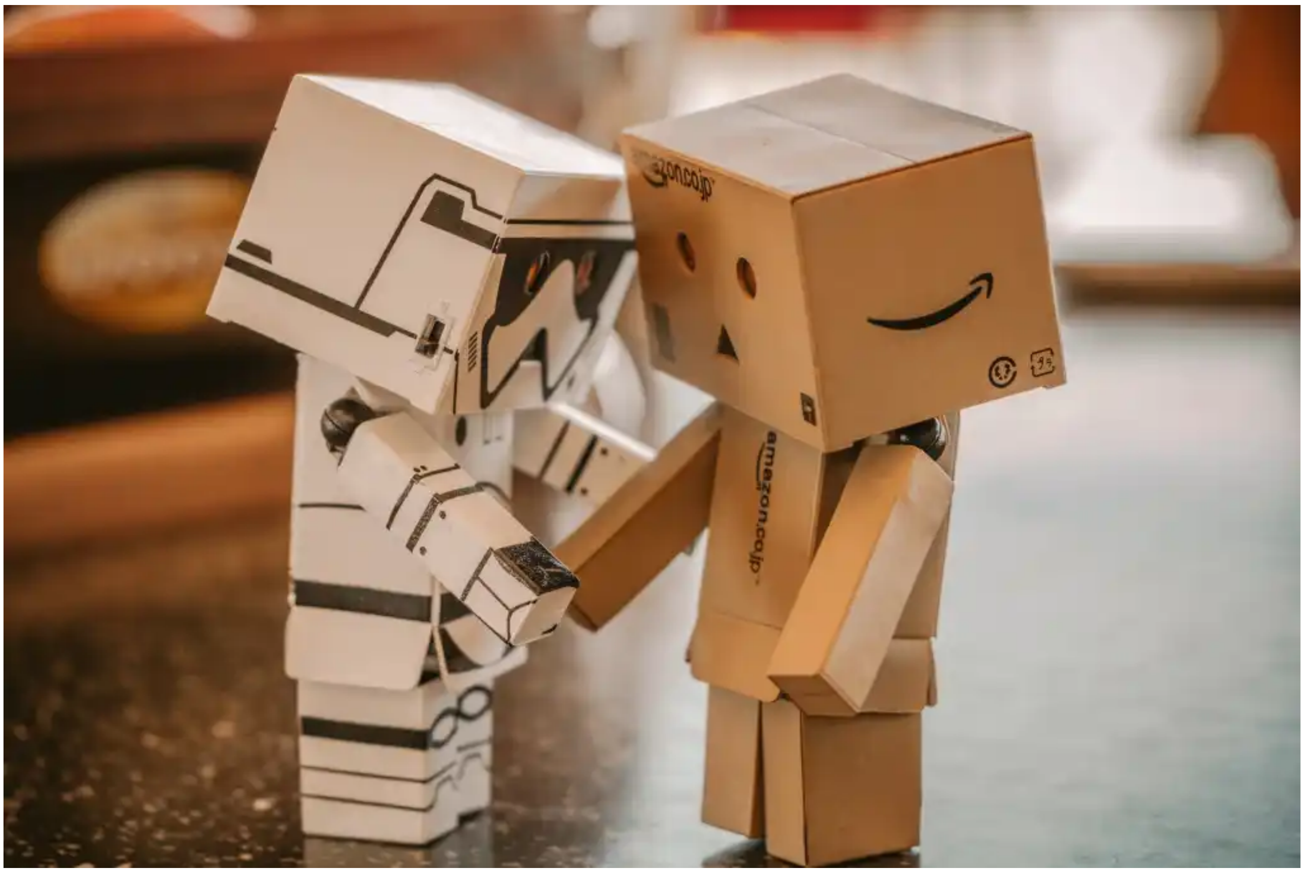
This blog is for me a platform to share my knowledge in the area of software development.

DEVOPS

Automated Acceptance Testing With Robot Framework



Date: May 19, 2020 Author: mydeveloperplanet 6 Comments

Robot Framework is an open source test automation framework. It is keyword driven and therefore very suitable to describe and automate acceptance tests. In this post, we will introduce Robot Framework and dive into the basic concepts.

1. Introduction

Robot Framework knows its origin at Nokia Networks, but has been open sourced since 2008. It is a framework for creating automated acceptance tests. Test scripts written with Robot Framework are keyword driven, making them very readable and understandable for everyone (assuming best practices are being used). Whether you are creating Test Cases based on Test Designs or based on User Story acceptance criteria, a Robot Framework test script will be your Test Case documentation and automated test script in one. Besides that, it is written in plain text and therefore very suitable for adding them to your version control system (like Subversion or Git). Get rid of all those tests described in Excel Workbooks! After running your test script, a test report is automatically generated, which gives you your test evidence.

