of the log message in which they would be rendered and their saving mechanisms.

### Constituents of Log4j

1. Loggers
2. Appenders
3. Layouts

**#1) Loggers**

The following steps need to done in order to implement loggers in the project.

**Step 1**: Creating an instance of Logger class

**Step 2**: Defining the log level

**Logger Class** – It is a java based utility that has got all the generic methods already implemented so that we are enabled to use log4j.

**Log levels** – Log levels are popularly known as printing methods. These are used for printing the log messages. There are primarily five kinds of log levels.

* error()
* warn()
* info()
* debug()
* log()

Thus, to be able to generate logs, all we need to do is to call any of the printing method over the logger instance. We will have a broader look into it during the implementation phase.

**#2) Appenders**

Now that we know how to generate these logs, the next thing that should pop up into our minds is that where do I get to view the logs? The answer to this question lies in the definition of “Appenders”.

Appenders are consistently used to specify the data source/medium where the logs should be generated. The scope of data sources stretches from various external mediums like console, GUI, text files etc.

**#3) Layouts**

At times, user wishes certain information to be pre – pended or appended with each log statement. For example I wish to a print a timestamp along with my log statement. Thus, such requirements can be accomplished by “Layouts”.

Layouts are a utility that allows the user to opt for a desired format in which the logs would be rendered. Appenders and Layout have a tight coupling between them. Thus, we are required to map each of the appender with a specific layout.

Take a note that user is leveraged to define multiple appenders, each mapped with a distinct layout.

Now that we are aware of the basics of log4j and its components, we shall motion our focus towards the implementation phenomenon.

**Let us understand the entire implementation process step by step.**

### Installation/Setup

For the installation and setup, we would be considering “Learning\_Selenium” project that we have already created in the earlier sessions of this series.

**Step 1**: The first and the foremost step is to download the latest jar for log4j API. The jar can be easily found at its official distribution website – “<http://logging.apache.org/log4j/1.2/download.html>”.

**Step 2**: The next step is to configure the build path and provide log4j.jar as an external library.

**Implementation**

**Logging using log4j can be implemented and configured in namely two ways:**

1. Programmatically via script
2. Manually via Configuration files

Both the above mentioned configuration methods have merit as well as demerits. For this module, we would consider configuring log4j manually via Configuration files based on its ease and simplicity. Configuration file is yet another xml file to configure artifacts related to log4j.

### ****Creation of log4j.xml file****

**Step 1**. Create a log4j.xml file. Copy and paste the code below in the configuration file.

|  |  |
| --- | --- |
| 1 | <?xml version="1.0" encoding="UTF-8" ?> |
| 2 | <!DOCTYPE log4j:configuration SYSTEM "log4j.dtd"> | |

|  |  |
| --- | --- |
| 3 | <log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"> |
| 4 | <appender name="consoleAppender"class="org.apache.log4j.ConsoleAppender"> | |

|  |  |
| --- | --- |
| 5 | <layout class="org.apache.log4j.PatternLayout"> |
| 6 | <!-- For Printing message with date , time & class name also | |

|  |  |  |
| --- | --- | --- |
| 7 | <param name="ConversionPattern" value="%d{dd MMM yyyy HH:mm:ss} %5p %c{1} - %m%n"/> | |
| 8 | --> |

|  |  |  |
| --- | --- | --- |
| 9 | <!-- For printing message only --> | |
| 10 | | <param name="ConversionPattern" value="%-5p[%c{1}]: %m%n"/> | |

|  |  |
| --- | --- |
| 11 | </layout> |
| 12 | </appender> | |

|  |  |  |
| --- | --- | --- |
| 13 | <appender name="fileAppender"class="org.apache.log4j.RollingFileAppender"> | |
| 14 | <param name="append" value="false"/> |

|  |  |  |
| --- | --- | --- |
| 15 | <param name="file"value="Logs/Webliv\_Automation\_Logs.log"/> | |
| 16 | <layout class="org.apache.log4j.PatternLayout"> |

