RTS Robot Framework

Run book

Author: Kalyan Gudikandula

Table of Contents

[Objective: 3](#_Toc367989010)

[Purpose: 3](#_Toc367989011)

[Pre-Requisites: 3](#_Toc367989012)

[Robot Framework Overview: 5](#_Toc367989013)

[Features 5](#_Toc367989014)

[Keyword-drivenTesting 5](#_Toc367989015)

[*Robot Framework* 6](#_Toc367989016)

[Remote Libraries 8](#_Toc367989017)

[Folder Structure: 9](#_Toc367989018)

[Resource Files: 10](#_Toc367989019)

[COMMON\_KEYWORDS.txt: 10](#_Toc367989020)

[COMMON\_OBJECTS.txt 10](#_Toc367989021)

[INCLUDE.txt: 10](#_Toc367989022)

[XXOLRS\_ALL\_OBJECTS.txt: 11](#_Toc367989023)

[XXOLRS\_TESTCASENAME\_TC.txt: 11](#_Toc367989024)

[TESTDATAFILENAME\_TESTDATA.xls 12](#_Toc367989025)

[XXOLRS\_TESTSUITENAME\_TESTSUITE.txt: 12](#_Toc367989026)

[Robot Framework IDE (RIDE): 13](#_Toc367989027)

[Edit Test Case: 14](#_Toc367989028)

[Auto Completion for Keywords: 15](#_Toc367989029)

[Search Keywords: 16](#_Toc367989030)

[Run Test Cases: 16](#_Toc367989031)

[Conclusion: 17](#_Toc367989032)

[Options in Pybot Command: 17](#_Toc367989033)

[Tips & Tricks: 18](#_Toc367989034)

[Version Control (CVS): 19](#_Toc367989035)

[Execution Procedure: 25](#_Toc367989036)

[Execution Results: 26](#_Toc367989037)

[1. log.html 26](#_Toc367989038)

[2. report.html 27](#_Toc367989039)

[3. output.xml 28](#_Toc367989040)

# Objective:

The objective of this document is to give complete information about the preparation of test cases, test data and test suite document, test execution and analyze results where tool generate in HTML format.

# Purpose:

The purpose of this document is to give knowledge to the person who would be writing and executing test cases using Robot Framework. This document would also talk about test execution steps and also where we would see execution results, how to analyze results.

# Pre-Requisites:

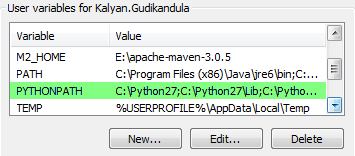
1. We must have Robot Framework Installed and Environment is set up (Follow below steps to get environment set up)
2. Visit the **Python** [download](http://www.python.org/download/) page and select the Windows binary installer for Python 2.7 (2.7.3 is the current release in June, 2012).
3. Run the installer (MSI) and choose all the defaults, including the install directory off of our boot hard drive (C:\Python27).
4. Verify that **Python** is installed by running the **Python CLI Shell** from the Windows Start Menu.

Enter exit() to terminate the shell.

1. Visit the **Robot Framework** [download](http://code.google.com/p/robotframework/downloads/list) page and select the Graphical installer for 32bit Windows. This installs the **Robot Framework** libraries.
2. Run the installer, and select the defaults. It should auto-detect our earlier install of Python 2.7 and use that path.
3. Visit the **wxPython** site and [download](http://wxpython.org/download.php#stable) the latest version that supports Unicode and Python 2.7 (2.8 is the current release in June, 2012).
4. Run the **wxPython** installer. Use all the default settings.
5. Go to <http://code.google.com/p/robotframework-ride/downloads/list> and download **Robot Framework IDE** (RIDE). Unpack the **RIDE** kit and move it to C:\Python27\Ride folder
6. Open the System control panel, and locate the Environment Variables button. Click on it

In the User Variables section, add the environment variable PATH with the value C:\Python27;C:\Python27\Scripts

Also add the environment variable PYTHONPATH with the value C:\Python27;C:\Python27\Lib;C:\Python27\libs;c:\Python27\Ride\lib\robot;C:\Python27\Ride\src



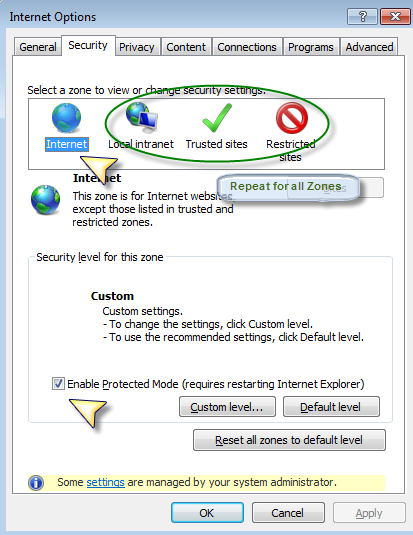
1. We must have Selenium2Library installed and Environment is set up

Pre-requisites: We must have **Python 2.7** installed and then **Robot Framework**.

1. Go to <https://github.com/rtomac/robotframework-selenium2library/downloads> and download the .tar.gz file and unzip it and run "python setup.py install" in a Terminal to install **Selenium2Library**.
2. Add the **Selenium2Library** file path to PATH system environment variable.
3. Download distribute-0.6.34.tar.gz from <http://pypi.python.org/pypi/distribute> and install using "python setup.py install".
4. Download decorator-3.4.0 from <http://pypi.python.org/pypi/decorator> and install using "python setup.py install".
5. Download selenium tar.gz from <https://pypi.python.org/pypi/selenium>. Unzip the .tar.gz file. This will give us a selenium-#.##.# directory. Go to this directory in a Terminal, and run "python setup.py install". This will create a selenium-#.##.#-py2.7.egg folder within C:\Python27\lib\site-packages. Inside the .egg folder, there will be a selenium folder. Cut and paste this folder into C:\Python27\lib\site-packages and overwrite the selenium folder that may or may not already be in site-packages folder.
6. Download IE driver (<http://code.google.com/p/selenium/downloads/list>) and place the .exe files inside C:\Python27 folder.
7. Copy all folders (xlrd-0.9.0, xlutils-1.5.2 and xlwt-0.7.4) from [N:\MDERT\RTS Tool\Excel Library](file:///N:\MDERT\RTS%20Tool\Excel%20Library) into our machine. Open a command prompt and go to a directory in a Terminal where xlrd-0.9.0 is saved and run command “python setup.py install”. Repeat the same for xlutils-1.5.2 and xlwt-0.7.4 folders. We would have all files installed in our machine.
8. Make sure Protected Mode is enabled in Security Tab for all Zones (Only for IE)

IE 🡪 Tools 🡪 Internet Options 🡪 Security 🡪

1. Select Internet (Under Zones) 🡪 Select Enabled Protected Mode Check Box
2. Select Local Internet (Under Zones) 🡪 Select Enabled Protected Mode Check Box
3. Select Trusted Sites (Under Zones) 🡪 Select Enabled Protected Mode Check Box
4. Select Restricted Site (Under Zones) 🡪 Select Enabled Protected Mode Check Box



# Robot Framework Overview:

This presentation document would give overview on Robot Framework 

Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). It has easy-to-use tabular test data syntax and utilizes the keyword-driven testing approach. Its testing capabilities can be extended by test libraries implemented either with Python or Java, and users can create new keywords from existing ones using the same syntax that is used for creating test cases.

Robot Framework is open source software released under [Apache License 2.0](http://www.apache.org/licenses/LICENSE-2.0.html). Its copyrights are owned and development supported by Nokia Siemens Networks.

## Features

* Enables easy-to-use tabular syntax for creating test cases in a uniform way.
* Allows using keyword-driven, data-driven and behavior-driven (BDD) approaches.
* Provides ability to create reusable higher-level keywords from the existing keywords.
* Provides easy-to-read reports and logs in HTML format.
* It is platform and application independent.
* The modular architecture supports creating tests even for applications with several diverse interfaces.
* Provides a simple library API for creating customized test libraries.
* Provides a command line interface and XML based outputs for integration into existing build infrastructure (continuous integration systems).
* Provides support for Selenium for web testing, Java GUI testing, running processes, Telnet, SSH, and so on.
* Remote library interface enables distributed testing and implementing test libraries in any programming language.
* Provides tagging to categorize and select test cases to be executed.

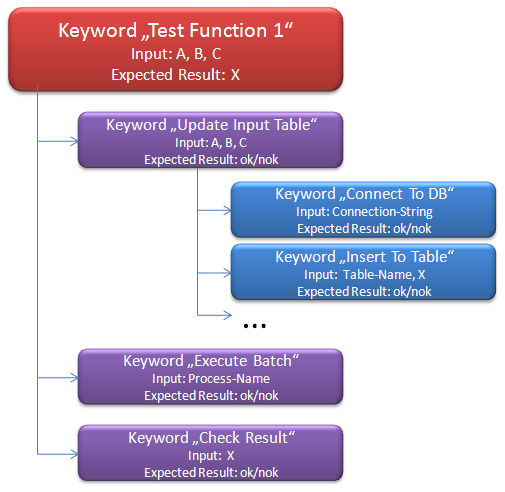
## Keyword-drivenTesting

*Keyword* is a function or method that can be used to test (or help test) one (smaller or bigger) aspect of the SUT (“System under Test”).