The official Robot Framework website contains a lot of good information and an extensive User Guide. There is an active community and each year the RoboCon conference is being held.

Enough introduction, let’s see how this looks like and start exploring Robot Framework by using some examples. The sources used in this post can be found at GitHub.

2. Installation of Robot Framework

Robot Framework is implemented in Python, so you will need a Python interpreter on your system. We are using Ubuntu 18.04, the instructions for other platforms can be found at the Python website. We will use the Python 3 version which is available in our Ubuntu release. You can also use Python 2, but it is recommended not to use this Python version anymore because it has reached its end of life and Robot Framework support will also end for this Python version. Elaborate installation instructions can also be found here.

Verify the Python installation:

```
$ python3 --version
Python 3.6.9
```

Install pip in order to be able to install Python packages easily:

```
$ sudo apt install python3-pip
```

Verify the pip installation:

```
$ pip3 --version
pip 9.0.1 from /usr/lib/python3/dist-packages (python 3.6)
```

Install the Robot Framework library:

```
$ pip3 install robotframework
```

Verify the Robot Framework installation:

```
$ robot --version
Robot Framework 3.1.2 (Python 3.6.9 on linux)
```

We are now ready for creating our first Robot Framework script.

3. Application Under Test

Before getting started, we need an application which we want to test. We will make use of a small Python script which will allow us to add an employee to a CSV file, retrieve the list of employees from the CSV file and to delete the complete list. We are using a Python script, but it must be clear that any kind of application can be tested by means of Robot Framework. The `employee.py` script is the following:

```python
import csv
import os
import sys

EMPLOYEE_FILE = 'employees.csv'


def add_employee(first_name, last_name):
    with open(EMPLOYEE_FILE, 'a', newline='') as csv_file:
        writer = csv.writer(csv_file, delimiter=',', quotechar='"',
        writer.writerow([first_name, last_name])


def list_employees():
    employees_list = []
    if os.path.exists(EMPLOYEE_FILE):
        with open(EMPLOYEE_FILE, newline='') as csv_file:
            reader = csv.reader(csv_file, delimiter=',', quotechar=

            for row in reader:
                employees_list.append(' '.join(row))

    print(employees_list)


def remove_all_employees():
    if os.path.exists(EMPLOYEE_FILE):
        os.remove(EMPLOYEE_FILE)
    else:
        print("The file does not exist")


if __name__ == '__main__':
    actions = {'add_employee': add_employee,
               'list_employees': list_employees,
               'remove_all_employees': remove_all_employees}

    action = sys.argv[1]
    args = sys.argv[2:]
    actions[action](*args)
```

The application is for your information only and only for demonstration purposes. More important is how to use it. The next commands will add an employee `John Doe` and an employee `Monty Python`, the employees are retrieved, the employees are deleted (actually, the CSV file is deleted) and the employees are retrieved again, showing an empty list.

```
$ python3 employee.py add_employee John Doe
$ python3 employee.py add_employee Monty Python
$ python3 employee.py list_employees
['John Doe', 'Monty Python']
$ python3 employee.py remove_all_employees
$ python3 employee.py list_employees
[]
```

4. Explore the Basics

A good starting point for learning Robot Framework, is the documentation section of the Robot Framework website. In this paragraph, we will highlight some of the basics but when you want to explore more features, it is really recommended to read and learn how to use the official documentation. The user guide contains very good documentation for general usage of Robot Framework, the standard libraries which are available and some built-in tools.

4.1 Sections and Reporting

In our first test, we will verify the output when we retrieve an empty list of employees.

The test script contains several sections (there are more, but we will cover that later on):

- **Settings**: used for importing libraries and for adding metadata;
- **Variables**: used for defining variables which can be used elsewhere in the test script;
- **Test Cases**: the actual test cases to be executed. All test cases together are contained in a Test Suite.

A test script is formatted like a table. You can choose to use whitespace characters in order to separate the columns, but we prefer using a pipe. Pipes are more clear as a separator and when using whitespace, it is not always clear whether the separator is present and this can cause a lot of analysis time when a script returns errors.

Let's take a look at our first test script `employee.robot` (the script is located in a subdirectory *test*):

```
| *** Settings ***    |
| Documentation       | Test the employee Python script
| Library             | OperatingSystem

| *** Variables ***   |
| ${APPLICATION}      | python3 ../employee.py

| *** Test Cases ***                |                  |
| Empty Employees List              | [Documentation] | Verify the outp
| | ${rc}                           | ${output} =      | Run and Return
| | Should Be Equal As Integers     | ${rc}            | 0
| | Should Be Equal                 | ${output}        | []
```

The **Settings** section contains a documentation setting which describes the purpose of the test suite and we import the `OperatingSystem` library. We need this library in order to be able to invoke the Python script. A library contains several ready-made functions. In Robot Framework terms, they are called **keywords**.

