# Guidelines for Test Automation in Robot Framework

Contents

[Guidelines for Test Automation in Robot Framework 1](#_Toc504568389)

[Document History 2](#_Toc504568390)

[Standards 2](#_Toc504568391)

[Introduction 2](#_Toc504568392)

[Robot Framework tool 3](#_Toc504568393)

[Introduction 3](#_Toc504568394)

[Installation 3](#_Toc504568395)

[Install Python 3](#_Toc504568396)

[Install Robot Framework and extensions 4](#_Toc504568397)

[Install Chrome extensions 4](#_Toc504568398)

[Install browser drivers 5](#_Toc504568399)

[First steps in Robot Framework IDE (RIDE) 5](#_Toc504568400)

[Start RIDE 5](#_Toc504568401)

[Create a project 5](#_Toc504568402)

[Add libraries 7](#_Toc504568403)

[Create and run the first test 9](#_Toc504568404)

[Structure of the framework 11](#_Toc504568405)

[Building up the test cases – UI layer 15](#_Toc504568406)

[Building up the test cases – Functional layer 18](#_Toc504568407)

[Building up the test cases – Test cases 19](#_Toc504568408)

[Robot Framework – Variables 20](#_Toc504568409)

[Robot Framework – Most Important built in keywords and variables 21](#_Toc504568410)

[Database connection 21](#_Toc504568411)

[Test Data from external CSV files 24](#_Toc504568412)

[Links and Guides 26](#_Toc504568413)

[XPath basics 26](#_Toc504568414)

[Useful links 26](#_Toc504568415)

[Deployment notes 26](#_Toc504568416)

[Jenkins integration 27](#_Toc504568417)

[Prerequisites 27](#_Toc504568418)

[Installation 27](#_Toc504568419)

[Download and install 27](#_Toc504568420)

[Start Jenkins (headless – not recommended) 27](#_Toc504568421)

[Start Jenkins (non-headless - recommended) 28](#_Toc504568422)

[First startup 28](#_Toc504568423)

[Install plugins 28](#_Toc504568424)

[Configure system settings & variables 28](#_Toc504568425)

[Set up Jenkins job 29](#_Toc504568426)

[Add new item 29](#_Toc504568427)

[Select Freestyle project 29](#_Toc504568428)

[Give the job a name 30](#_Toc504568429)

[Set up Git repository (or other SCM) 30](#_Toc504568430)

[Timeouts 31](#_Toc504568431)

[Build command 31](#_Toc504568432)

[Setting up Robot Framework results 32](#_Toc504568433)

[Troubelshooting 33](#_Toc504568434)

[Build fails, no robot reports generated 33](#_Toc504568435)

[Test pass count is above the pass threshold, but build still fails. 34](#_Toc504568436)

[Error when opening log files 34](#_Toc504568437)

[Screenshots are not available in the logs. 34](#_Toc504568438)

# Document History

*Document location: QC repository*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Description** | **Date** | **Author** | **Reviewer** |
| 0.1 | Initial version of the document | 29/05/2017 | Szabolcs Hudak |  |
| 0.2 | Added AutoIt and Jenkins install notes | 1/9/2018 | Áron Fernbach |  |
| 0.3 | Fixed several mistakes in Jenkins installation. | 1/23/2018 | Áron Fernbach |  |

# Standards

## Introduction

The purpose of this document is to describe standards and guidelines to be followed when designing and developing framework code. This document will help ensure consistency across the code, resulting and increased usability and maintainability of the developed code.

The scope of this document is to provide standards and guidelines for designing and developing Test Automation Framework code for UI test automation.

This document provides you guidelines for:

* installation of Robot Framework IDE
* structure of the framework
* usage of variables and keywords
* general guidelines
* Jenkins integration

# Robot Framework tool

## Introduction

The Robot Framework IDE (RIDE) is the integrated development environment to implement automated tests for the Robot Framework. The Robot Framework is a generic test-automatation framework.

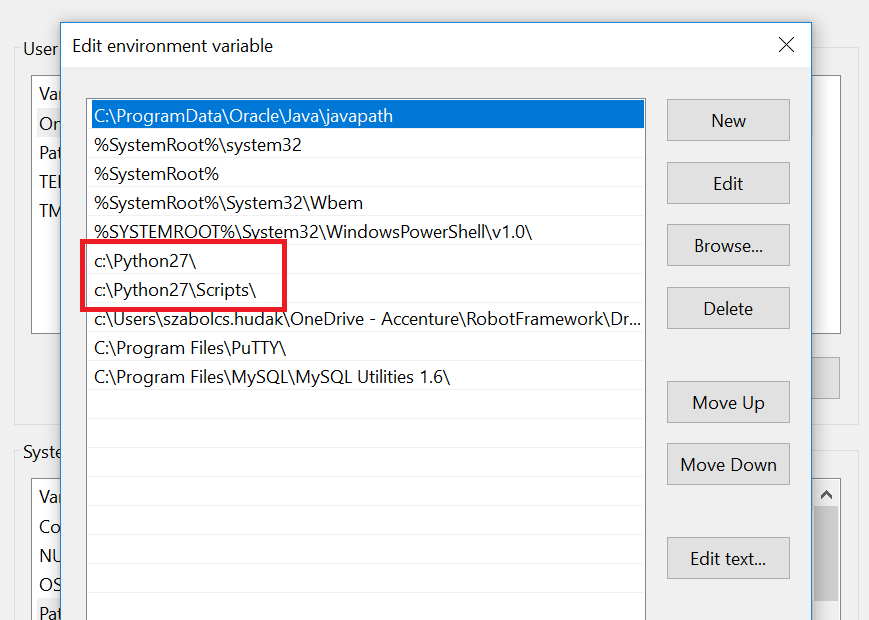
Robot Framework IDE supports functional testing of all browsers FireFox, Internet Explorer and Chrome using SeleniumLibrary for web testing. We can add different kind of libraries to the tool for different kind of testing tasks. For example testing the operation system, strings, collections, create database connection, etc…

## Installation

### Install Python

Download and install Python 2.7.1 (python-2.7.13.msi) from [http://python.org](http://python.org/). The target directory should be ***C:\Python27***

Edit the ***Path*** environmental variable, add ***C:\Python27\*** and ***C:\Python27\Scripts\*** to the ***Path***.



Download and install ***wxpython*** (install *wxPython2.8-win32-unicode-2.8.12.1-py27.exe*) from <https://wxpython.org/>. It is very important to download the ***32 bit*** version even you got ***64 bit*** machine.

### Install Robot Framework and extensions

Open a command line and install the different kind of python packages:

***pip install robotframework-ride***

***pip install robotframework-seleniumlibrary***

***pip install dialogs***

***pip install collections-extended***

***pip install pypiwin32***

***pip install robotframework-autoitlibrary***

AutoIt library is used to control the host OS, like clicking on window buttons, opening email client, etc. Its documentation is not straightforward:

* Keyword definitions for Robot framework can be found in the installation folder after **robotframework-autioitlibrary** is installed (C:/RobotFramework/Extensions/AutoItLibrary/AutoItLibrary.html on windows)
* Most of these keywords have no description whatsoever, so you will need to check the function docs at <https://www.autoitscript.com/autoit3/docs/functions.htm>.
* If cannot be found in functions, check the other docs at <https://www.autoitscript.com/autoit3/docs/>

Needless to say, if you want to test some 3rd party app in windows, the adequate software must be installed (e.g. Outlook)

### Install Chrome extensions

Install **SelAssist** from chrome web store. This will help identify elements by xpath, css or by any other [Locator](http://robotframework.org/SeleniumLibrary/SeleniumLibrary.html#Locating%20elements) in SeleniumLibrary.

### Install browser drivers

Next step, we must download and set up all of the drivers for different kind of browsers

Download and unzip the ***geckodriver*** for ***FireFox*** testing from this link <https://github.com/mozilla/geckodriver/releases>

Download and unzip the ***chromedriver*** for ***Google Chrome*** testing from this link <https://sites.google.com/a/chromium.org/chromedriver/downloads>

Download and unzip the ***iedriver*** for ***Internet Explorer*** testing from this [link http://selenium-release.storage.googleapis.com/index.html](../../AppData/Local/Microsoft/Windows/INetCache/Content.Outlook/6VVKOG4L/link%20http:/selenium-release.storage.googleapis.com/index.html)

Caution! Before IE11 can be used, there are several things that should be configured, and you **must use the 3.4. version of IEDRiverServer.exe** for it to work. Follow [this configuration guide](https://github.com/SeleniumHQ/selenium/wiki/InternetExplorerDriver#required-configuration). If you use an Accenture machine, there is a high possibility that the firewall will block the site you want to test. Contact administrators to solve this problem. Even after these are solved, IE will still be painfully slow and unstable. Be prepared!

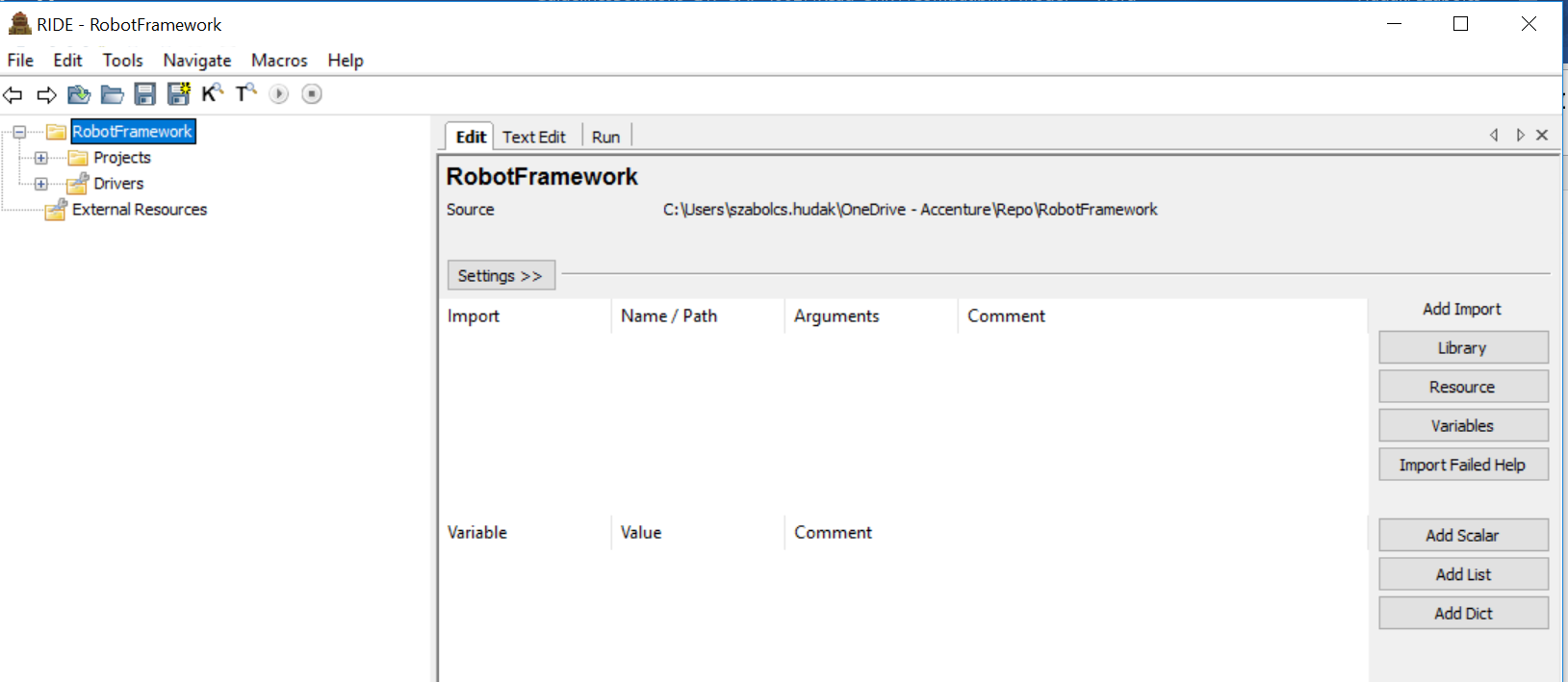
After that, copy all the drivers in one **common library\*** and set this to the system path.