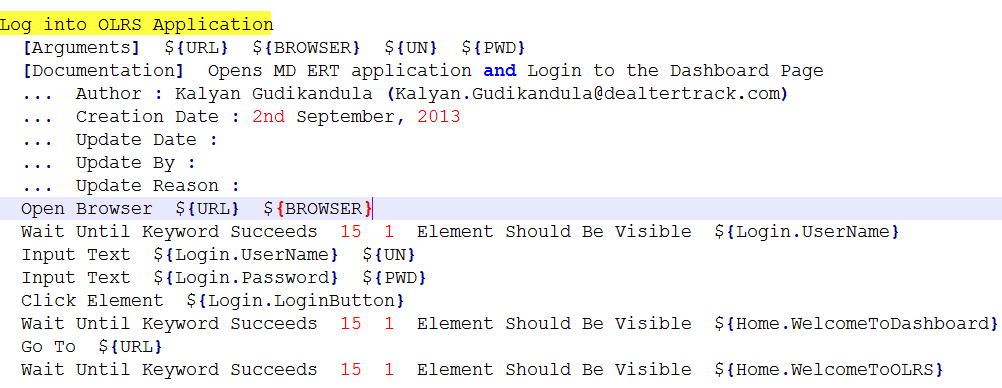
The really cool – and really powerful – thing about this now is that one *keyword* can be defined from other *keywords* and so on. That is why we are often talking of:

* **Higher-level keywords:** Those are really testing a concrete aspect of the business logic of the system under test.
* **Lower-level keywords:** To keep the implementation of the higher-level keywords at a decent size one is often breaking down the required functionality to several lower-level keywords.
* **Technical keywords:** Those provide the technical implementation to access and thus test the system.

The following figure depicts this based on an example keyword.



Technical *keywords* are written as a combination of already existing keywords.



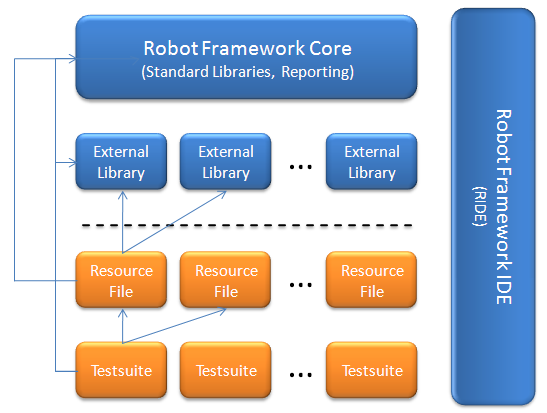
This above image is the definition of a *keyword* **Log into OLRS Application** using *keywords* from the *Selenium Library*. This leads us to the fact that there are already a lot of predefined *keywords* defined in so called *Test Libraries*.

## Robot Framework

When we install the *Robot Framework* we get the core framework and a set of *Standard Test Libraries* that are bundled with the core installation.

In addition to the *Standard Test Libraries* there are a lot of [additional external *Test Libraries*](http://code.google.com/p/robotframework/wiki/TestLibraries#External_test_libraries) available. Those are often contributed by the *Robot Framework* community and serve different purposes. The very good thing on all this libraries is that we can mix all the *keywords* from all the different libraries together in defining our own *keywords* or when writing a specific Test Case.

Thus we can for example easily write tests for a web application using keywords from the *Selenium Library* to remote control the web frontend and use at the same time the *Database Library* to check results of certain operations from the database. In an ideal case all this can be done by combining existing keywords from existing libraries to high-level *keywords*.



In addition to the core functionality and the *Test Libraries* the *Robot Framework* provides a graphical user interface called RIDE.

*Resource File:* Our Test Cases are bundled in so called *Test Suites*, which makes perfect sense. Now it would be possible to add new (higher-level) *keywords* into those *Test Suites* as well. But it is much better to have them defined in some external file and that file is then called a *Resource File*.

Finally this means that we have three artifacts that must/can be developed when writing tests using the *Robot Framework*.

**Test Suites:** This is where Test Cases (checks) are implemented. Typically every project will at least have one *Test Suites*. In bigger projects it makes for sure sense to divide the test functionality further into different *Test Suites*.

**Resource Files:** As it makes much sense from a test-design point of view one will almost always define own higher-level *keywords*. In turn this means that one will typically have own *Resource Files*. Especially in product development or in some long-running projects it makes sense to implement certain *keywords* centrally that fit to the product/project very well and can then be used by all teams.

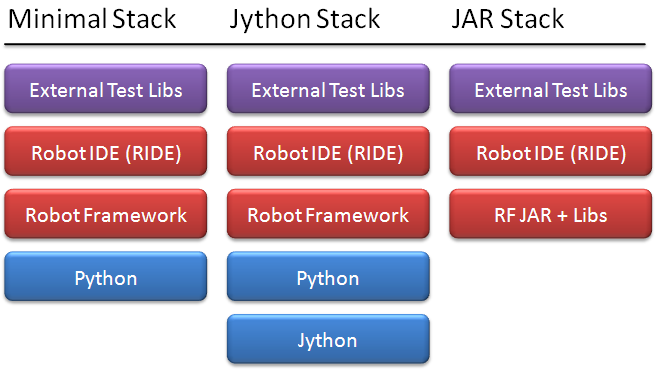
**Test Library:** Typically we are not required to write new technical *keywords* – for which we would need to implement a new *Test Library* – unless we are using a very specific technology. Nevertheless this is possible and does not always necessarily mean a lot of effort.

The *Robot Framework* itself and its *Core Libraries* are written in Python.

With **Jython** it is possible to run Python code in a Java Virtual Machine. This enables us to use Java for writing own *Test Libraries* or basically any programming language that is compiling to Java Byte Code.

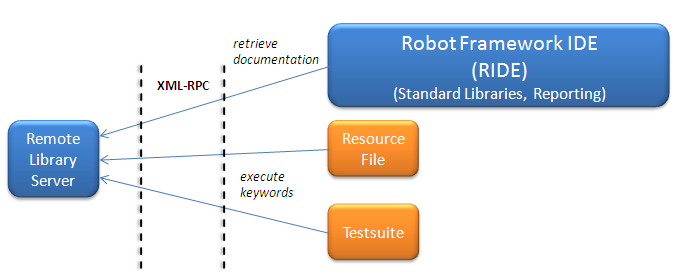
Using **IronPython** we get a similar integration into the .NET world that we get into the Java world when using Jython.

Following possible installation stacks:



## Remote Libraries

Remote Libraries enable us to write Test Libraries in basically any programming language supporting the [XML-RPC protocol](http://code.google.com/p/robotframework/wiki/RemoteLibrary) and running them on different machines (than the one running the *Robot Framework* as such) if required.



The usage of *Remote Libraries* does no differ from using “normal” *Test Libraries* beside the way they are imported in *Test Cases* and/or *Resource Files*. Another big advantage here is that RIDE also fully supports this feature and is able to retrieve the documentation from *Remote Libraries* as well.

*Remote Libraries* are started as a remote server and the *Robot Framework* is then acting as the client. Of course we can use those libraries as well completely locally.

# Folder Structure:

Following is the standard folder structure we use in Robot Framework

1. **..\COMMON**

This folder contains all the Robot Framework resource files common to multiple test cases. It generally holds

1. **INCLUDE.txt**

We specify all resource files required for Robot Framework scripts to execute, like selenium library, common file, include file, keywords files etc…

1. **COMMON\_KEYWORDS.txt**

Holds custom keywords (Ex: Login, Logout, Close Browser etc…)

1. **COMMON\_OBJECTS.txt**

Holds variables with values assigned to them which are global (can be accessed across OLRS application scripts) and Property values of all fields in application assigned to variables which are common to all OLRS applications.

1. **..\XXOLRS TRANSACTIONS (XX – State Code)**

This folder contains all the Robot Framework resource files specific to a state OLRS application

1. **\KEYWORDS\XXOLRS\_TESTCASENAME\_TC.txt**

Keywords/Test Cases specific to a state OLRS application

1. **\OBJECTS\XXOLRS\_ALL\_OBJECTS.txt**

Property values of all fields in application assigned to variables which are specific to a state OLRS application

1. **\TESTSUITE\XXOLRS\_TESTSUITENAME\_TESTSUITE.txt**

All Test Cases would be putting together into a single test suite file. Executing a Suite file would pull the test cases matching execution criteria

1. **\TESTDATA\XXOLRS\_TESTCASENAME\_TESTDATA.xls**

Test Data required to execute respective test case would be placed here. Make sure column name in the test data file does match with the parameter name used in respective test case

1. ExcelLibrary.py and ExcelLibrary.pyc (Compiled Python File)

This test library provides some keywords to allow opening, reading, writing, and saving Excel files from Robot Framework

1. **RESULTS** (Optional)

Results will be saved in this folder only if we specify –d “..\RESULT” with pybot command. Default is the directory where tests are run from. (Ex: log.html, report.html and output.xml)

# Resource Files:

## **COMMON\_KEYWORDS.txt**:

This file contains Keywords which can be used across multiple test cases/suites. It has two sections

1. \*\*\* Settings \*\*\*

In this section, we mention all resource files needed to run this text file

Syntax: Resource <Tab Space>< File Name> (File Name could be relative/absolute path)

Ex: Resource INCLUDE.txt

1. \*\*\* Keywords \*\*\*

In this section, we write all keywords needed to use in test case. When we define keyword, we have multiple options to specify

1. [Arguments] All arguments we pass to the current keyword written in ‘{‘

Syntax: [Arguments]<Space><Space>${Arg1}<Space><Space>${Arg2}

Ex: [Arguments] ${Target} ${Data}

ii. [Documentation] Description/Purpose of Keyword will be written under this

Syntax: [Documentation]<Space><Space><Purpose>

Ex: [Documentation] This is Description

**Note**: Use … to enter description in a new line

## **COMMON\_OBJECTS.txt**

This file contains variables which can be used across OLRS application scripts and locators of objects common across OLRS applications used to identify an object by Robot Framework during test execution. It has two sections

1. \*\*\* Settings \*\*\*

Description/Purpose of this file

Syntax: Documentation <Tab Space> <Description>

Ex: Documentation Global Variables, Object locators which can be used across OLRS applications

1. \*\*\* Variables \*\*\*

In this section, we write all variables with values assigned to them needed to use in test case. Syntax: ${Variable Name}<Tab Space><Value>

Ex: ${Browser} ie

## **INCLUDE.txt:**

This file contains information of all library and resource files, we include in across test cases.