|  |  |  |
| --- | --- | --- |
| 17 | <param name="ConversionPattern" value="%d{dd-MM-yyyy HH:mm:ss}%x %-5p[%c{1}]: %m%n"/> | |
| 18 | </layout> |

|  |  |  |
| --- | --- | --- |
| 19 | </appender> | |
| 20 | <root> |

|  |  |
| --- | --- |
| 21 | <level value="INFO"/> |
| 22 | <appender-ref ref="consoleAppender"/> | |

|  |  |  |
| --- | --- | --- |
| 23 | <appender-ref ref="fileAppender"/> | |
| 24 | </root> |

|  |  |
| --- | --- |
| 25 | </log4j:configuration> |

**Walkthrough of Configuration File**

**consoleAppender**

|  |  |  |
| --- | --- | --- |
| 1 | <appender name="consoleAppender"class="org.apache.log4j.ConsoleAppender"> | |
| 2 | <layout class="org.apache.log4j.PatternLayout"> |

The consoleAppender is used to print the log statements on the console.

------------

**fileAppender**

|  |  |  |
| --- | --- | --- |
| 1 |  | <appender name="fileAppender"class="org.apache.log4j.RollingFileAppender"> |

The fileAppender is used to print the log statements within an external file. User is leveraged to set an on and off value for the append tag which would tell the system to append and log statements to the previously created one or to overwrite the previously created logs and generate the fresh logs altogether.

|  |  |  |
| --- | --- | --- |
| 1 | <param name=<em>"append"</em> value=<em>"false"</em>/> | |
| 2 | <param name="file" value="Logs/TestLogs.log"/> |

The value of the file parameter is set to a particular location to notify the system to create the anticipated log file at the said location. We also specify the log file name within the value parameter.

**Layout**

|  |  |
| --- | --- |
| 1 | <layout class="org.apache.log4j.PatternLayout"> |
| 2 | <param name="ConversionPattern" value="%d{dd-MM-yyyy HH:mm:ss}%x %-5p[%c{1}]: %m%n"/> | |

|  |  |
| --- | --- |
| 3 | </layout> |

As discussed in the early sections of this module, the layouts are used to specify the rendering mechanism for log statements. Log4j provides various layout patterns. User is leveraged to specify the desired pattern in the value of ConversionPattern parameter.

**The output of the above layout should be something like:**

01-07-2014 12:56:32 INFO [GmailLogin]: Sample log message

In the output above:

* First field – Date of execution
* Second field – Exact time in hh:mm:ss at which the test step was executed
* Third field – One of the log level
* Fourth field – Name of the test class
* Fifth field – Log message

**Step 2**. As soon as we are done with the creation of log4j.xml file, the next step is to put the log4j.xml file into the project’s root folder/base directory.

**Program level Implementation**

**Step 3**: The next step is to use any of the configurator to configure and parse the log4j.xml file.

**Syntax:**

|  |  |
| --- | --- |
| 1 | package com.logExample; |
| 2 | import org.apache.log4j.xml.DOMConfigurator; | |

|  |  |
| --- | --- |
| 3 | import org.junit.AfterClass; |
| 4 | import org.junit.BeforeClass; | |

|  |  |  |
| --- | --- | --- |
| 5 | import org.junit.runner.JUnitCore; | |
| 6 | import org.junit.runner.Result; |

|  |  |
| --- | --- |
| 7 | import org.junit.runner.RunWith; |
| 8 | import org.junit.runner.notification.Failure; | |