*\*I’ve set C:\Selenium\Drivers but you can use any path.* [Jenkins install notes](#_Installation) *uses this directory, so replace it with your own accordingly.*

## First steps in Robot Framework IDE (RIDE)

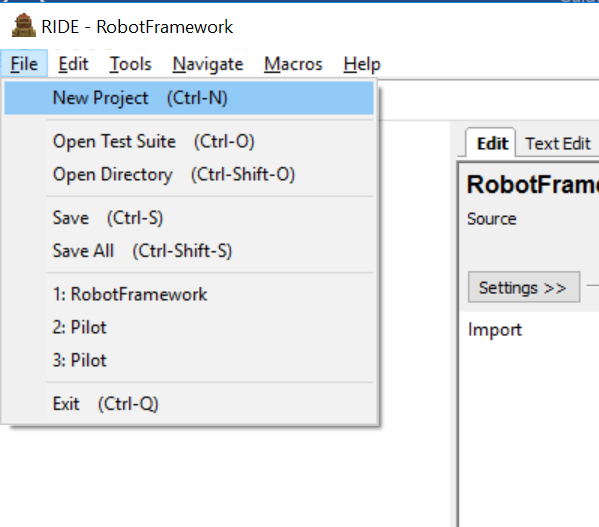
### Start RIDE

Open Robot Framework IDE from command line. Type ***ride.py***

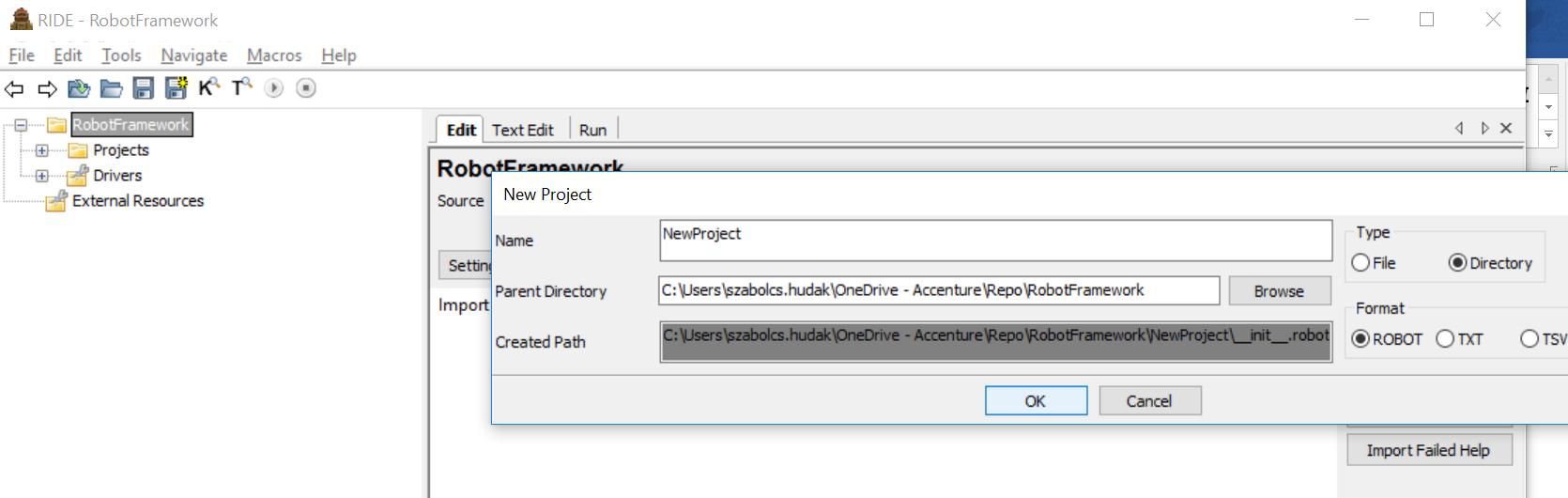


### Create a project

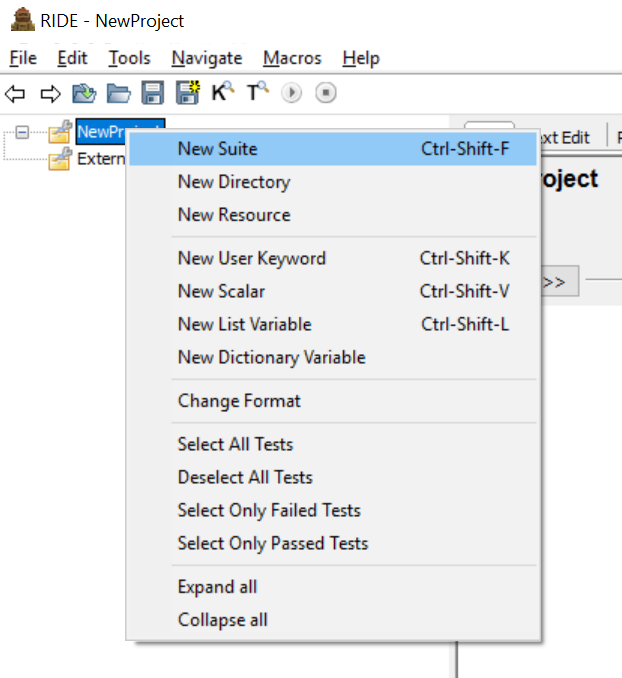
Open ***File*** – ***New Project***



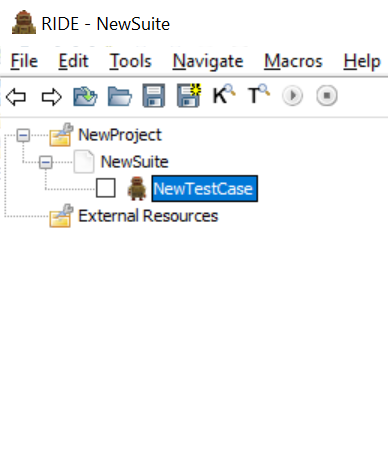
Make a new project called ***NewProject*** and press ***OK***



Right Click on ***NewProject*** and Select ***New Suite***. Call this ***NewSuite*** and press ***OK***

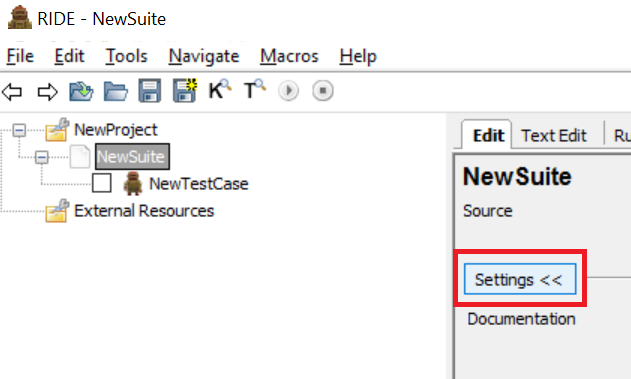


Right Click on ***New Suite*** and press ***New Test Case.*** Call this ***NewTestCase*** and press ***OK***. You will get the following structure.

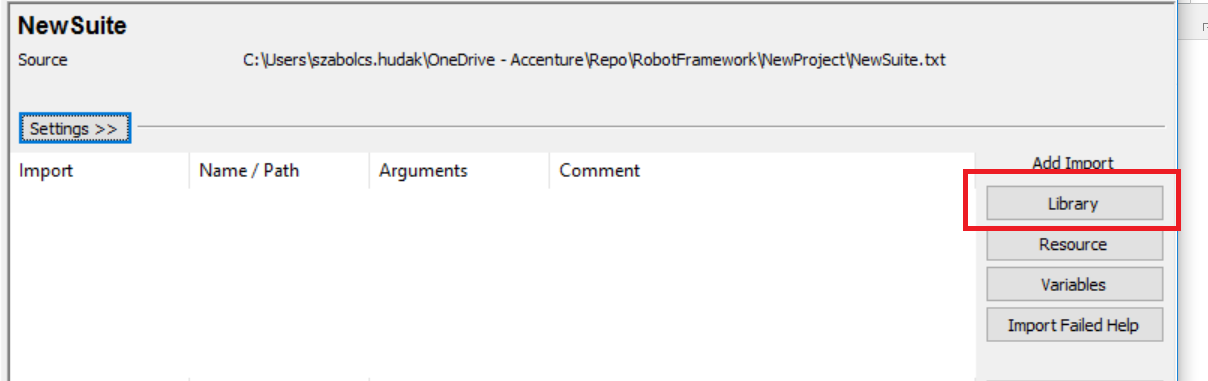


### Add libraries

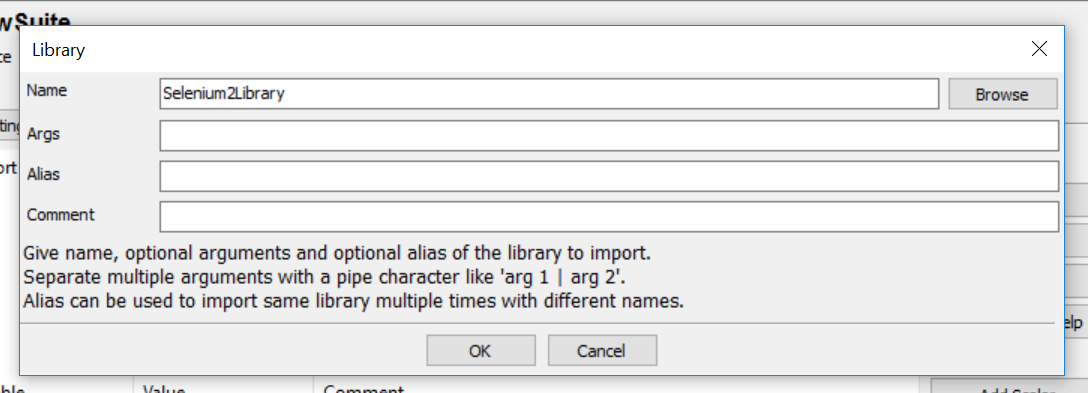
Next step is to make usable the python libraries which we installed previously with pip installation. Click on the ***NewSuite*** in the tree. Click on ***Settings***



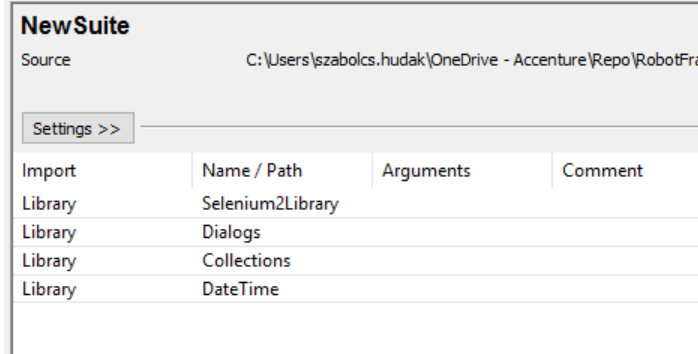
Add the installed libraries by pressing the ***Library*** button



Add the name of the library and press ***OK***

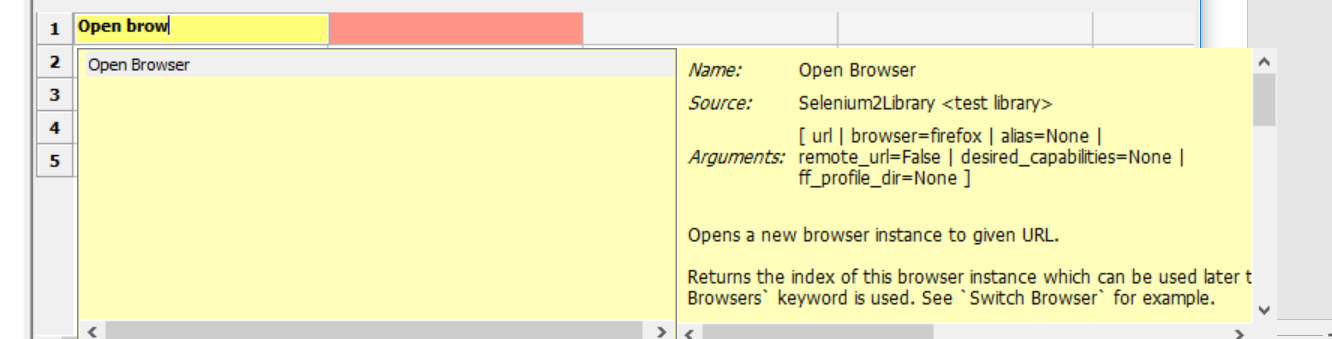


Add the following libraries in this way

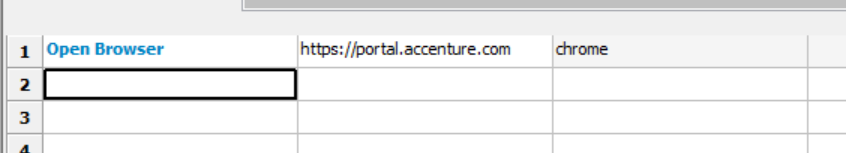


### Create and run the first test

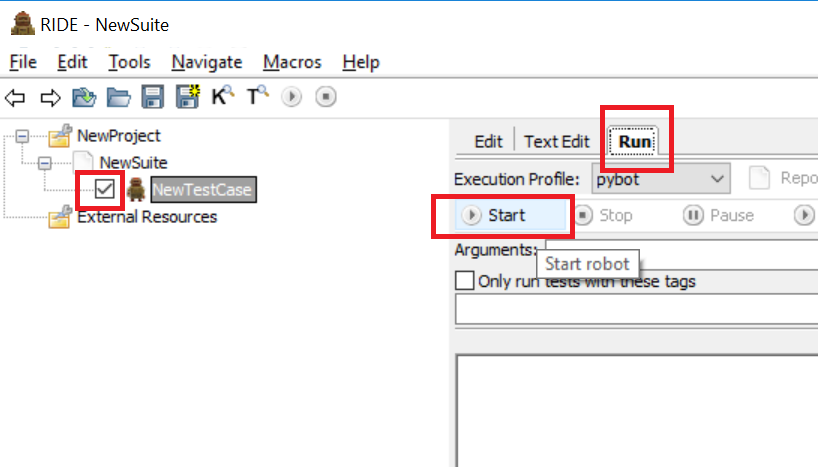
Now we create our first test case. Press on the ***NewTestCase***. In the grid, you can add your test steps one after another. First we will open a browser. Start typeing ***Open Bro***… and press ***CTRL+SPACE***