It has only one section in it

1. \*\*\* Settings \*\*\*

Syntax: Documentation <Tab Space> <Description>

Ex: Documentation One place for all libraries

Syntax: Library <Space><Space><File Name>

Ex: Library Selenium2Library implicit\_wait=60

Library String

Library Collections

Library OperatingSystem

Library ExcelLibrary.py

Syntax: Resource <Space><Space><File Name>

Ex: Resource COMMON\_OBJECTS.txt

Resource COMMON\_KEYWORDS.txt

Resource ../MDOLRS TRANSACTIONS/OBJECTS/MDOLRS\_ALL\_OBJECTS.txt

Resource ../MDOLRS TRANSACTIONS/KEYWORDS/MDOLRS\_NEWREGISTRATION\_TC.txt

## **XXOLRS\_ALL\_OBJECTS.txt:**

This file contains variables which can be used in respective state OLRS application scripts and locators of all objects in respective state OLRS application used to identify an object by Robot Framework during test execution. We place locators of all objects in a single file.

It has two sections in it

1. \*\*\* Settings \*\*\*

Description/Purpose of this file

Syntax: Documentation <Tab Space> <Description>

Ex: Documentation Recognition of the Web Components\Elements through X-path, Id

1. \*\*\* Variables \*\*\*

In this section, we write all variables with object id/Path values assigned to them

Syntax: ${Variable Name} <Space><Space><Value>

Ex: #Parameters

${Path} [Application URL](file:///C:\Users\jagadish.chandran\Downloads\Application%20URL)

#Login Page

${Login.UserName} j\_idt32:loginForm:UserID

${Login.Password} j\_idt32:loginForm:Password

${Login.LoginButton} j\_idt32:loginForm:loginBtn

## **XXOLRS\_TESTCASENAME\_TC.txt:**

This file contains all Sub Routines (Similar to Functions). These Sub Routines are created grouping set of individual statements which need to be accessed multiple times in multiple test cases.

Similar to functions:

1. We call inner statements by Sub Routine name in Test Cases.
2. We can pass arguments to the Sub Routine using “[Arguments]” keyword and when we call this keyword in other test cases we pass argument values.
3. We cannot use pre-defined keywords as Sub Routine name.

This file has two sections in it

1. \*\*\* Settings \*\*\*

Information about Resource file(s) name

Syntax: Resource<Space><Space><File Name>

Ex: Resource ../COMMON/INCLUDE.txt

1. \*\*\* Keywords \*\*\*

Log into OLRS Application //Sub Routine Name

[Arguments] ${URL} ${BROWSER} ${UN} ${PWD}

**[Documentation]** Opens OLRS application and Login to the Dashboard Page

**... Author :**

**... Creation Date :**

**... Update Date :**

**... Update By :**

**... Update Reason :**

Open Browser ${URL} ${BROWSER}

Wait Until Keyword Succeeds 15 1 Element Should Be Visible ${Login.UserName}

Input Text ${Login.UserName} ${UN}

Input Text ${Login.Password} ${PWD}

Click Element ${Login.LoginButton}

Wait Until Keyword Succeeds 15 1 Element Should Be Visible ${Home.WelcomeToDashboard}

Go To ${URL}

Wait Until Keyword Succeeds 15 1 Element Should Be Visible ${Home.WelcomeToOLRS}

## TESTDATAFILENAME\_TESTDATA.xls

This test data file contains data; we need for a test case to execute. To locate the test data file respective to any test case file, we follow a naming convention.

Here we are naming test data file as test case name.

Syntax: <TESTCASENAME>\_TESTDATA.xls

Ex: If the Test Case name is **RTS\_T1\_L1\_23** thentest data file namewould be **RTS\_T1\_L1\_23\_TESTDATA**.xls

## XXOLRS\_TESTSUITENAME\_TESTSUITE.txt**:**

This file contains all test cases designed as a suite file. When we execute this file from command prompt using “***pybot TestSuite.txt***” command, then all test cases added in this file will be executed in sequential order.

Test Data required for test case to execute will be read from the excel file passed an argument from the column matching variable name mentioned in the respective test case.

This file has two sections in it

1. \*\*\* Settings \*\*\*

Information about Resource file(s) name

Syntax: Resource<Space><Space><File Name>

Ex: Resource ../COMMON/COMMON\_KEYWORDS.txt

Resource ../COMMON/INCLUDE.txt

\*\*\* Test Cases \*\*\*

RTS\_T1\_L1\_23

[**Documentation**] Verify the T1 Pend transaction and click on Void

... **Flow** : Login --> Transaction Page --> New Soundex --> Pend --> Void

... **Author** :

... **Creation Date** :

... **Update Date** :

... **Update Reason** :

Test Data Iterations C:/RFW/TestData\_23.xls Owner with default data for Pend and Void Transaction

RTS\_T1\_L1\_167

[**Documentation**] Verify Sales Tax for Body Style PE is no longer Tax Exempt

... **Flow** : Login --> Transaction Page --> Body Style = PE --> S.T is 6%

... **Author** :

... **Creation Date** :

... **Update Date** :

... **Update Reason** :

Test Data Iterations C:/RFW/TestData\_167.xls Validate Tax Exemption for Vehicle PE

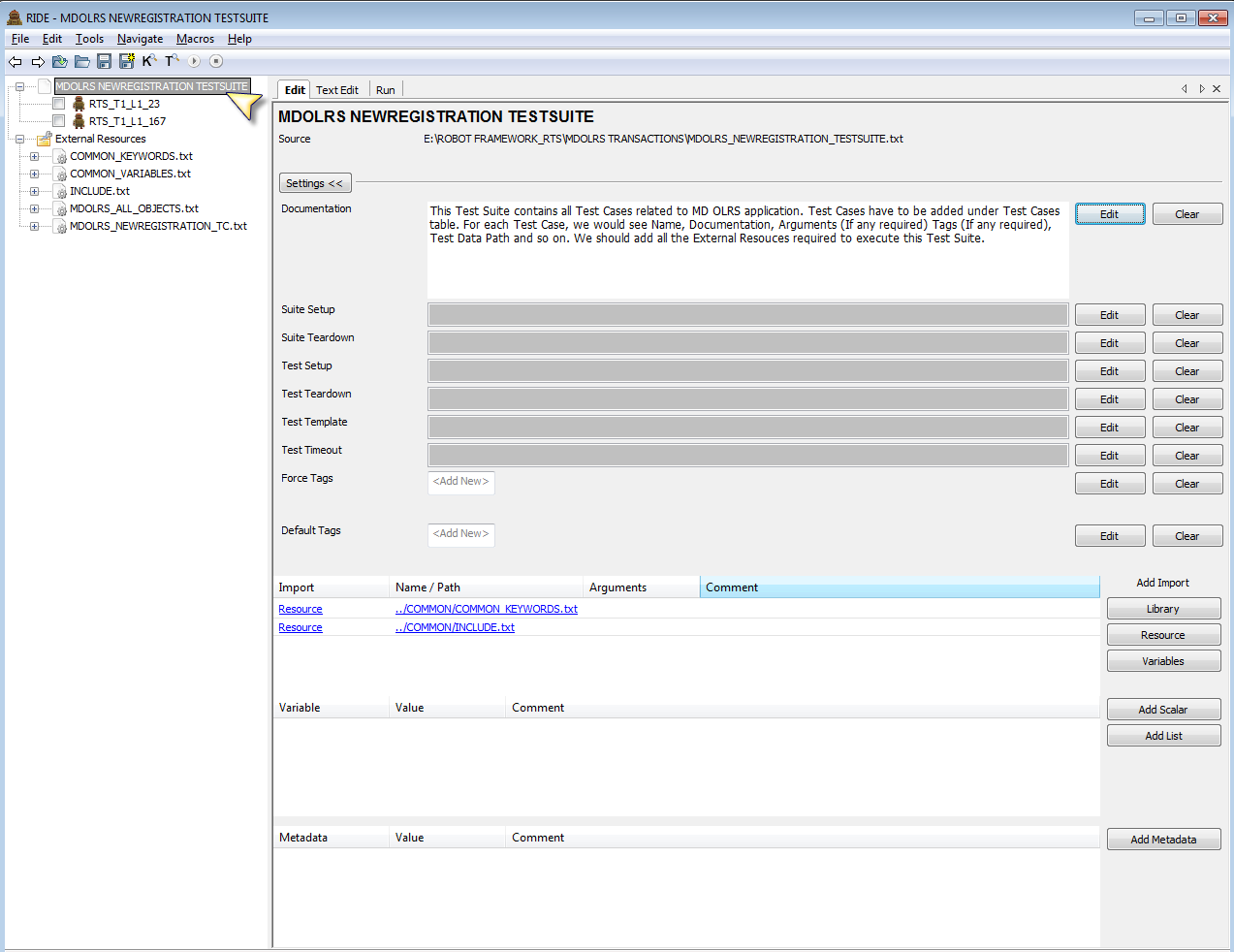
In the above examples defined in Test Cases section, we have total 3 sub routines:

1. Test Data Iterations - Execute respective test case Iterations equal to data rows in excel data file
2. Owner with default data for Pend and Void Transaction - Executes set of statements placed in it
3. Validate Tax Exemption for Vehicle PE - Executes set of statements placed in it