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | import org.junit.runners.Suite; | | |
| 10 | | @RunWith(Suite.class) |

|  |  |
| --- | --- |
| 11 | @Suite.SuiteClasses({ |
| 12 | Demo.class | |

|  |  |  |
| --- | --- | --- |
| 13 | }) | |
| 14 | public class TestSuite { |

|  |  |
| --- | --- |
| 15 | /\*\* |
| 16 | \* Setup method to set system property for log file name | |

|  |  |
| --- | --- |
| 17 | \*/ |
| 18 | @BeforeClass | |

|  |  |
| --- | --- |
| 19 | public static void Setup() { |
| 20 | // loading log4j.xml file | |

|  |  |  |
| --- | --- | --- |
| 21 | DOMConfigurator.configure("log4j.xml"); | |
| 22 | } |

|  |  |
| --- | --- |
| 23 | /\*\* |
| 24 | \* @param args | |

|  |  |
| --- | --- |
| 25 | \*/ |
| 26 | public static void main(String[] args) { | |

|  |  |
| --- | --- |
| 27 | Result result = JUnitCore.runClasses(TestSuite.class); |
| 28 | for (Failure failure : result.getFailures()) { |

|  |  |
| --- | --- |
| 29 | System.out.println("\nTEST NAME: " + failure.getTestHeader()); |
| 30 | System.out.println("\nERROR: " + failure.getMessage() + "\n"); |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 | System.out.println(failure.getTrace()); | | |
| 32 | | System.exit(1); |

|  |  |  |
| --- | --- | --- |
| 33 | } | |
| 34 | } |

|  |  |
| --- | --- |
| 35 | } |

**Note**: Logs can be implemented at the class level also instead of Test suite level. All you need to do is to make the required changes in the test class rather than in the test suite.

**Step 4**: The very next step is to create a test class “GmailLogin.java” under the project. Implement the Gmail login functionality with in the class.

**Step 5**: The next step is to import the logger class to be able to implement the log statements.

**Syntax:**

import org.apache.log4j.Logger;

**Step 6**: The next step in the process is to instantiate the object of the Logger class.

**Syntax:**

//Object initialization for log

**static**Loggerlog= Logger.getLogger(Demo.**class**.getName());

**Step 7**: The above created log variable of type Logger would be used across the entire test class to generate the log statements. Refer the following code for the same.

**Syntax:**

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void testGmailLogin() throws Exception{ | |

|  |  |
| --- | --- |
| 3 | // enter a valid email address |
| 4 | driver.findElement(By.id("Email")).sendKeys("TestSelenium1607@gmail.com"); | |

|  |  |  |
| --- | --- | --- |
| 5 | log.info("Entered a valid Email Address."); | |
| 6 | // enter a invalid password |

|  |  |  |
| --- | --- | --- |
| 7 | driver.findElement(By.id("Passwd")).sendKeys("InvalidPassword"); | |
| 8 | log.info("Entered a invalid Password."); |

|  |  |  |
| --- | --- | --- |
| 9 | // click on sign in button | |
| 10 | | driver.findElement(By.id("signIn")).click(); | |

|  |  |  |
| --- | --- | --- |
| 11 | log.info("Clicked on the Sign In Button."); | |
| 12 | try{ |

|  |  |
| --- | --- |
| 13 | //Verify the home page |
| 14 | assertTrue("Verification Failed: User successfully landed on the Home Page.", driver.getTitle().equals("Gmail")); | |

|  |  |  |
| --- | --- | --- |
| 15 | log.info("Verified that the user landed on the Home Page."); | |
| 16 | } |

|  |  |  |
| --- | --- | --- |
| 17 | catch (Exception e) | |
| 18 | { |

|  |  |  |
| --- | --- | --- |
| 19 | log.error("Unsuccessfull Login."); | |
| 20 | } |

|  |  |
| --- | --- |
| 21 | } |

**Result in the log file**

01-07-2014 12:56:11 INFO [GmailLogin]: Uploaded the file to the System: FileExample.txt  
01-07-2014 12:56:11 INFO [GmailLogin]: Submitting the changes  
01-07-2014 12:56:15 ERROR [GmailLogin]: Unsuccessful Login.

### ****Conclusion****

In the current module, we pressed our focus on the **technical implication while implementing logging in a framework.**We exploited log4j utility to implement logging. We discussed the basic components those constitute log4j from a usability perspective. With the Appenders and layouts, user is leveraged to choose the desired logging format/pattern and the data source/location.