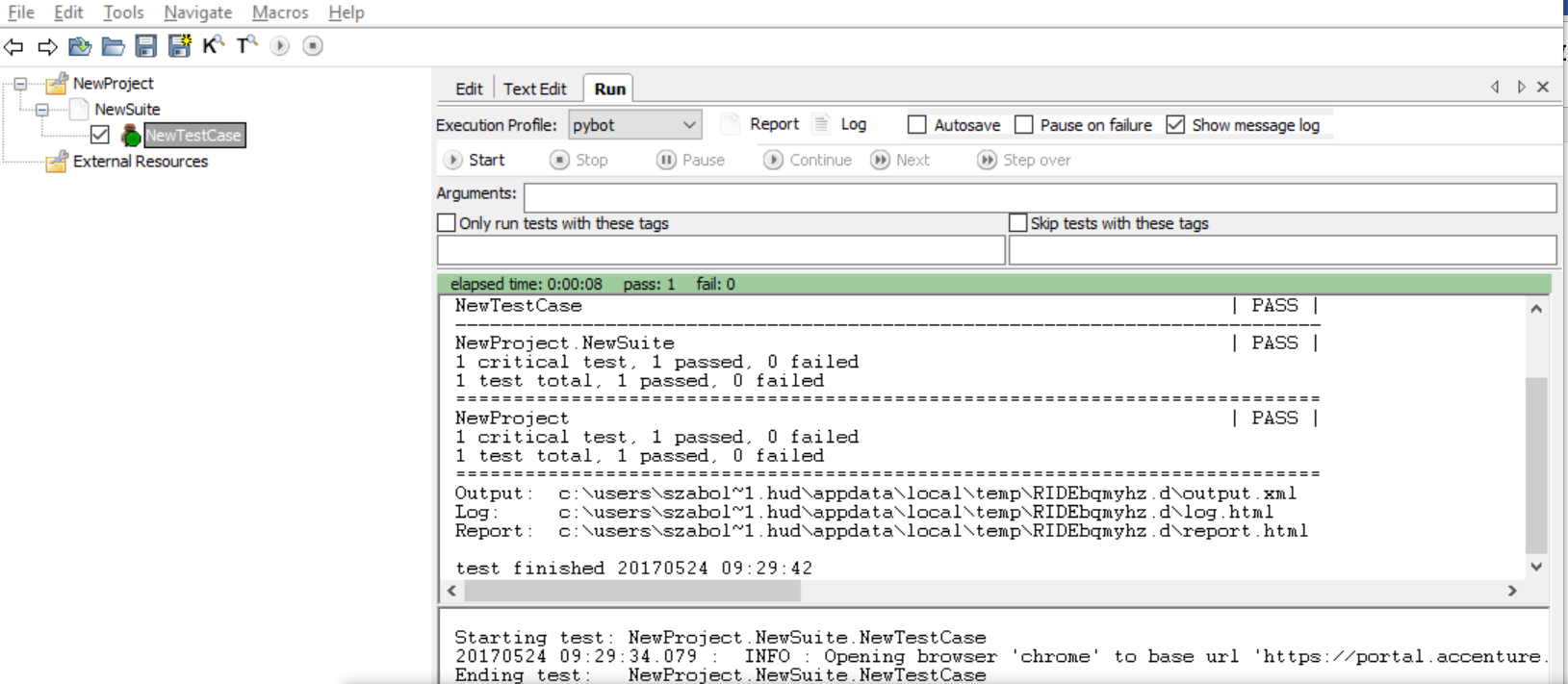
You will see a shortcut of the ***Open Browser*** keyword from the ***SeleniumLibrary***. Fill the mandatory parameters: ***url*** and the ***browser***



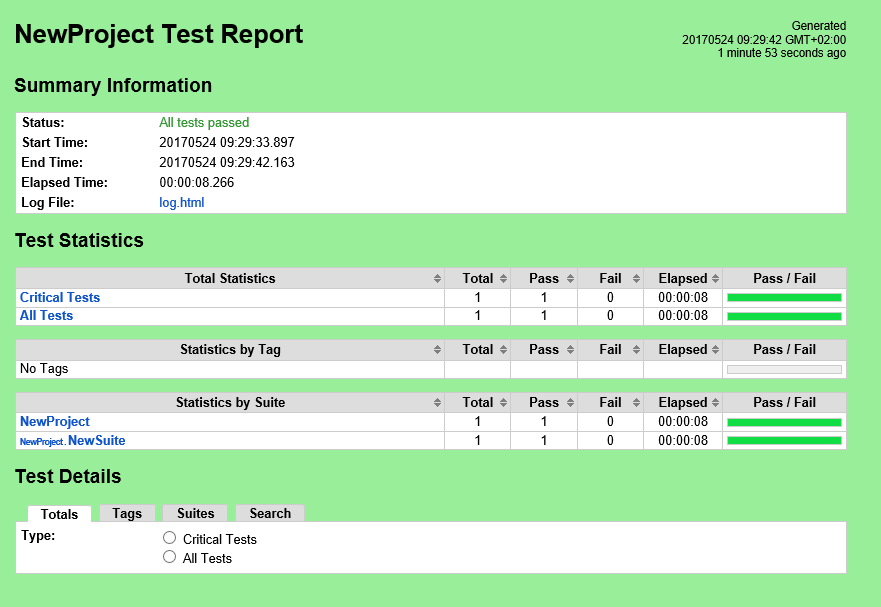
To run our test, select the test case in the tree view and press the ***Run*** tab. On the ***Run*** tab press the ***Start*** button

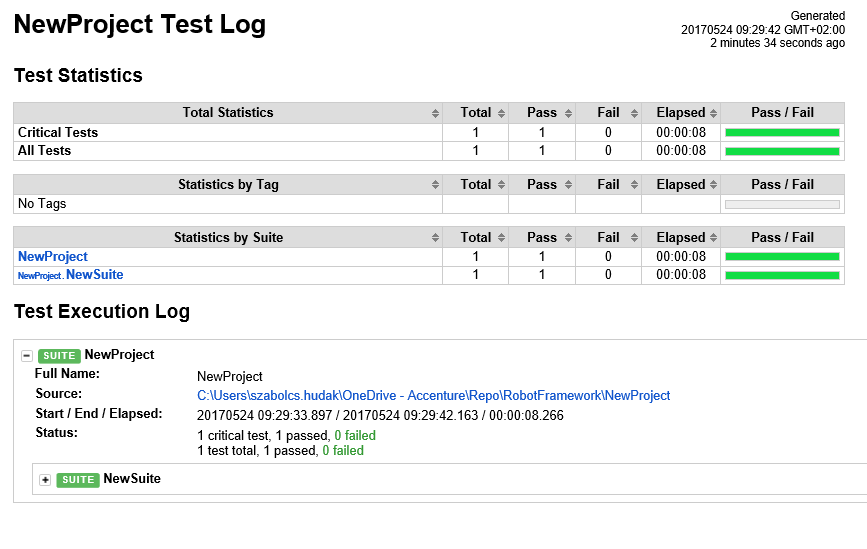


You will see the result of the test execution



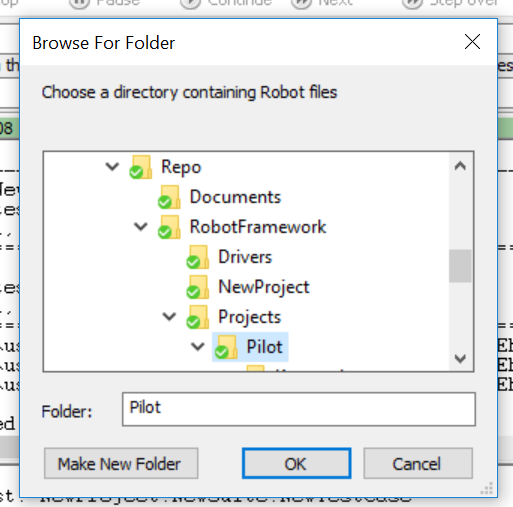
By pressing on the ***Log*** or ***Report*** button you will see the detailed result of the test run.