# Robot Framework IDE (RIDE):

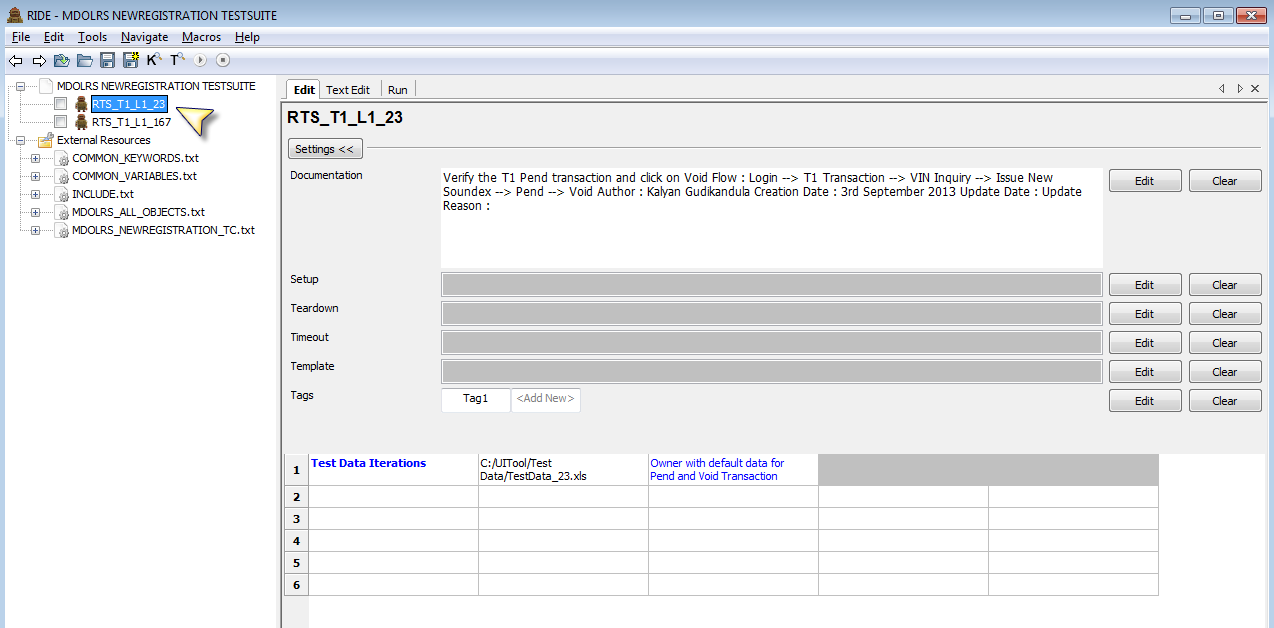
The *Robot Framework IDE* (*RIDE*) is the integrated development environment to implement automated tests for the [*Robot Framework*](http://code.google.com/p/robotframework/). The *Robot Framework* is a generic test-automation framework. RIDE downloads are only available from [GitHub](https://github.com/robotframework/RIDE/downloads).

Once a Test Suite has been loaded (or alternatively a whole directory containing several Test Suites) it will be displayed in a tree-like structure on the left hand side of the editor. For each Test Suite individual Test Cases can be selected for editing from this navigation tree. It is positive that every referenced resource file is automatically loaded and shown in the navigation tree under *External Resources*. As long as the Test Suite is selected it is possible to configure its global settings, for example Suite-Setup and Suite-Teardown

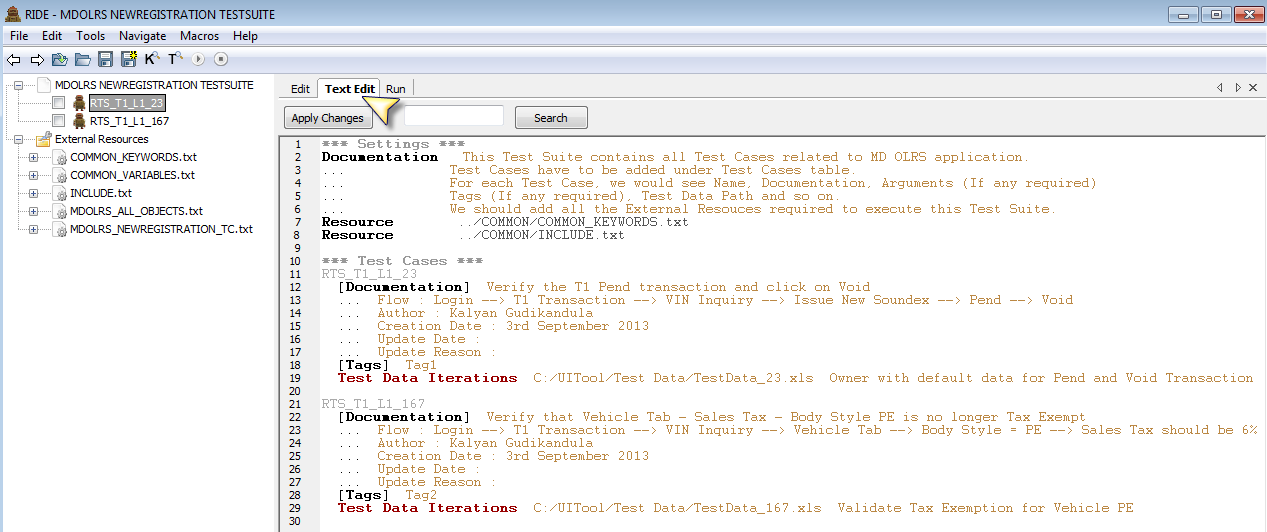


## Edit Test Case:

To edit an individual Test Case one has to select that Test Case from the navigation tree first. Generic fields for this Test Case – for example documentation and tagging – can then be defined in the upper part of the editor. In the lower part the Test Case is written in tabular format as a sequence of corresponding keyword calls. Thereby the keyword is written in the first column and all possible parameters in the other columns. This is intuitive and gets supported by additional features that are described in the following



So let’s switch to the corresponding panel. First I am a little bit surprised that the pure text format is shown here and not the HTML format as RIDE is storing files per default in that format. But of course this makes sense as hardly anyone would like to edit HTML directly. Internally all data structures are anyway handled independently from the output format.



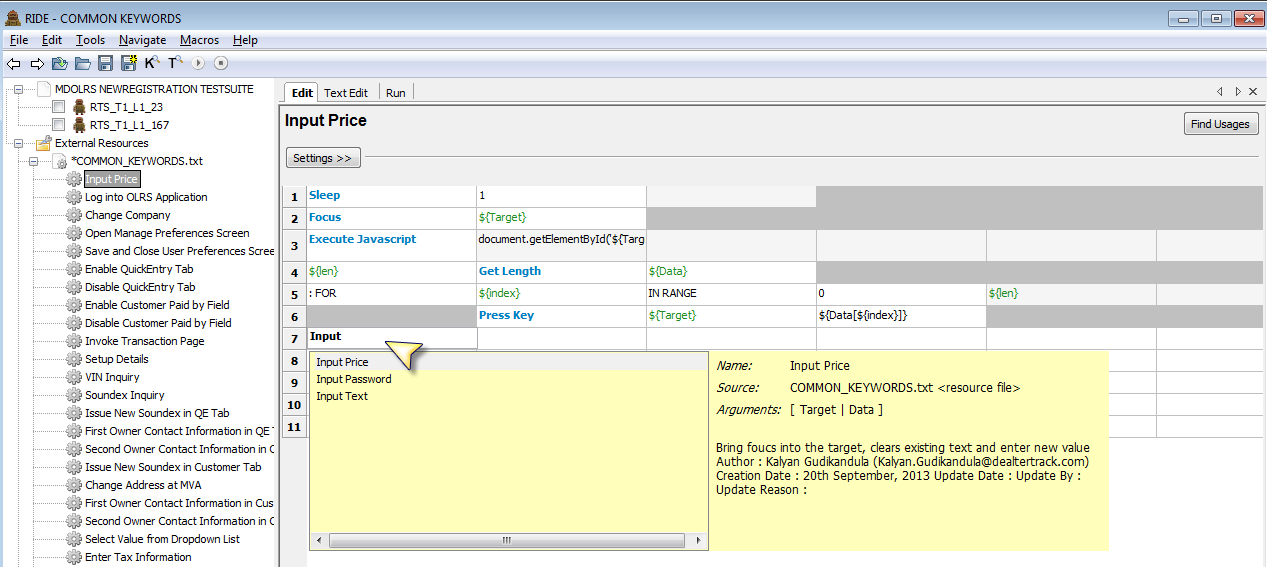
A quick test shows that changes in the text editor show up in the visual editor and vice versa. Switching between both views is thus working very smoothly and as expected

**Note**: To see Syntax Highlighter in Text Edit mode, Download [Pygments](http://pygments.org/download/) (Unzip into Python folder and Change Directory to Pygments folder in command prompt. Now run ***easy\_install Pygments*** command).