The **Variables** section contains one variable `APPLICATION`, which contains the command we need to invoke for running the Python script.

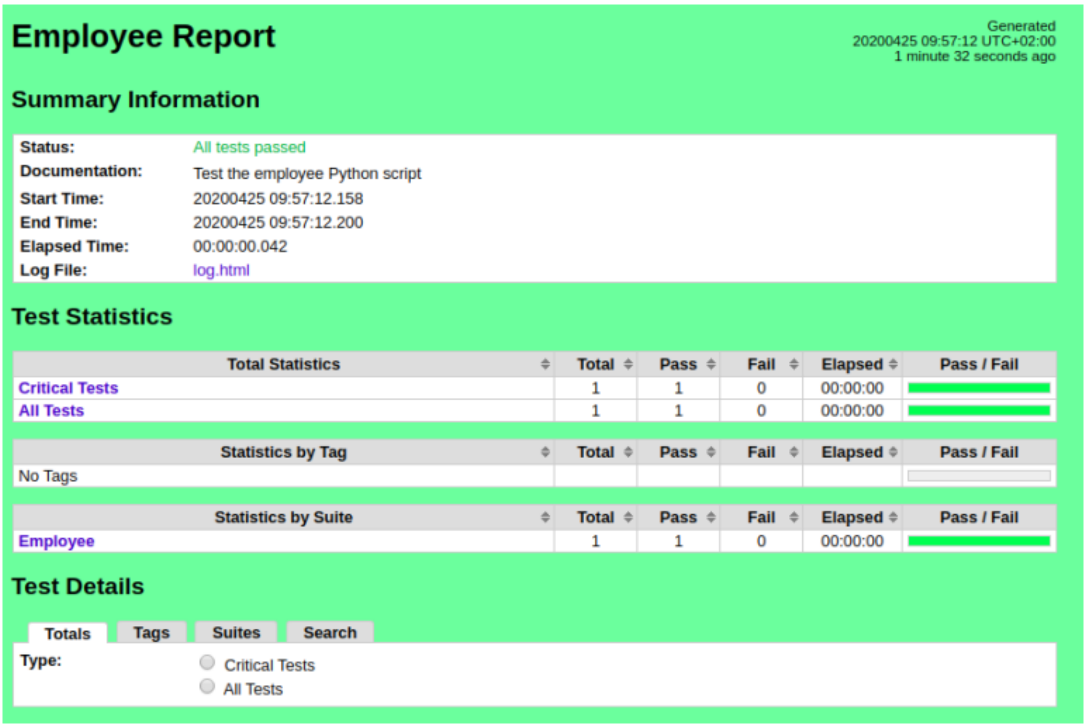
The **Test Cases** section contains one test `Empty Employees List`. It also contains a documentation tag which describes the purpose of the test. In the next line, we invoke the command for retrieving the list by using the keyword `Run and Return RC and Output` from the `OperatingSystem` library. We capture the output of the command and also its return code. With the built-in keywords `Should be Equal As Integers` and `Should Be Equal`, we can verify the return code and the output.

Run the test:

```
$ robot employee.robot
==============================================================
Employee :: Test the employee Python script
==============================================================
Empty Employees List :: Verify the output of an empty employees lis
--------------------------------------------------------------
Employee :: Test the employee Python script
1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
==============================================================
Output:  .../MyRobotFrameworkPlanet/test/output.xml
Log:     .../MyRobotFrameworkPlanet/test/log.html
Report:  .../MyRobotFrameworkPlanet/test/report.html
```

After running the test, we notice that the test case has passed and three files have been created. The `output.xml` file contains the results in XML format and can be used for processing the results into other applications. That is all you need to know for now about this output file.

The `report.html` provides us an overview of the test results. In case of one test it might seem a bit trivial, but this is really useful when your test suite contains a lot of tests.



Employee Report

Generated
20200425 09:57:12 UTC+02:00
1 minute 32 seconds ago

Summary Information

Status:	All tests passed
Documentation:	Test the employee Python script
Start Time:	20200425 09:57:12.158
End Time:	20200425 09:57:12.200
Elapsed Time:	00:00:00.042
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:00	🟩
All Tests	1	1	0	00:00:00	🟩