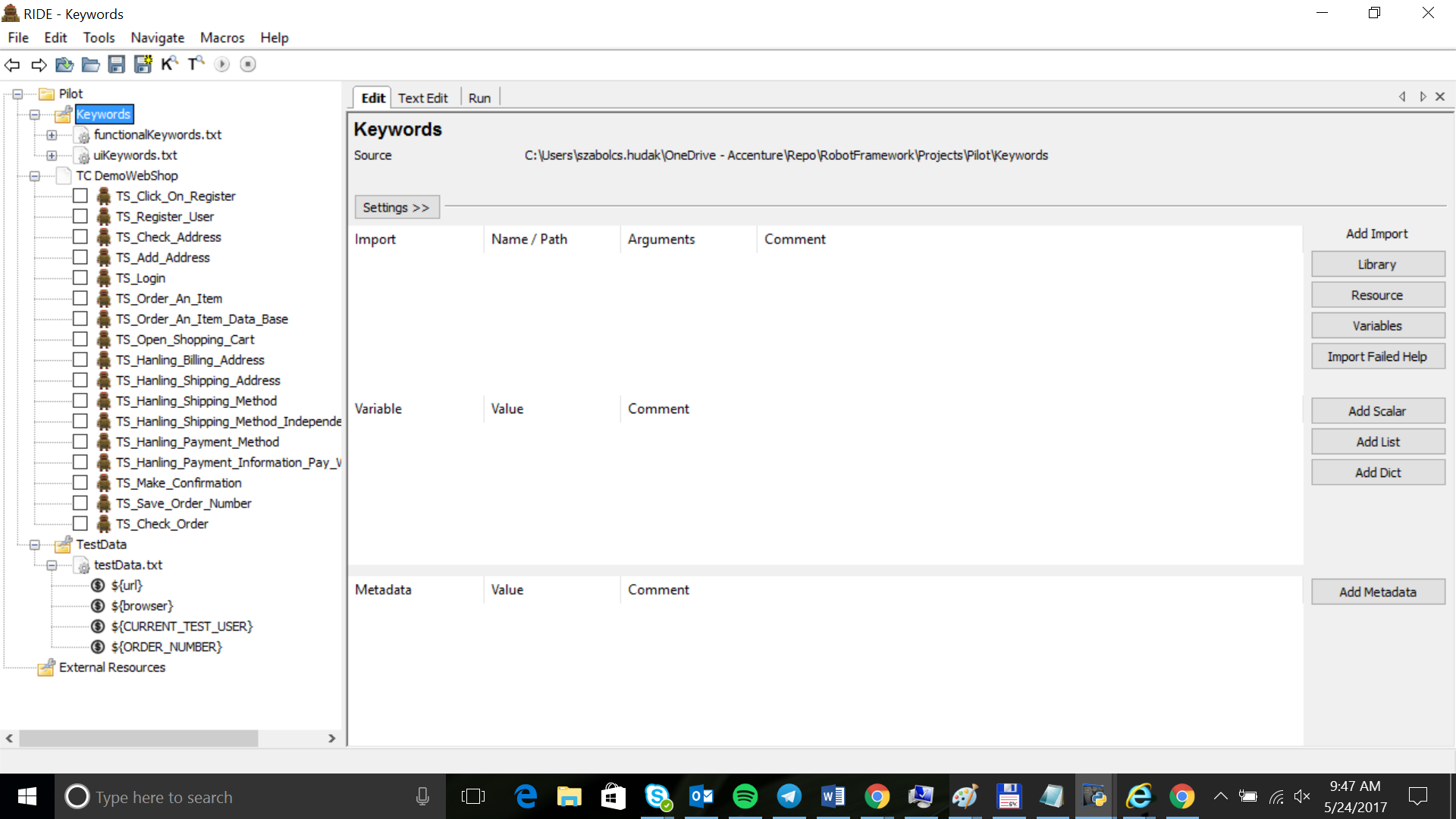


## Structure of the framework

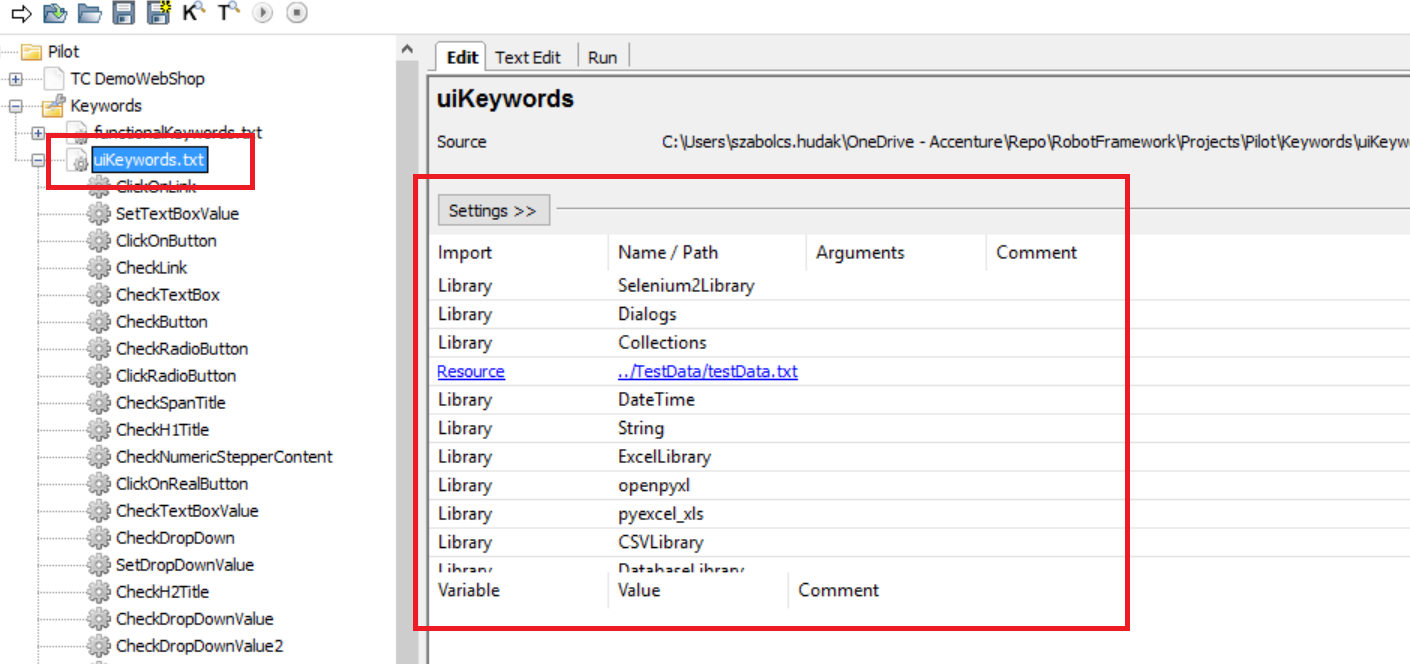
From the repository open our Pilot project. ***File*** – ***Open Directory***. Select the ***Pilot*** project and press ***OK***



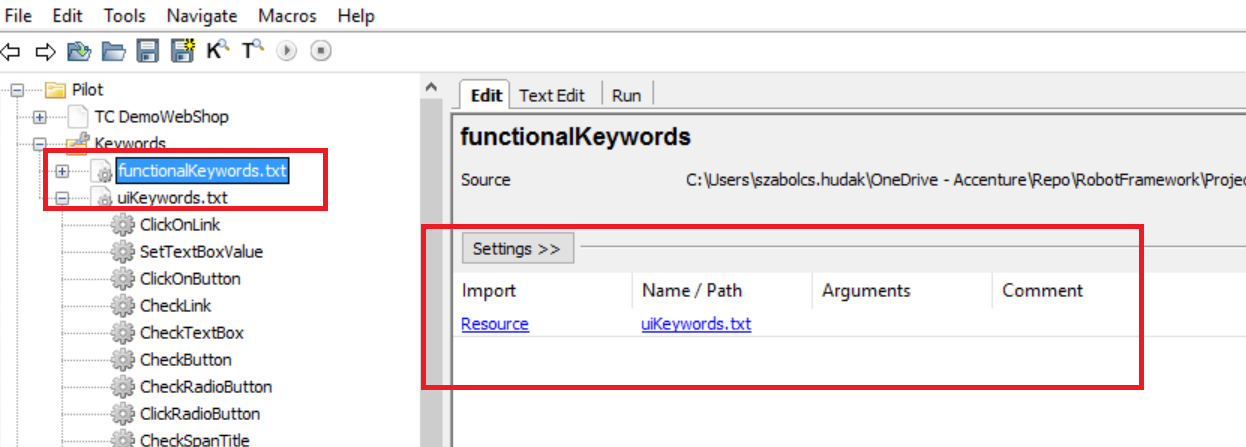
On the tree view you can see the structure. Under the TC ***DemoWebshop*** there are the test cases itself. On the Keywords library you can see the 2 logical level ***functionalKeywords*** and ***uiKeywords***. These resource files contain our keywords to handle the logical functionalities and the ui layer itself. Under test data you can find the global variables necessary for testing.



We built up our test framework solution from 3 different layers. The first layer is the ui layer. This layer contains keywords which are directly handle the elements of the user interface in the browser. For example checking, clicking, typing into the elements, etc… For the identification we use ***xPath***. Our ui layer (***uiKeywords.txt***) directly uses the different kind of python packages (***SeleniumLibrary***, ***Dialogs***, ***Collections***, ***CSVLibrary***, ***DatabaseLibrary***,…) and the built in keywords of these packages.

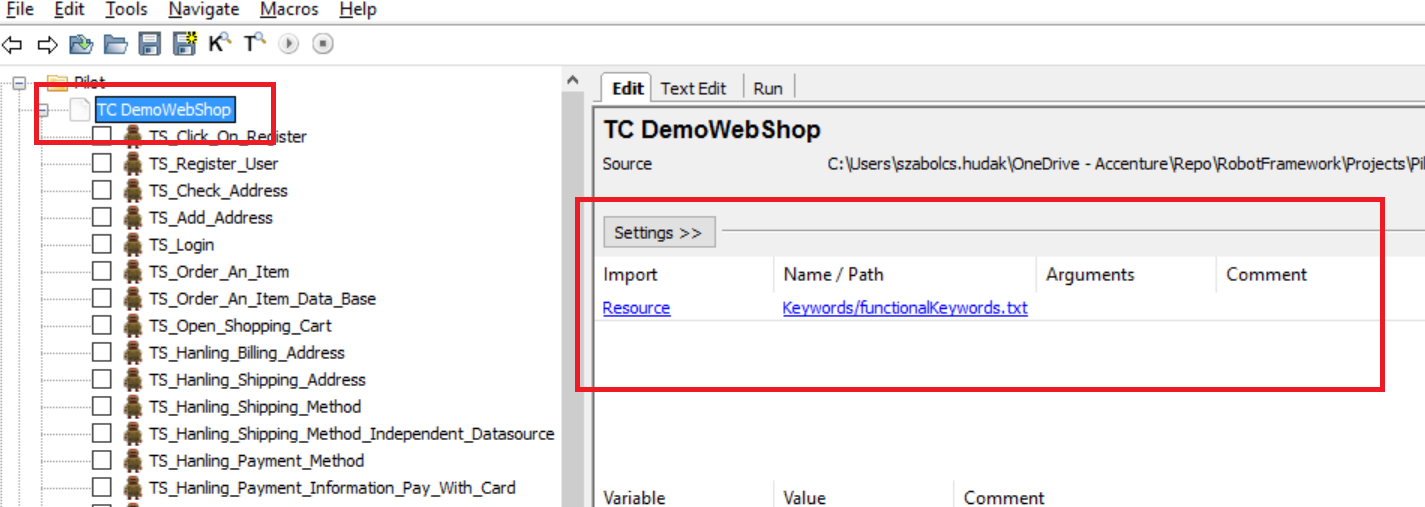


The next level is the functional layer (***functionalKeywords.txt***). This contains keyword which are simulating more complex user interactions, functional and technical workflows.



We can call all of the keywords from the ui layer, and through this we can use all of the python packages.

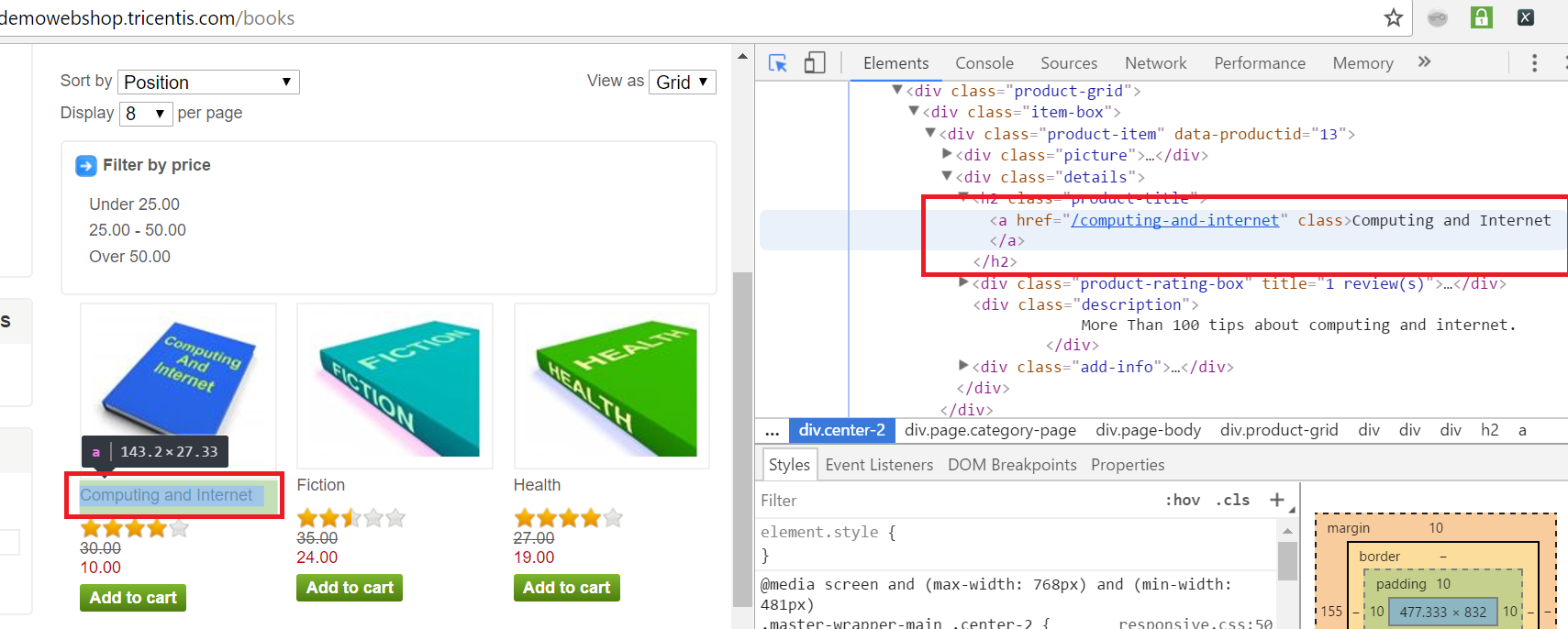
We store the test cases itself on the highest level. We use the functional layer as a resource, and the keywords of the functional layer as test case templates



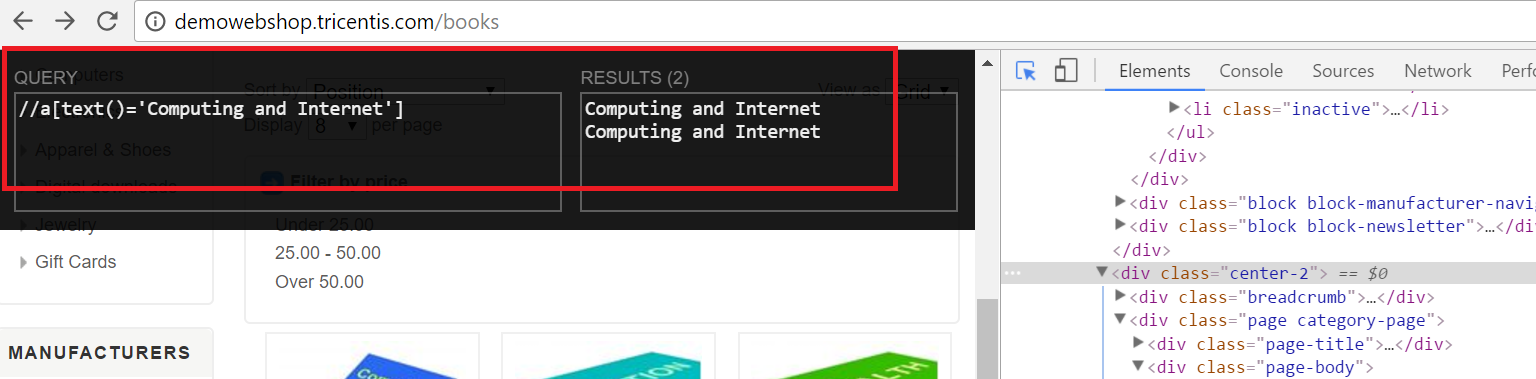
### Building up the test cases – UI layer

The step we wanted to automate is to select the ***Books*** category from the webshop application. Select the ***Computing and Internet*** book and press the ***Add to cart*** button. And check if the shopping cart shows ***1***.