## Auto Completion for Keywords:

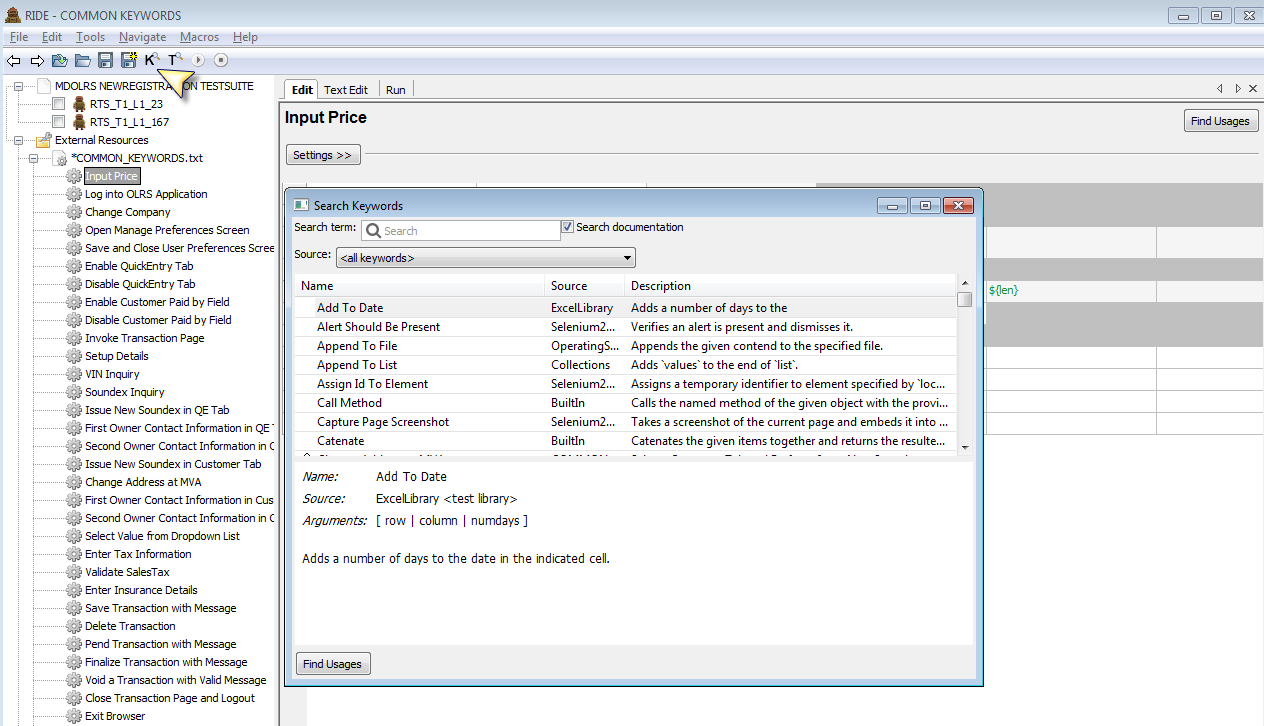
*RIDE* implements this support already for some time by offering an auto completion for keywords. This can be activated by pressing “CTRL-Space” when entering a keyword.

In an empty field this will then show all available – read from all included test libraries – keywords. Otherwise it is possible to start typing and then get the auto completion for all keywords starting with the already typed text. In addition the available documentation for those keywords is also shown with description.

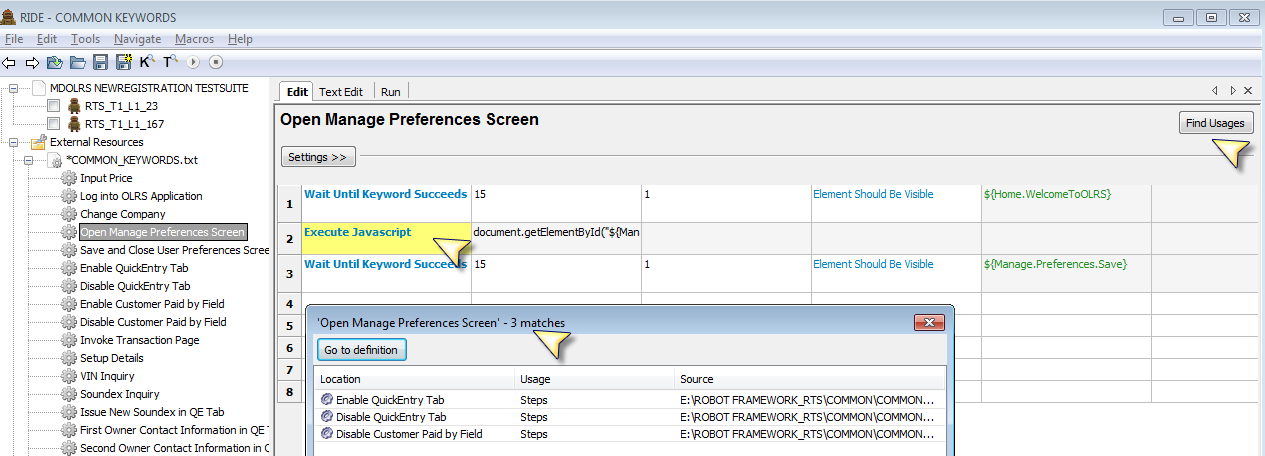


Search Keywords:

To get an overview of all those keywords under the menu option Tools 🡪 Search Keywords (F5)

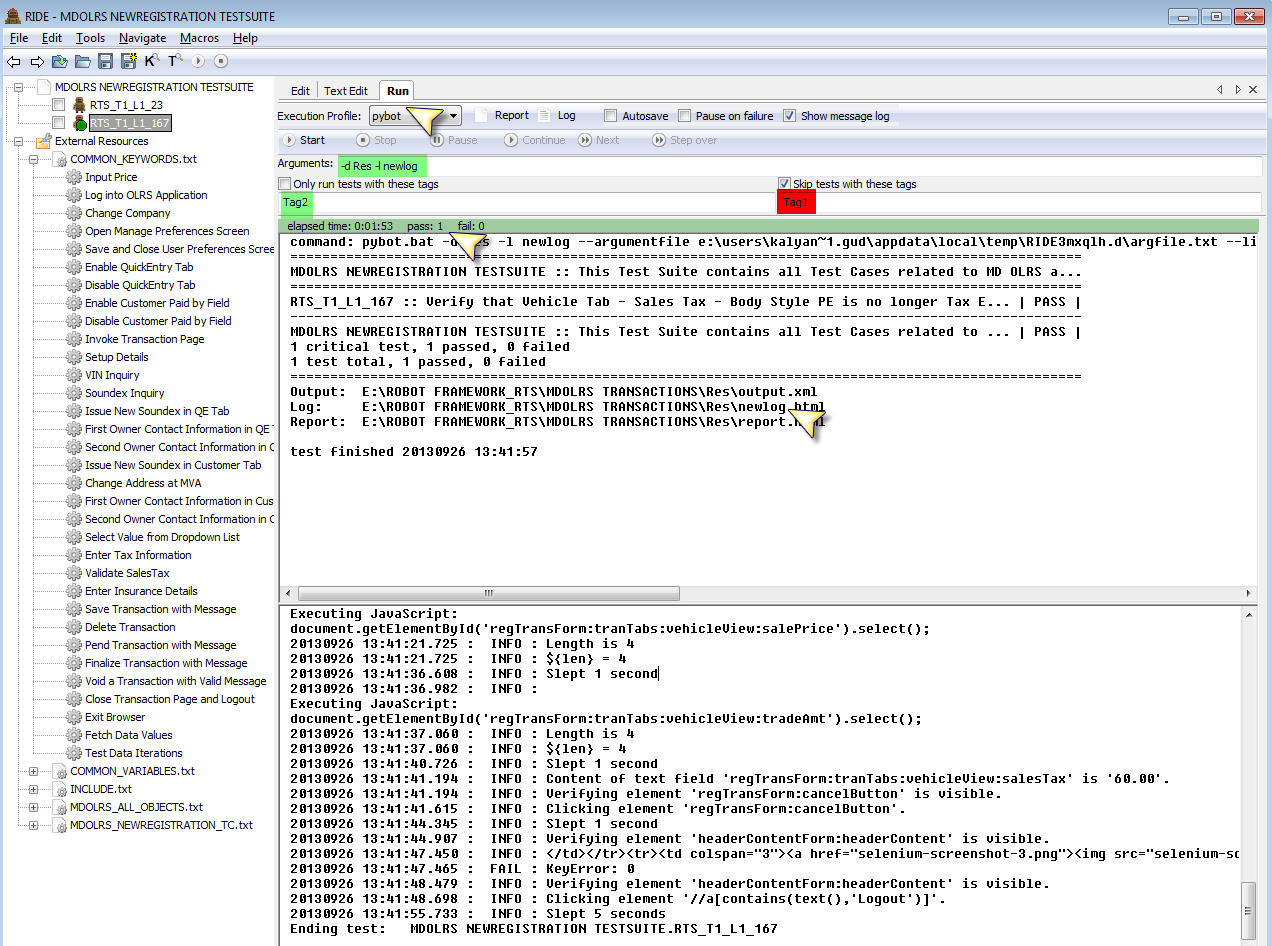


A very nice feature is finding all places where a specific keyword has been used. This can be done by pressing the button Find Usages in the corresponding editor page. As already mentioned it is also possible here to search for all places in all open files where such a keyword is used. This is especially useful when refactoring Test Cases. By pressing on one of the search results RIDE jumps directly to the corresponding place in the editor.



## Run Test Cases:

Finally let’s have a look at the third panel that is named Run. This enables the user to directly execute the tests for the currently opened Test Suite. This can be done by using **pybot**, **jybot** or a customized script. For smaller projects one of the first two options might work quite well. But bigger projects usually also have quite some additional parameters and potentially other startup configuration that is defined in individual startup scripts. It is a very good solution that those ones can be executed from here. The output of the script execution can be seen in the editor. The following shows an example where all Test Cases have been passed



## Conclusion:

RIDE is evolving more and more to a very complete IDE for developing tests with the *Robot Framework*. This tool offers a lot of guidance and internal documentation. Especially for less technical users this is a big advantage. In the other extreme, there is still the possibility to use [Eclipse and Maven to develop tests](http://blog.codecentric.de/en/2010/03/robot-framework-acceptance-tests-develop-with-eclipse-run-with-maven/). For all users of RIDE I think it will be very interesting to see how the tool will evolve in the upcoming releases.

# Options in Pybot Command:

|  |  |
| --- | --- |
| Option | Description with **Example** |
| -i --include tag | Select test cases to run by tag  **Ex: pybot -i --Functional SampleSuite.txt** |
| -e --exclude tag | Select test cases not to run by tag  **Ex: pybot -e --Regression SampleSuite.txt** |
| -d --output directory | Where to create output files. Default is the directory where tests are run from and the given path is considered relative to that unless it is absolute.  **Ex: pybot -d <C:\Robot Framework\Results> SampleSuite.txt** |
| -o --output file | Creates XML output file with given name. If file Name contains spaces put it in quotes  **Ex: pybot -o “My Report” SampleSuite.txt** |
| -l --log file | Creates HTML log file with given name. Can be disabled by giving NONE. Default is log.html  **Ex: pybot -l Mylog SampleSuite.txt** |
| -r --report file | Creates HTML report file with given name. Can be disabled by giving NONE. Default is report.html  **Ex: pybot -r MyReport SampleSuite.txt** |
| -T --timestamp output | When this option is used, timestamp in a format ‘YYYYMMDD-hhmmss’ will be added to all generated output files between their filename and extension.  **Ex: pybot -T SampleSuite.txt** |
| --logtitle title | Title for generated log file. Default is <TestSuiteName><Space>Test Log  **Ex: pybot --logtitle MyTitle SampleSuite.txt** |
| --reporttitle title | Title for generated report file. Default is <TestSuiteName><Space>Test Report  **Ex: pybot --reporttitle MyReport SampleSuite.txt** |

# Tips & Tricks:

1. If user edit test cases in Notepad++ application then
   1. Follow Language🡪P🡪Python navigation to view in python syntax
   2. All Robot Framework resource files should be saved with .txt (Text File Format) extension
   3. If Robot Framework Keyword has multiple words in it, only single space should be given between the words

Ex: Input<Space>Text

* 1. We need to give two spaces between Keyword and Argument

Ex: Input Text<Space><Space><Target>

* 1. Two spaces between the Arguments

Ex: Input Text Username<Space><Space>Bob

1. Use Robot Framework IDE (RIDE) to write/edit test cases. RIDE is a light-weight and in-built editor for Robot Framework test case files.
2. In command prompt, first do Change Directory to the path where test case to run exist and then use **pybot** command
3. If application opens desired page in another browser
   1. Use **Get Window Title**  keyword to get the window titles opened by Robot Framework
   2. Use **Select Window** **<Title Name>** keyword to switch to a window specified by title

If the window is found, all subsequent commands use that window, until this keyword is used again. If the window is not found, this keyword fails. If multiple windows with same identifier are found, the first one is selected.

1. If we want to navigate to other URL from the same browser window, we could use below keyword. New page opens within the same browser window

Syntax: Go To<Space><Space>${URL}

Ex: Go To [www.dealertrack.com](http://www.dealertrack.com)

1. While running the script in the Firefox, we would see that execution stuck while invoking the FF browser in few cases and do not proceed further. In such cases, just we need to do mouse hover on the browser page, then execution would continue.
2. Instead of using **Wait Until Page Contains Element** keyword for synchronization, we could use the below alternate

**Wait Until Keyword Succeeds** 15 1 **Element Should Be Visible** ${Target}

# Version Control (CVS):

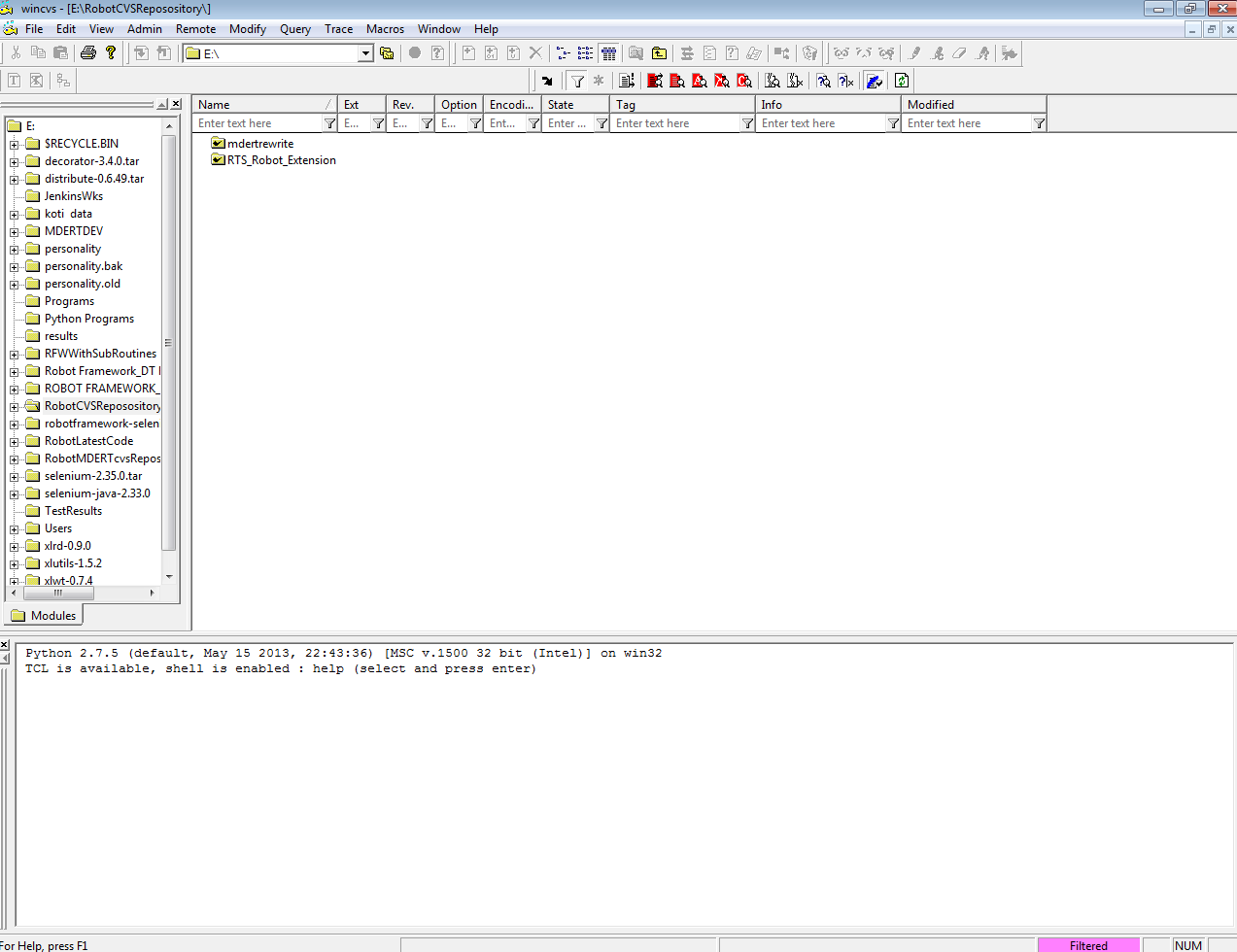
CVS stands for *Concurrent Versioning System*. A version control system stores one master copy of our source code in a central repository and allows us to checkout files to make changes to them.

CVS helps if we are part of a group of people working on the same project. It is all too easy to overwrite each other’s’ changes unless we are extremely careful. CVS solves this problem by insulating the different developers from each other. Every developer works in his own directory and CVS merges the work when each developer is done. Using CVS, we can also record the history of our source files and can easily retrieve old versions to see what modifications we made.

CVS uses client–server architecture: a server stores the current version(s) of a project and its history, and clients connect to the server in order to "check out" a complete copy of the project, work on this copy and then later "check in" their changes. Typically, the client and server connect over a [LAN](http://en.wikipedia.org/wiki/Local_area_network) or over the [Internet](http://en.wikipedia.org/wiki/Internet), but client and server may both run on the same machine if CVS has the task of keeping track of the version history of a project with only local developers.

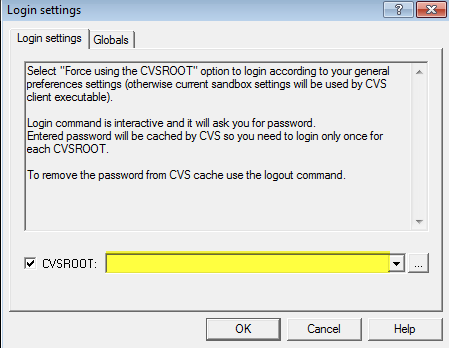
**Pre-Requisites:**

1. User should have access to CVS Central Repository
2. Make sure Win CVS tool has been installed in the local machine
3. Launch Win CVS Tool



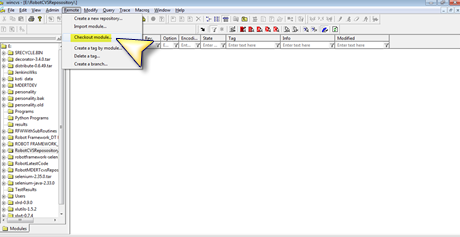
1. Go to Admin 🡪 Login.

Enter CVSROOT path and click OK. User will be prompted for password. It will prompt only once for each CVSROOT.

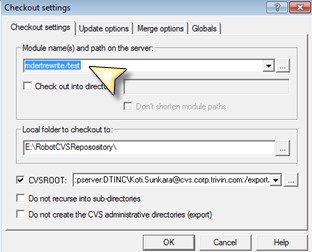


1. **Check Out Module**

Choose Remote 🡪 Checkout Module



In the dialog box that pops up as below, select the name of the module that we want to check out (If we have multiple modules to checkout, repeat the process for each module). Specify the name of the local folder, when the checkout modules should get save.



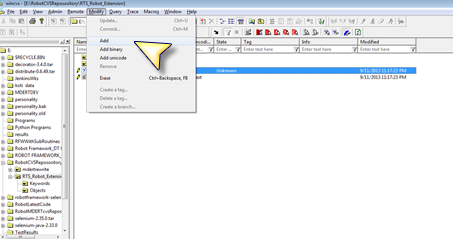
Press the OK button.

**Note**: Make sure all further work we do, is done on the files in the folder we have checked out the module to. Any existing file in the folder, where check out is done, does not come under version-control.

1. **Adding Files and Directories**

When we create new files that we want to include in the repository, we must tell CVS to handle the files. If the directory containing the files is not under CVS control, we will have to add it before adding the files.