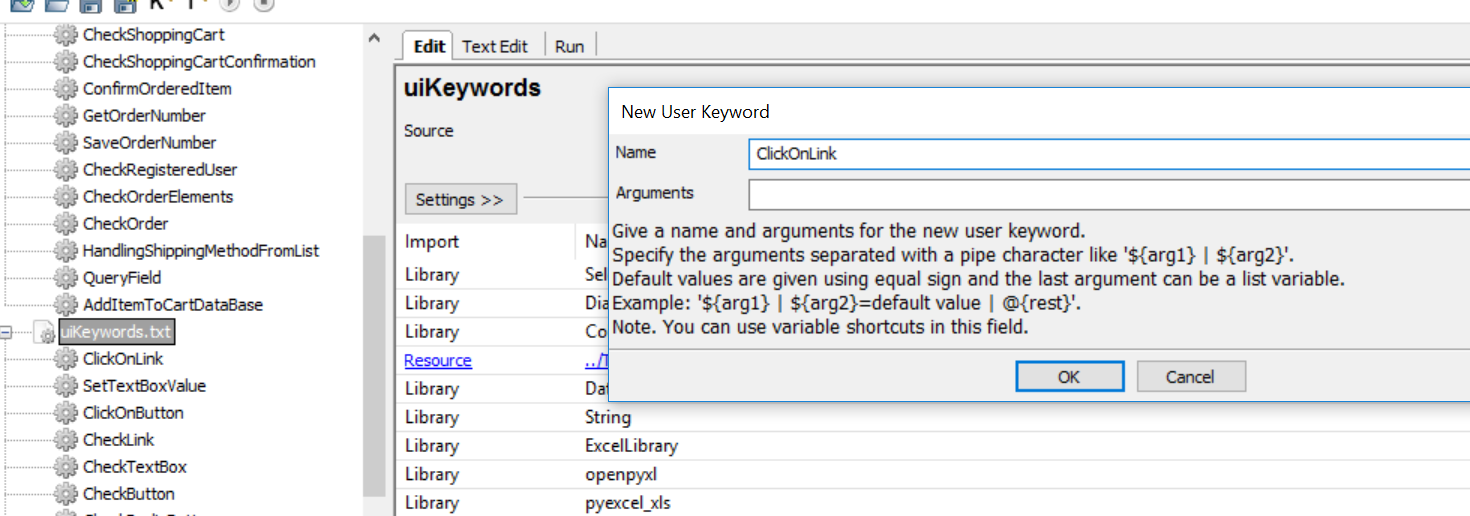
First step we localize the elements manually in the browser (Chrome). Open the console in the browser by pressing ***F12***. For example we want to localize the ***Computing and Internet*** link on the left, because we wanted to press on it. Search the element in the HTML DOM with the inspector



After that open the ***xPath Helper ,*** and first create one unique ***xPath*** expression for the element



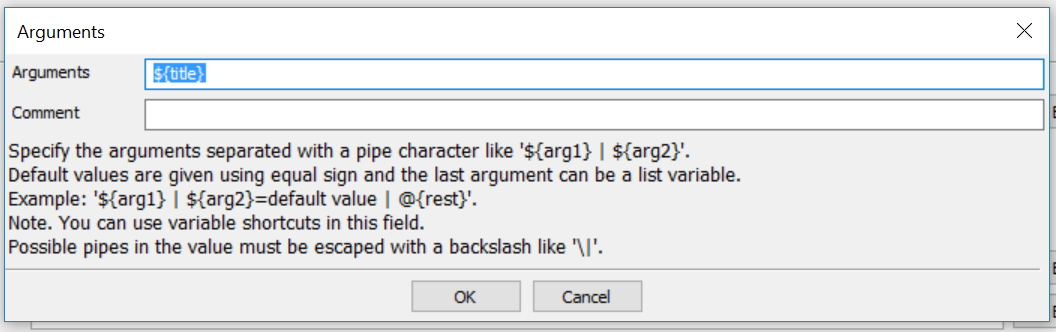
Our next step will to create one ui related keyword which will directly handle (click on it) any kind of links, not just this one. Right click on the ***uiKeyWords.txt*** resource file and select ***New User Keyword***. Call the new keyword ***ClickOnLink*** and press ***OK***.



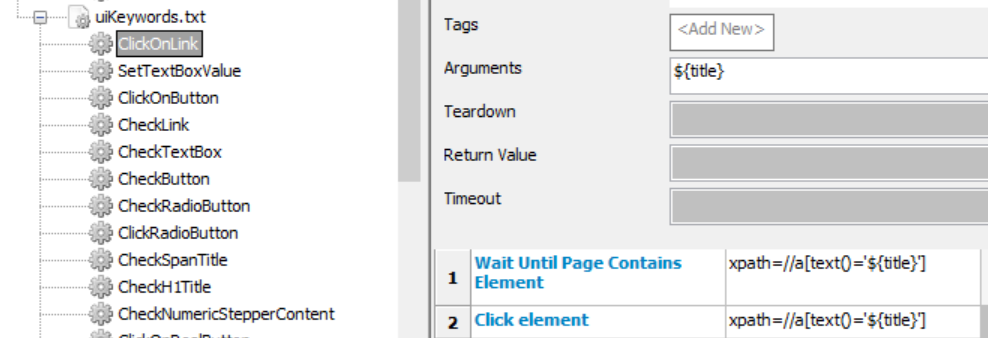
Make an universal ***xPath*** by replacing ***Computing and Internet*** with variable ***${title}***. This will be the input parameter of this keyword.

*//a[text()='Computing and Internet']* => *xpath=//a[text()='****${title}****']*

Add ***${title}*** as an argument of the keyword



Use the built in ***Selenium*** keywords to simulate the user interaction. First we have to wait until this element appears on the page. We can use the ***Wait Until Page Contains Element*** keyword. Use our universal ***xPath*** for the location parameter. After that the user clicks on this element. We can use the ***Click Element*** built in selenium keyword to do this with our universal ***xPath*** too.

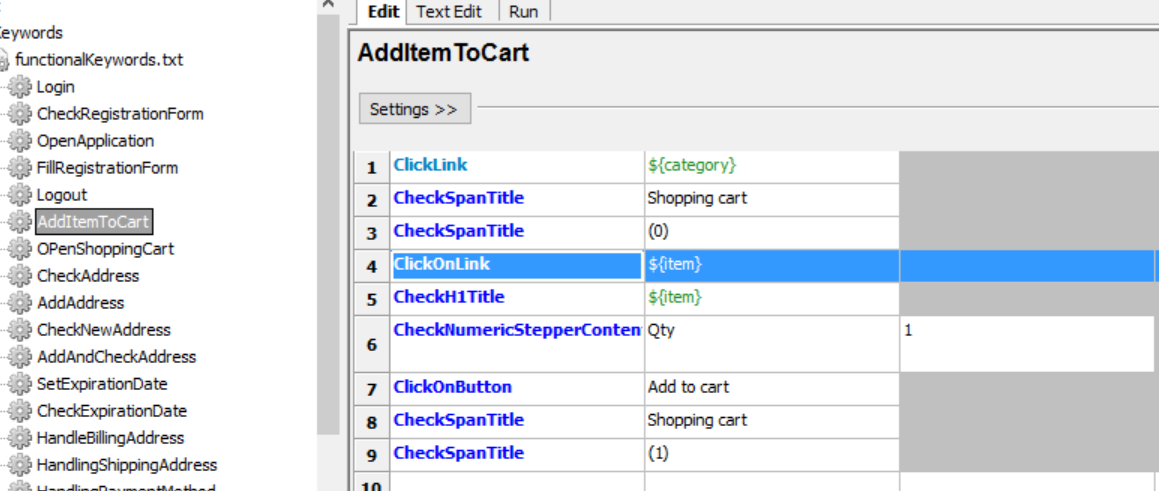


In this way, we created an universal keyword which can click on every link with name of ***${title}***.

With the keywords of the ***uiKeyWords.txt*** resource we can handle the ui elements directly in the same way.

### Building up the test cases – Functional layer

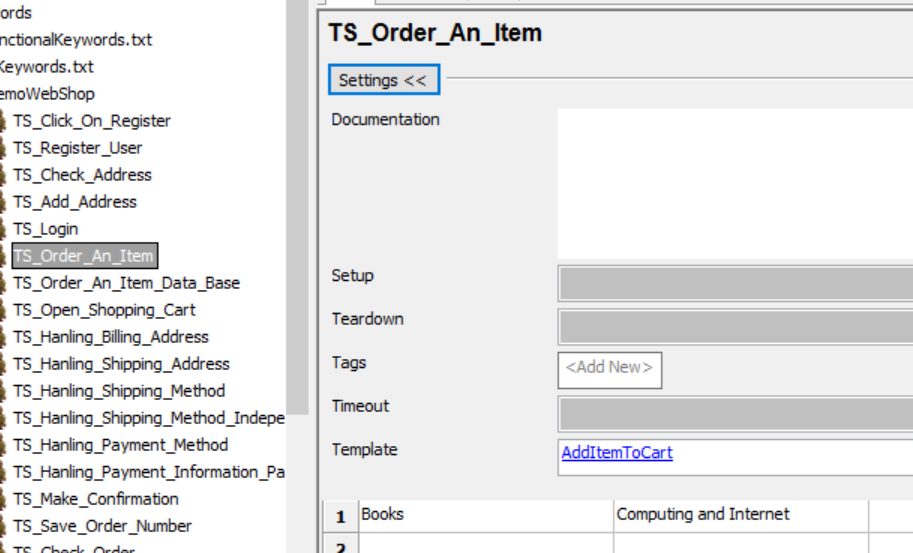
In this level we implement the keyword which describes a functionality or workflow itself, calling the reusable keywords from the ui layer.



This keyword adds an item to the shopping cart. Clicking on the category, checking the title of the page, clicking on the item in category, checking the title of the item, pressing the ***Add to cart*** button and checking if the number of the element in the cart increased by one. You can see the steps one after another. Every step is a keyword call with parameters from the ui layer or directly from the ***Selenium library***. In the line 4 we call the previously created ***ClickOnLink*** keyword from the ui layer. This ***AddItemCart*** keyword got 2 input parameters what passes away to the keywords of the ui layer.

### Building up the test cases – Test cases

We can build up our test cases with keywords from functional layer, using these as test case templates.

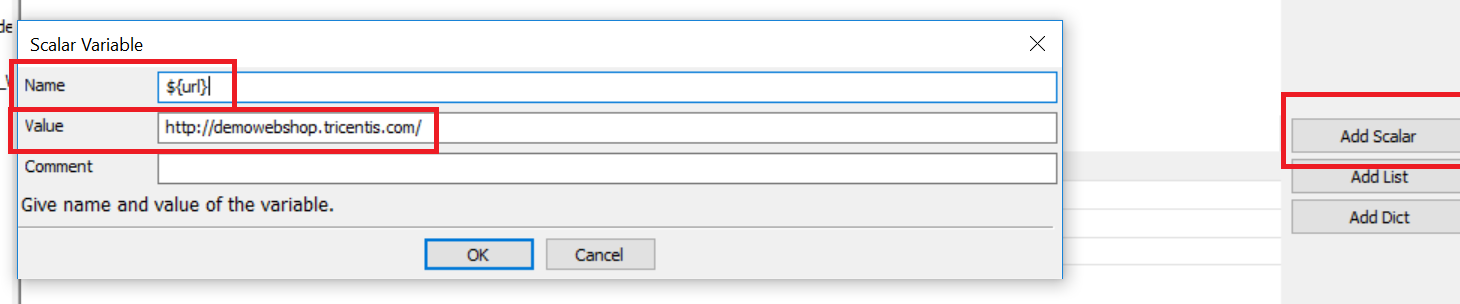


Fill the Template field with a keyword name. In this case this will the ***AddItemCart***. The first row will contains the two parameters of the **AddItemCart**, so we will call this keyword with these parameters. If we got more rows the ***AddItemCart*** keyword will be executed many times (one row, one execution, one variation of the test case). This grid will be the data table, where the separated test data will be stored belong to the current test case.