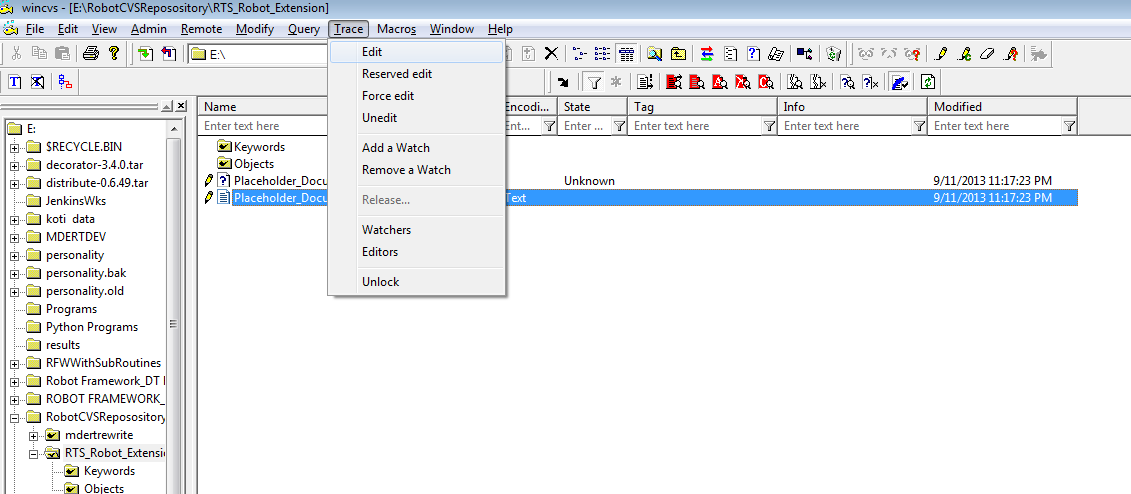
1. Select the directory, file or files that we want to add
2. Choose Modify 🡪 Add



1. Files are only marked for addition; we have to commit them to get them into the repository
2. **Editing a File**

We will need to explicitly tell WinCVS, which files we intend to edit before editing them. This may be cumbersome, but it enables other developers to track who is currently editing given files (see step 5 below).

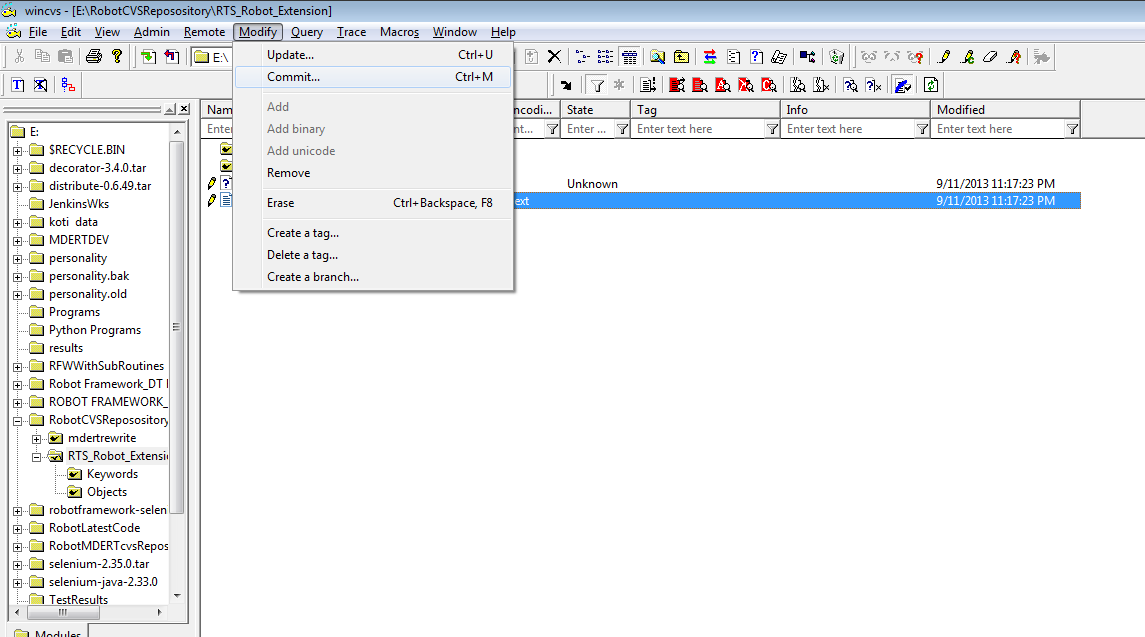
1. Select a file that we want to edit by clicking on it in the workspace
2. Choose Trace 🡪 Edit



1. The icon next to the filename changes to reflect the fact that we can edit it now
2. Open the file by double-clicking on it and save the necessary changes. The icon next to the file becomes red, to indicate that it is a dirty file
3. Make sure to unedit the file, if we don’t intend to check it in by choosing Trace 🡪 Unedit
4. It is possible to ask CVS who is currently editing a file
5. Select the file we are curious about and choose Trace 🡪 Editors. The list of known editors of the file is displayed in the output window
6. **Committing Changes**

All changes we make in the file that we have checked out are only to our local copy. If we want to make those changes available to others who access that file in the repository, we need to commit our changes to the repository by checking it in.

1. Select the directory, file or files that we want to commit
2. Choose Modify 🡪 Commit



1. In the Commit Settings dialog box, enter a log message to describe what changes we made and press OK
2. If the commit was successful, we will see the following message in the output:

*cvs commit -m "no message" readme.txt (in directory C:\Group1\)*

*Checking in readme.txt;*

*/usr/local/cvsroot/Group1/readme.txt,v <-- readme.txt*

*new revision: 1.3; previous revision: 1.2*

*done*

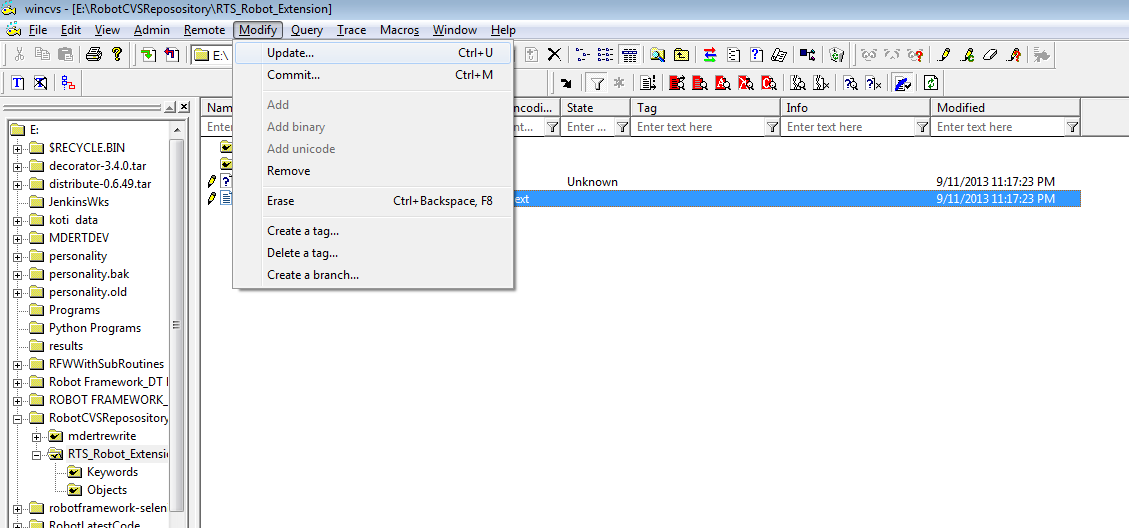
*\*\*\*\*\*CVS exited normally with code 0\*\*\*\*\**

**Note:** Before committing, we should do an update to make sure there are no conflicts

1. **Updating Files**

There might be more than one person working on the same file. One of the developers will have to merge in the changes made by the other developers before committing the changes. The process of getting changes from the server to our local copy is called updating.

1. Select the directory, file or files that we want to update
2. Choose Modify 🡪 Update



1. A merge conflict will result if two developers modify the same portion of a file. If a merge conflict is detected, WinCVS displays a red C icon next to the file. It's up to the developer to resolve conflicts by hand. CVS will not let us commit a file that has unresolved merge conflicts in it.

For example, the following output is generated in the output window:

*cvs update readme.txt (in directory C:\Group1\Group1)*

*RCS file: /usr/local/cvsroot/Group1/readme.txt,v*

*Retrieving revision 1.3*

*Retrieving revision 1.4*

*Merging differences between 1.3 and 1.4 into readme.txt*

*rcsmerge: warning: conflicts during merge*

*cvs server: conflicts found in readme.txt*

*C readme.txt*

**Note**: If a merge conflict occurs, we need to manually resolve the conflict before we can proceed to commit our changes

1. **Resolving Conflicts**

Two developers can modify the same file at the same time. One of the developers will have to merge in the other developer's changes before committing the file. If we don't check in our changes frequently we will be merging a lot. It's best to communicate with our team members to make sure we don't step on each other's toes.

Let's say that someone modified a file that we’ve been working on. Now we have to merge in the other developer's changes. This process is called "updating." *(For more information see the section updating files*). A merge conflict will result if two developers modify the same portion of a file. If a merge conflict is detected, WinCVS displays a red C icon next to the file. It's up to the developer to resolve conflicts by hand. CVS will not let we commit a file that has unresolved merge conflicts in it.

During an update, CVS modifies the file to show us the conflicting edits. But not to worry -- the original file is saved to another file like **readme.txt.1.3** before the merge takes place, so we will not lose our work.

*The beginning of the conflict is marked with <<<<<<<*

*The end of the uncommitted changes is marked with =======*

*The end of the committed changes is marked with >>>>>>>*

For example, a file that has a merge conflict might have the following text:

*Good, this seems to work :)*

*<<<<<<< readme.txt*

*Hi there !!! abcdef=======*

*Hi there !!! okay>>>>>>> 1.4*

Both set of changes must be resolved to a single set of changes before committing the file. This normally involves communication both the developers who worked on that file and made changes to it. Once the conflict has been resolved, go ahead and commit the changes.

1. **References**

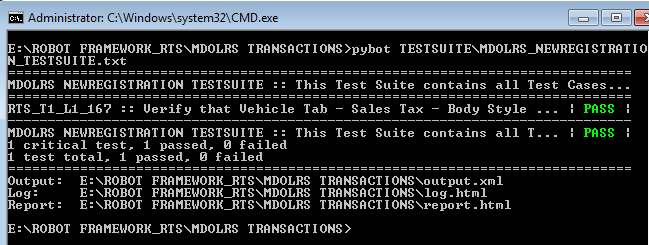
We might find the following resources helpful, if we run into any questions/problems while using WinCVS.

1. **WinCVS Daily Use Guide**: *http://www.computas.com/pub/wincvs-howto*/ (Excellent reference material with helpful tips for basic functions)
2. **WinCVS Version 1.1 User Guide**: *http://www.wincvs.org/winhtml/wincvs11.htm* (Step by step guide for every function available)
3. **CVS and WinCVS QuickStart Guide**: *http://devguy.com/fp/cfgmgmt/cvs/startup*/ (Step-by step instructions with nice pictures)
4. **CVS Information Manual**: *http://www.cvshome.org/docs/manual/cvs.html* (How to use and administer CVS)

# Execution Procedure:

First user should create a Test Suite file adding the required test cases into it

1. Make sure all resource/library file paths are added in the respective .txt files
2. Open a Command Prompt
3. Change Directory to the folder where all state OLRS application resource files are saved
4. Type “**pybot TESTSUITE\TestSuiteFileName.txt**” command and Hit Enter
5. Test Suite will be executed and show the results in command prompt after completion



# Execution Results:

After test execution, PASS/FAIL would be shown in command prompt for each test case separately and few results files will also be generated.

## log.html

This file shows the results of all test cases exist in the suite file in two sections

1. **Test Statistics**

This section shows information in tabular column and it has three sub sections in it.

*Total Statistics* : Total Executed, #Pass, #Fail, Total Execution Time, Pass/Fail %

*Statistics by Tag* : Shows results group by Tag name

*Statistics by Suite* : Total Suites Executed, #Pass, #Failed, Elapsed Time

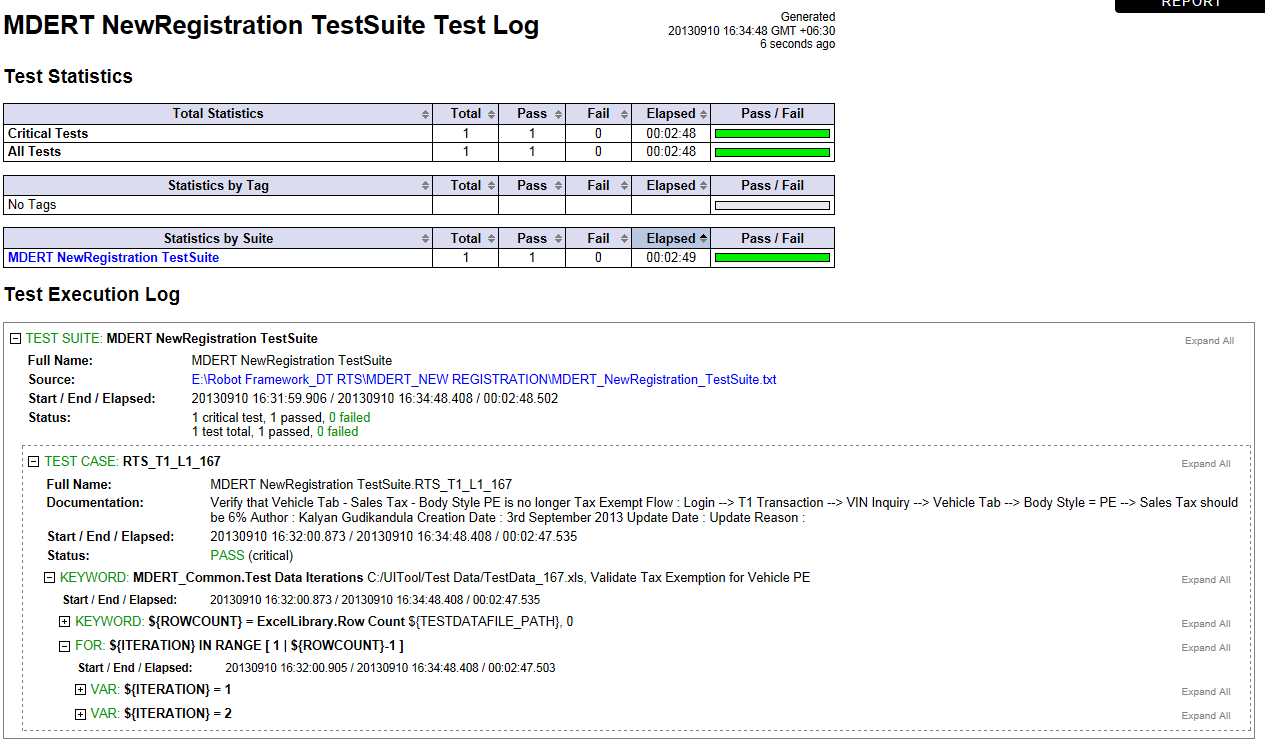
1. **Test Execution Log**

This section shows result of all individual statements for all test cases. It has two sub sections in it.

*TEST SUITE* : Suite Name, Path, Start/End/Elapsed Time and Status

*TEST CASE* : Test Case Name, Documentation, Start/End/Elapsed Time and Status

(This way it shows information for each individual test case)



## report.html

This file shows the test summary information and a link to log.html file. It has three sections in it

1. **Summary Information**

Status : All tests passed

Start Time :

End Time :

Elapsed Time :

Log File : log.html (Hyperlink)

1. **Test Statistics**

Same as defined in log.html file

1. **Test Details**

Suite Name : Dropdown (If we have executed multiple test suites)

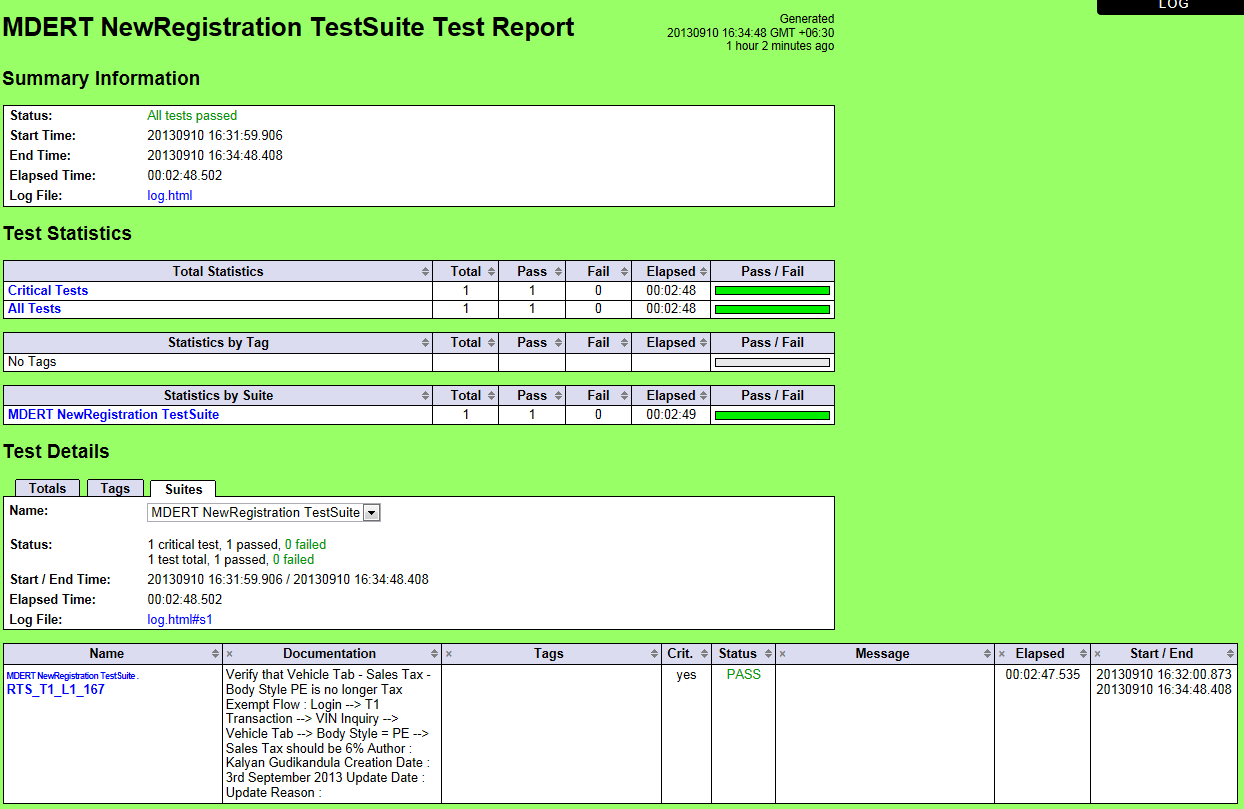
Status :

Start/End Time :

Elapsed Time :

Log File :

Table with Test Suite Name (Hyperlink, Clicking on this link will take us to log.html), Documentation, Tags, Critical, Status, Elapsed and Start/End columns



## output.xml

This file shows above results in .xml file