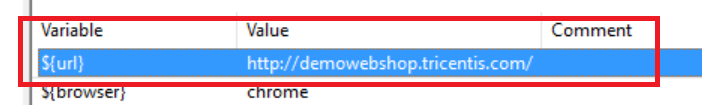
### Robot Framework – Variables

Variables can be defined on ***test scenario level*** and ***resource file level*** too. But you can define a variable ***inside one keyword*** too. The visibility of the variable depends on where we defined them. For example, variable defined inside a keyword exists only inside this keyword, etc…

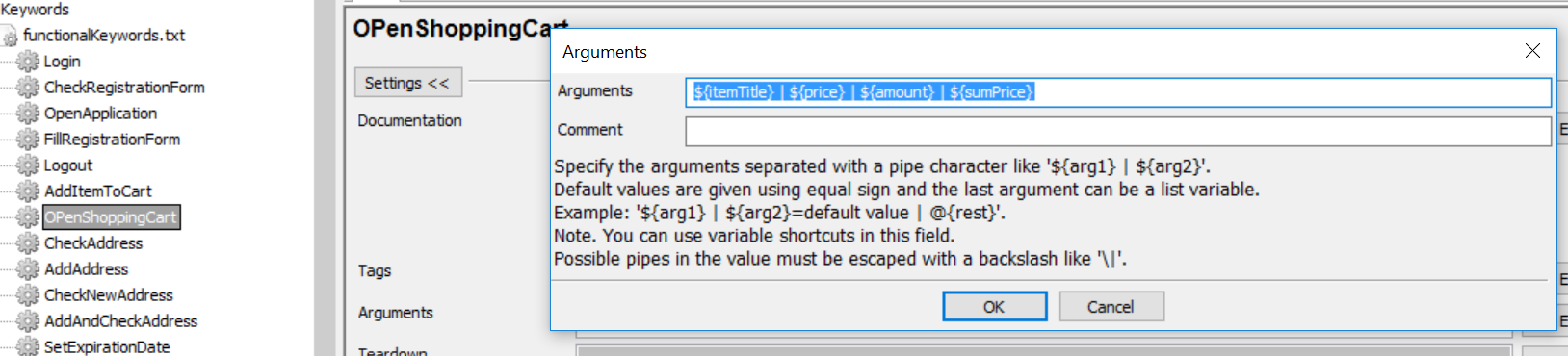
To define a variable on scenario or resource level select the scenario or the resource and press ***Add scalar*** button



Press ***Add Scalar***, add a name and a value and press OK.



Define variable on ***keyword level***



Open the keyword definition. Press in the ***Arguments*** field and define variables here separated by |. If you want to give default value to a variable => ***${sumPrice}=10***

If you want a variable to be ***global*** define it in a resource file and link this file everywhere when you want to use this variable. Give value to a global variable this way:

***Set Global Variable*** *${variableName} value*

### Robot Framework – Most Important built in keywords and variables

***${CURDIR}*** *- the current directory, directory of the project*

***Pause Execution*** *– pauses the test execution*

***Open Browser*** *– opens one browser instance*

***Close Browser*** *– closes the opened browser instance*

***Click Link*** *– clicks on a link*

***Click Element*** *– clicks on an element*

***Click Button*** *– clicks on a button*

***Element Text Should Be*** *– compares the text of the element with a current value*

***Get Text*** *– returns with the text of an element*

***Wait Until Page Contains*** *– waits until page contains a text*

***Wait Until Page Contains Element –*** *waits until page contains an element*

***Sleep*** *– waits for a current time period*

**HTML elements**

***a*** *– link*

***label*** *– label*

***span*** *– label*

***div*** *– label*

***input*** *– textfield, checkbox, radio button, button*

***button*** *- button*

***select*** *– list*

***h1*** *–* ***h2*** *– header element*

***td*** *– table cell*

***tr*** *– table row*

***table*** *– html table*

***img*** *– image, picture*

## Database connection

We can store our test data in databases. For database connection first we need to install two python packages: ***DatabaseLibrary*** and ***pymysql*** library.

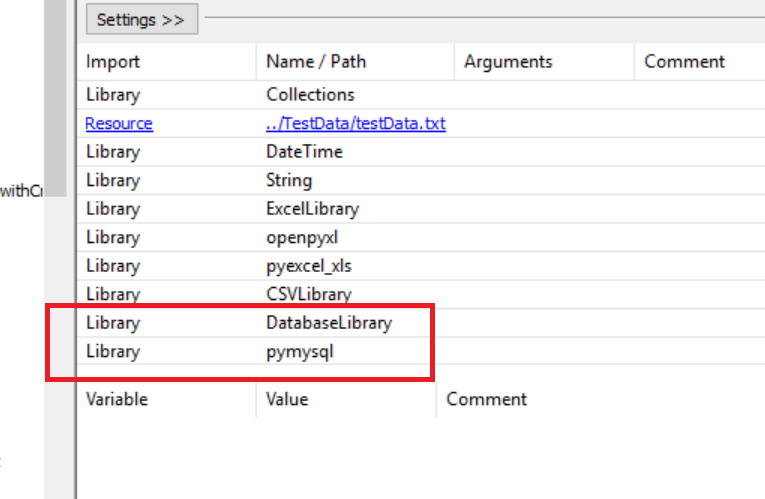
***pip install robotframework-databaselibrary***

Download ***pymysql*** from <https://pypi.python.org/pypi/PyMySQL/0.7.11>

And install this from the downloaded library

***python setup.py install***

Next step to add the installed library to the project



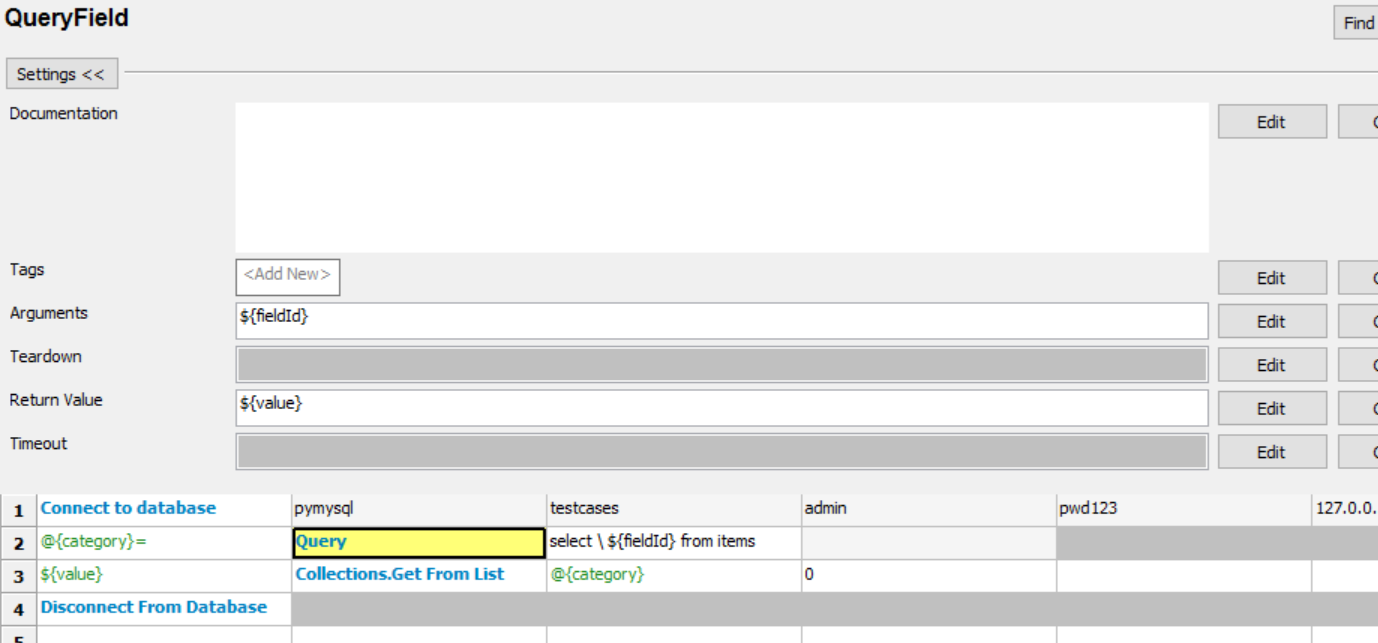
We got a sample mysql database called ***testcases***. In this we got a data table called ***items***. First we build up a database connection with the wollowing built in keyword…

***Connect to database*** *pymysql testcases admin pwd123 127.0.0.1 3306*

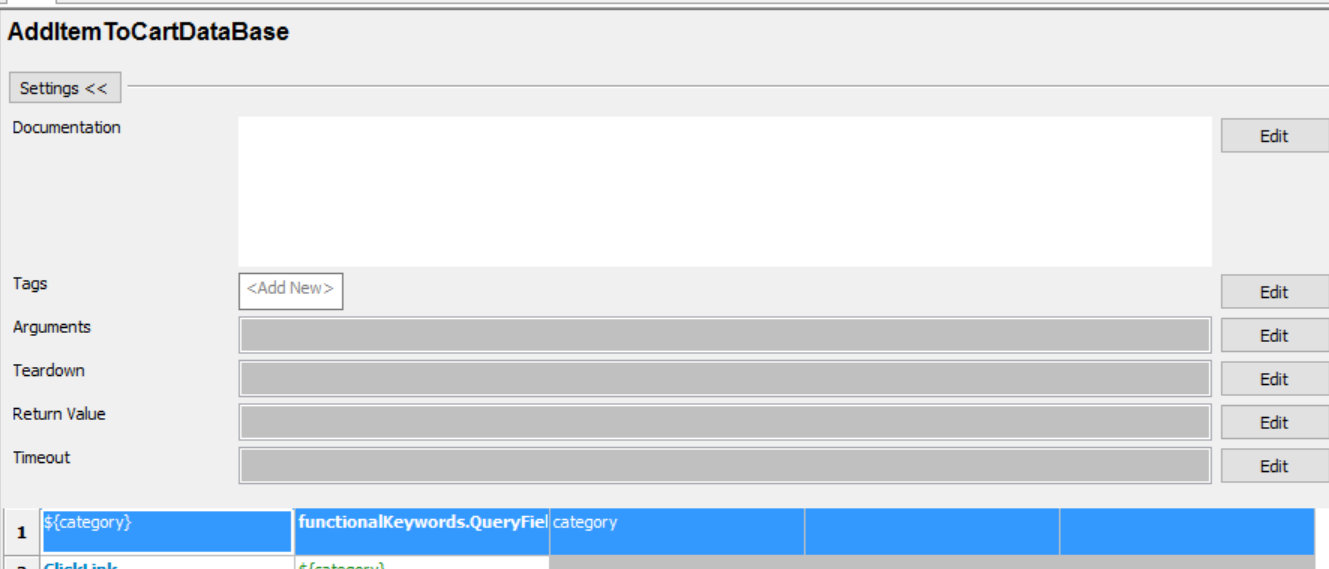
Where ***pymsql*** is the connection protocol, ***testcases*** the name of the *database*, ***admin*** is the *root user*, ***pwd123*** is the *password*, ***127.0.0.1*** is the *ip address* of the database server and ***3306*** is the *port* what we using.

In the next line with the ***Query*** keyword we make a query from the items table (the ***${fieldId}*** field).

At the end the keyword returns with the result of the query ***${value}***.



We use this keyword by calling this with the name of the field we want to use from the table as test data.



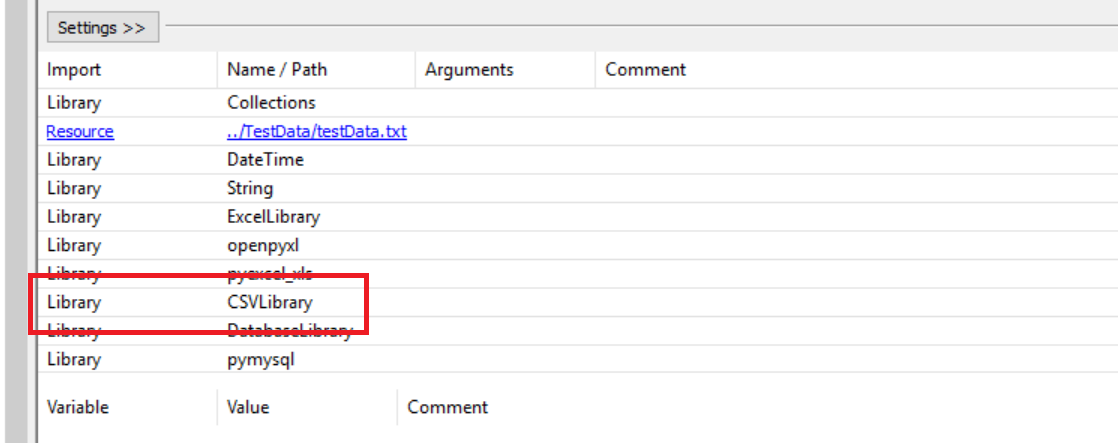
## Test Data from external CSV files

This tool provides a possibility to get test data from external data files. The ***excel modul*** currently is ***not working*** but we can use ***csv*** file instead of this.

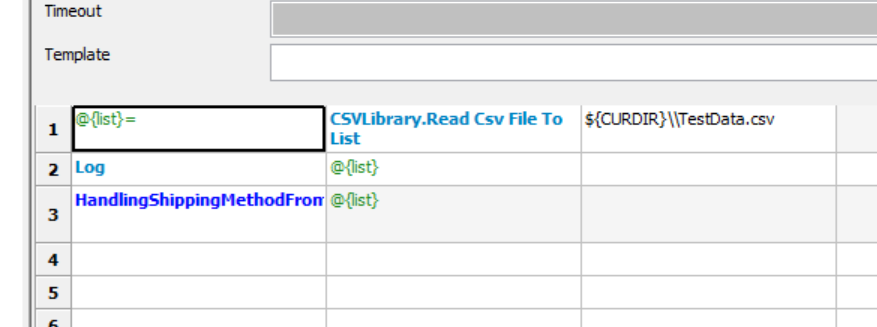
First we have to install our ***csv library***

***pip install robotframework-CSVLibrary***

Next step to configure this library into ***Robot Framework IDE***

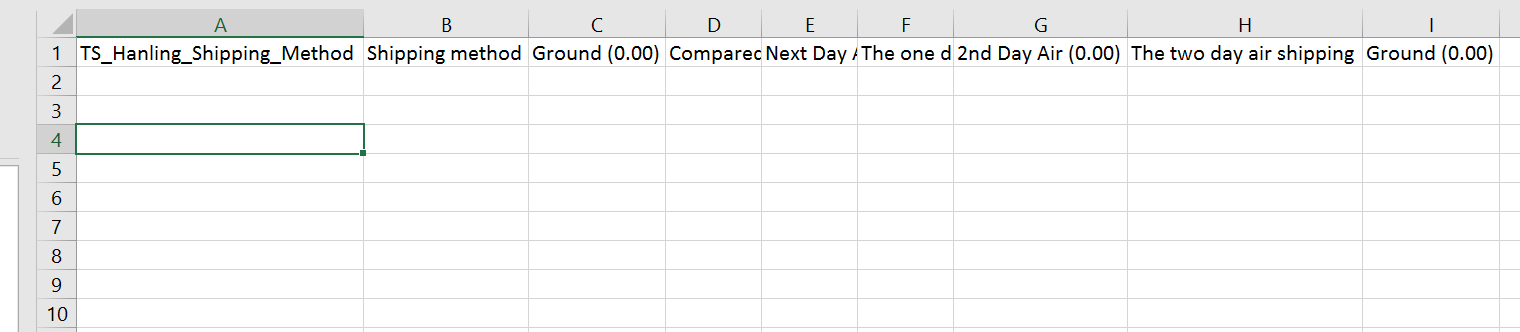


Open the ***TS\_Hanling\_Shipping\_Method\_Independent\_Datasource*** test case in the tree view.

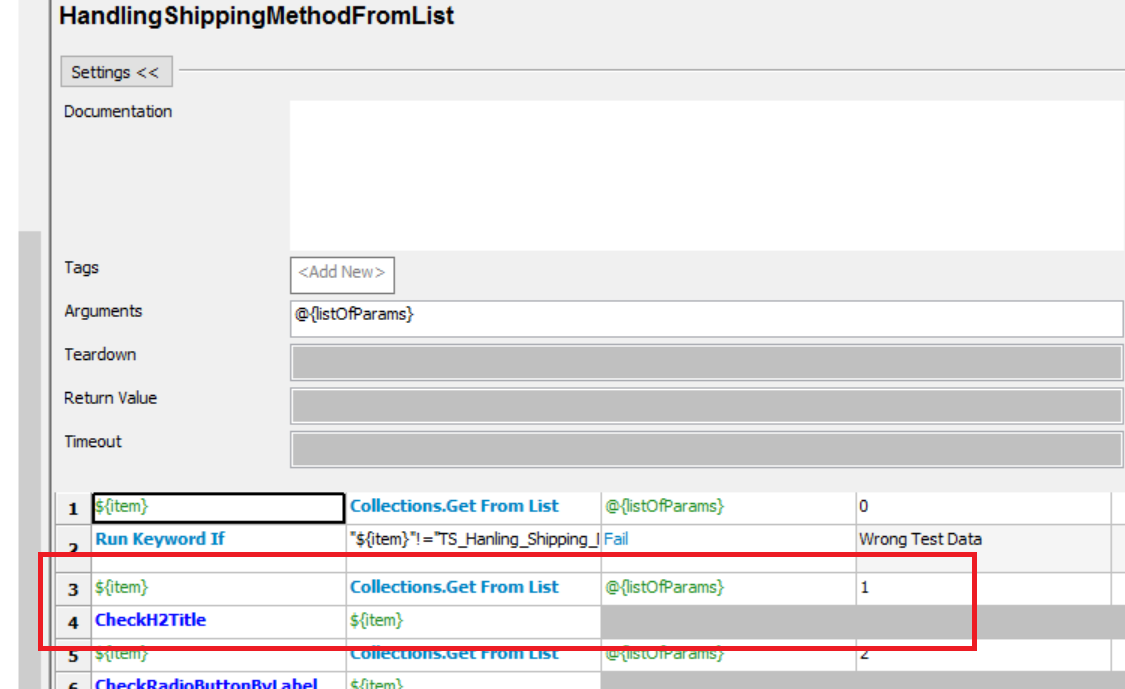


We can read one row from our external data file called ***TestData.csv*** with the ***Read Csv File To List*** built in keyword. After that we can pass this list as a list of input parameters to our ***HandlingShoppingMethodsFromList*** keyword.

Look at the content of the ***TestData.csv*** itself.



We pass these parameters to our keyword. Our keyword will gain the information needed from this list.



For example the content of the ***B1*** cell will be the second item (***Shopping method***) of the parameter list we gained from the csv file. With ***Get From List*** built in keyword we gain this item. The first parameter is the list itself and the second one is the index of the item we want to use. After that we pass this item (***Shopping method***) to our ***CheckH2Title*** keyword which will check if a ***H2*** tag appears on the page with text ***Shopping method***.

# Links and Guides

## XPath basics

With different kind of ***xpath*** expressions you can localize elements in the ***HTML DOM.*** See following simple examples…

*//a* - found every link on the page

*//\** - found every element on the page

*//a[@id=’****element\_id****’]* -found the link on the page with id ***element\_id***

//\*[@id=’**element\_id**’] -found all elements on the page with id ***element\_id***

//a[@class=’**element\_class**’] -found the link on the page with css ***element\_class***

//\*[@class=’**element\_class**’] -found all elements on the page with css ***element\_class***

//\*[@id="**fox**"]/a -found the first link child element of all elements with id ***fox*** on the page

//\*[@id="**fox**"]/../ -found the parent element of all elements with id ***fox*** on the page

//button(contains(., '**press me**')] -found the first button element on the page which text contains ***press me***

//input[@value='**press me**'] - found the first input button element on the page which text contains ***press me***

//\*[text()=*'****the visible text****'*] -found all elements on the page which got ***the visible text*** text

(//a)[3] -found the third link element on the page

(//input[@type="***text***"])[4] -found the fourth textbox element on the page

(//div[@class=*'****abc****'*])[2]/a -found the first link child element of the second div element with ***abc*** css on the page

## Useful links

* Robot Framework users guide – <http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html>
* Robot Framework IDE tutorial – <https://www.youtube.com/watch?v=6F_xGKdoN1E>
* XPath examples – <https://msdn.microsoft.com/en-us/library/ms256086(v=vs.110).aspx>
* Extended Selenium 2 Library – https://github.com/rickypc/robotframework-extendedselenium2library
  + Contains useful keywords like "scroll to element" which otherwise you have to write JavaScript for
  + Downside is it’s built on a previous version of SeleniumLibrary: Selenium2Library.. (yep, Selenium2Library is older than SeleniumLibarary. Makes perfect sense.)

## Deployment notes

* Ensure that target environment has all the software / tools installed
* Set all timeouts to 15 sec minimum. Most test environments are run on a VM which is considerably slower than a developer laptop. If too many tests fail due to timeout issues, try setting it to a higher value.

# Jenkins integration

Jenkins is a free job runner and CI tool. Basically any CI tool can run Robot Framework because it’s a simple command line tool. What makes Jenkins special is the robot framework plugin which generates visual representations of the tests and also makes the logs available for viewing.

## Prerequisites

All of these should be installed:

* Git
* Anything the test project needs:
  + Python
  + Robot Framework
  + Selenium
  + Robotframework libraries
  + Browser drivers
  + 3rd party apps (e.g. email client)

## Installation

### Download and install

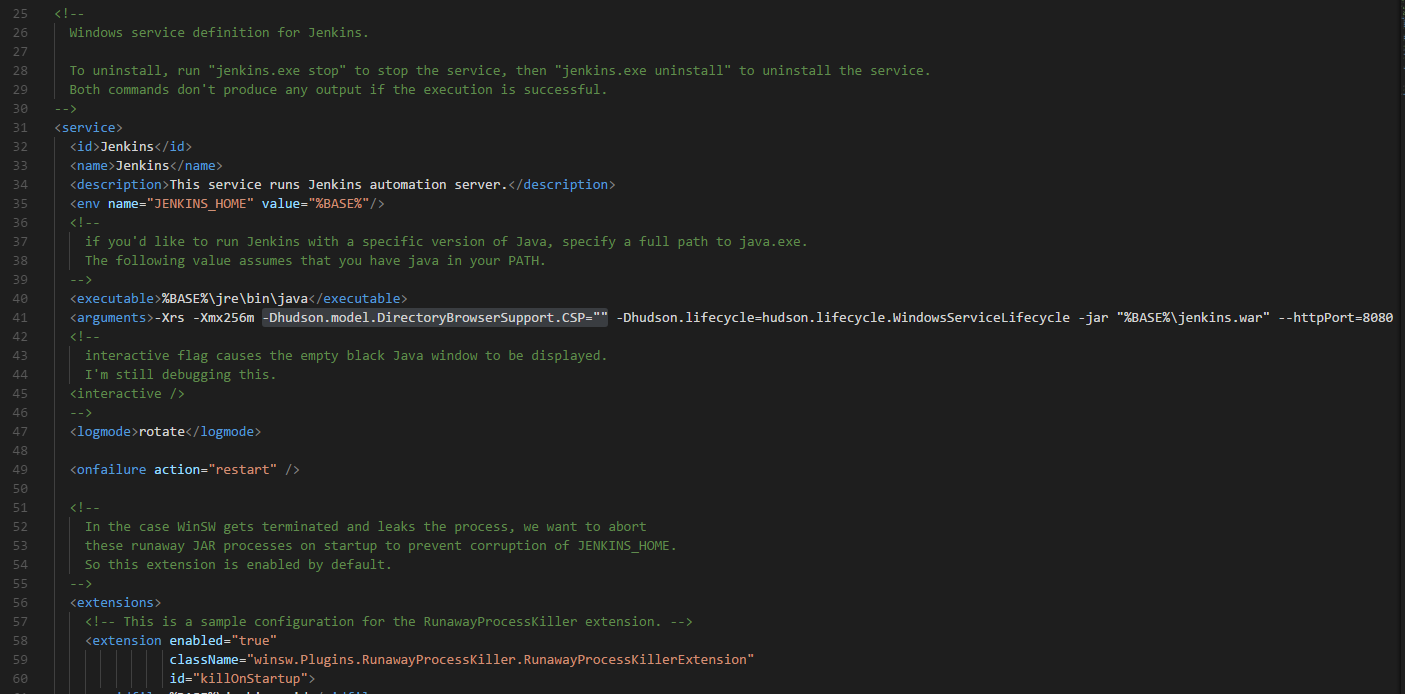
Download and install Jenkins <https://jenkins.io/download/thank-you-downloading-windows-installer-stable/>

Do not go to localhost:8080 yet.

### Start Jenkins (headless – not recommended)

In this mode, Jenkins won’t open browsers or any other windows on the host. It is faster, but AutoIt tests are not available (e.g. Outlook testing). Also, some features won’t work in headless browsers, so more stests will fail probably.

To run Jenkins in headless mode:

1. Follow the instructions on <http://localhost:8080/> after installation is complete.
2. Add -Dhudson.model.DirectoryBrowserSupport.CSP="" to arguments in *C:\Program Files (x86)\Jenkins\jenkins.xml*  
   

### Start Jenkins (non-headless - recommended)

In this mode, Jenkins will interact with the host OS the same as robot framework, opening windows, apps, browsers, etc.

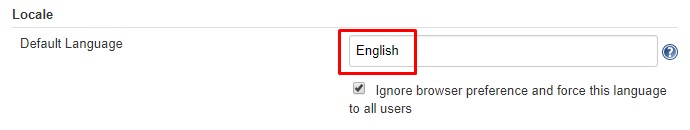
To run Jenkins in non-headless mode:

1. After installation, do not log in to Jenkins, close the browser if opened. (we won’t use the service any more)
2. Stop and disable Jenkins service in **services.msc**
3. Create a batch script containing these commands:  
   java -Dhudson.model.DirectoryBrowserSupport.CSP= -jar jenkins.war
4. Add the script to startup scripts [following this guide](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc770556%28v%3dws.10%29).
5. Restart the machine OR run the script manually to start Jenkins for the first time.

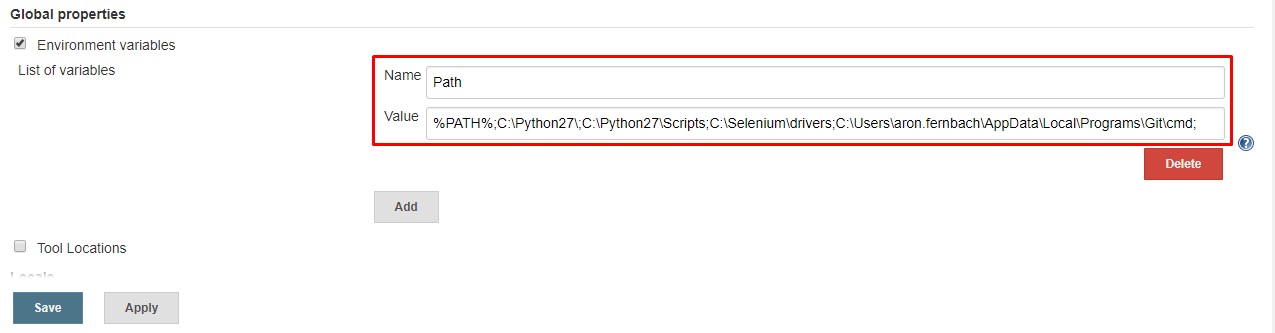
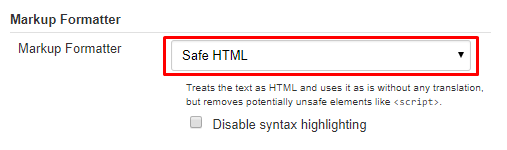
### First startup

1. Go to <http://localhost:8080/>, follow instructions
2. When asked, install recommended plugins

### Install plugins

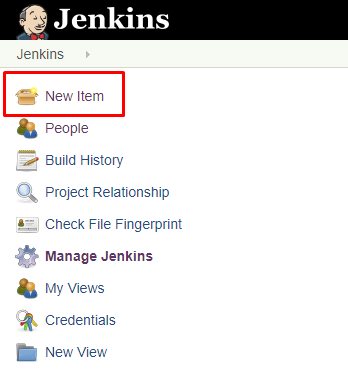
1. Install plugins at *Manage Jenkins -> Manage plugins  
   -* robotframework plugin  
   - locales plugin (optional)
2. (optional) set language to English in *Manage Jenkins -> System Configuration  
   *

### Configure system settings & variables

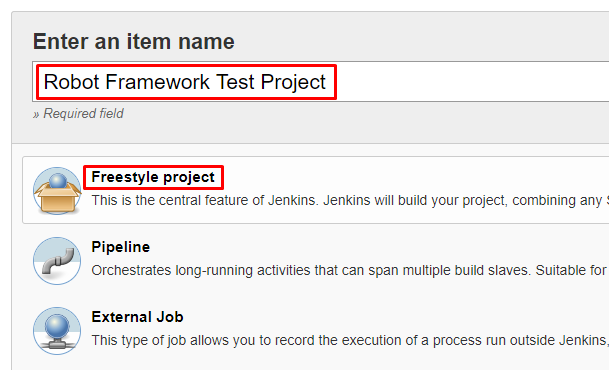
1. set path in *Manage Jenkins -> System Configuration* to:  
   ***%PATH%;C:\Python27\;C:\Python27\Scripts;C:\Selenium\drivers;C:\Users\USERNAME\AppData\Local\Programs\Git\cmd;***
2. Set safe html in *Manage Jenkins -> Global Security Options*, restart jenkins (localhost:8080/restart)  
   

## Set up Jenkins job

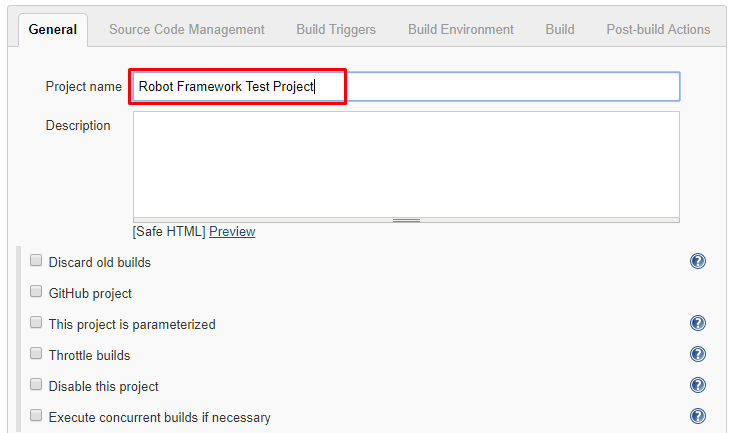
### Add new item



### Select Freestyle project



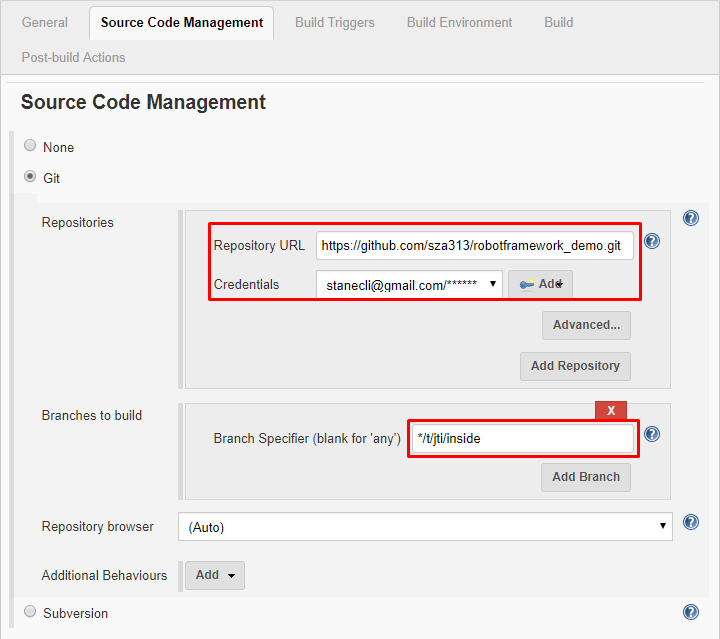
### Give the job a name



### Set up Git repository (or other SCM)

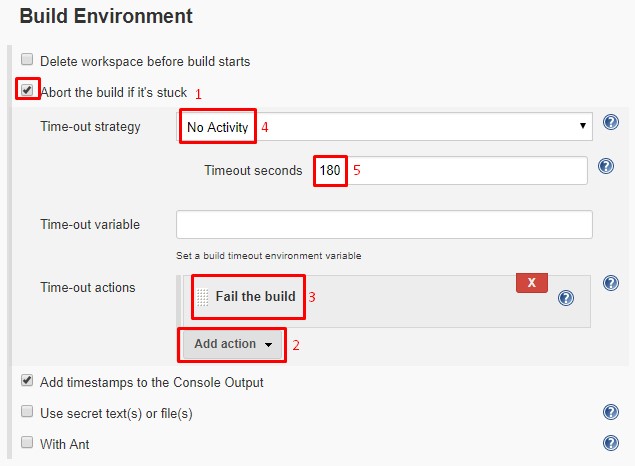
In Source code management tab, type in the repository name and credentials

Optionally, set branch and other options below.



### Timeouts

Because selenium tests can hang up unexpectedly, it is best practice to abort a build after a few minutes of idle time. Just remember not to set any timers in your tests that exceed this limit. 3 minutes should be OK for most test cases.



### Build command

On build tab we add the build command.

Click **Add build step**, then select **Execute Windows batch command.**

Add this command to the field: pybot --nostatusrc --outputdir [results dir] -C off -W 168 [robot resources]

Where:

* Results dir: robot framework will build the logs and reports in this folder (relative to Jenkins workspace dir. For windows its most likely *C:\Program Files (x86)\Jenkins\workpsace\[job name]*. Jenkins will copy these files to its web server so it can be accessed from the job build page.
* Robot resources: **relative path to the project root** where the tests suites exist OR individual test files separated with spaces.

See an example below:



### Setting up Robot Framework results

In post-build action tab we instruct the robotframework Jenkins plugin to grab our reports and make them available for view. It also generates charts which represent succeeded / failed tests.

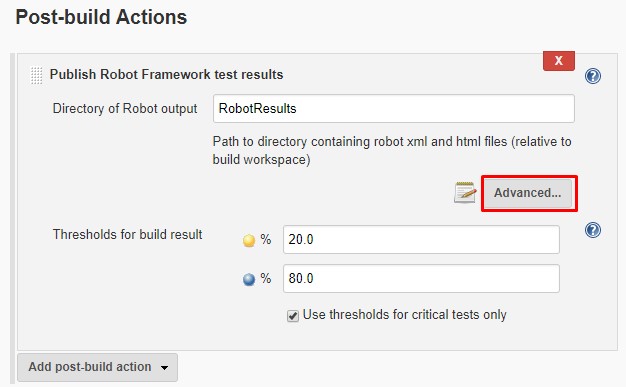
Click on **Add post-build action** and select **Publish robot framework test results**.

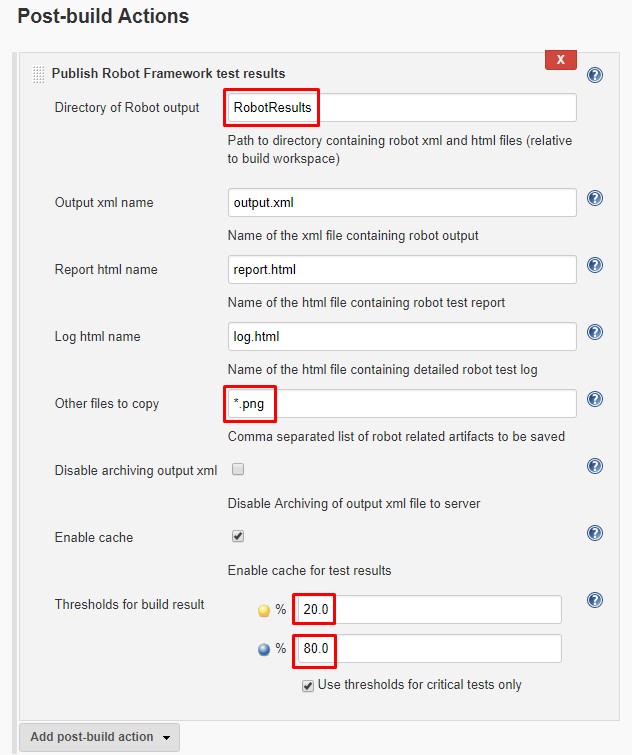
Click the advanced button on the right.

In the **Directory of Robot output** field, input the **outputdir** argument added in previous step.

Set thresholds if you like.

Important! Set \*.png in other files to copy to keep screenshots in the logs accessible.





Click save.

## Troubelshooting

### Build fails, no robot reports generated

Click “Console output” inside the current build, check what the console says

Common problems are:

* Console cannot find Selenium
* Console cannot find Git
* Console cannot find Pybot / Python

These are related to **Path.** Check if Path is correctly configured. See [installation instructions](#_Installation).

### Test pass count is above the pass threshold, but build still fails.

Check if --nostatusrc option is set in the shell script in job configuration. See the [Build command](#_Build_command) section.

### Error when opening log files

Check if Jenkins is configured properly to access html files. See **step 6 and 7** in [installation](#_Installation) section.

Also, check if shell results directory set in the shell command script is the same as in the Post-build actions -> publish robot framework test results -> Directory of Robot output field.

### Screenshots are not available in the logs.

Check if copying of images are set correctly in job configuration. See [Setting up Robot Framework results](#_Setting_up_Robot) section.