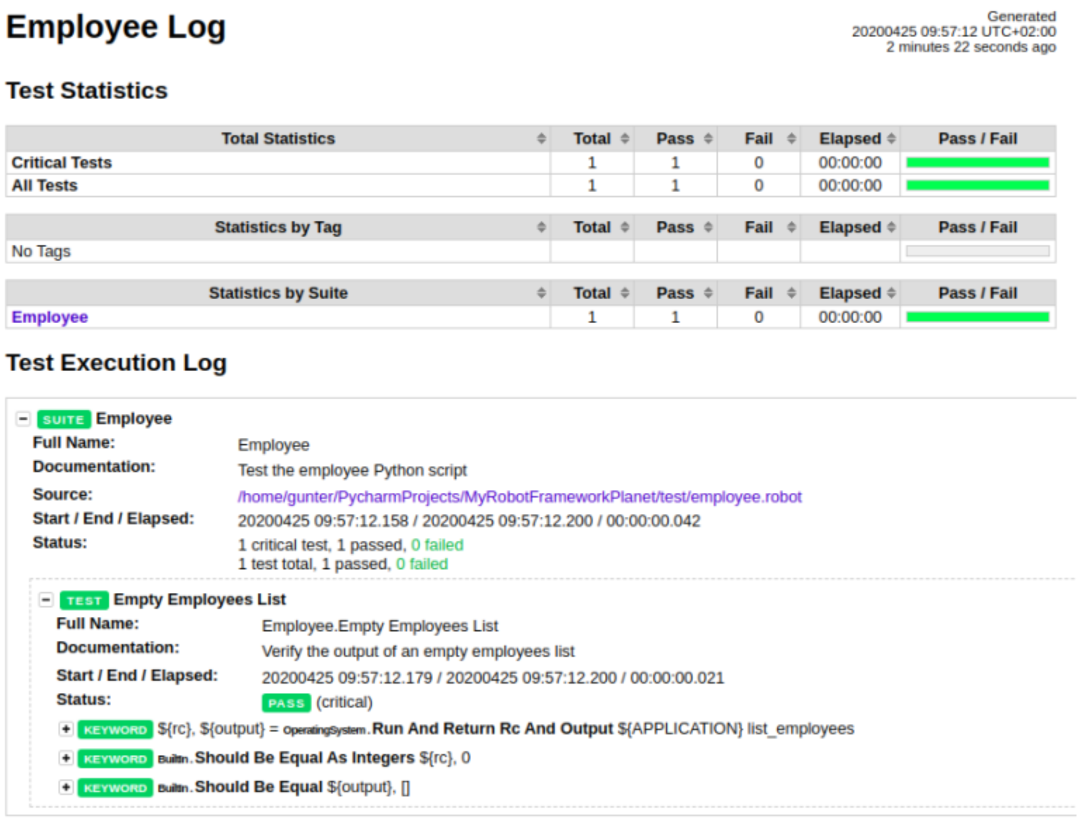
Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Employee	1	1	0	00:00:00	🟩

Test Details

Type: Critical Tests / All Tests

The most important file is the `log.html` file. This file contains the details of the test execution. Especially when something goes wrong during the test, this file contains the detailed information for analyzing what has gone wrong but it also serves as your test evidence.



Employee Log

Generated
20200425 09:57:12 UTC+02:00
2 minutes 22 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:00	🟩
All Tests	1	1	0	00:00:00	🟩

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Employee	1	1	0	00:00:00	🟩

Test Execution Log

```
SUITE Employee
  Full Name:            Employee
  Documentation:        Test the employee Python script
  Source:               /home/gunter/PycharmProjects/MyRobotFrameworkPlanet/test/employee.robot
  Start / End / Elapsed: 20200425 09:57:12.158 / 20200425 09:57:12.200 / 00:00:00.042
  Status:               1 critical test, 1 passed, 0 failed
                        1 test total, 1 passed, 0 failed

  TEST Empty Employees List
    Full Name:            Employee.Empty Employees List
    Documentation:        Verify the output of an empty employees list
    Start / End / Elapsed: 20200425 09:57:12.179 / 20200425 09:57:12.200 / 00:00:00.021
    Status:               PASS (critical)
    KEYWORD ${rc}, ${output} = OperatingSystem.Run And Return Rc And Output ${APPLICATION} list_employees
    KEYWORD BuiltIn.Should Be Equal As Integers ${rc}, 0
    KEYWORD BuiltIn.Should Be Equal ${output}, []
```

4.2 Setup and Keywords

Let's add another test and test the adding of an employee. We add the employee, verify the return code and then retrieve the list of employees to verify whether the employee is really added.

```
| Add Employee                     | [Documentation] | Verify adding an
| | ${rc}                          | ${output} =     | Run and Return R
| | Should Be Equal As Integers    | ${rc}           | 0
| | ${rc}                          | ${output} =     | Run and Return R
| | Should Be Equal As Integers    | ${rc}           | 0
| | Should Be Equal                | ${output}       | ['John Doe']
```

We run both tests again and both tests pass. But when we run the tests again, both tests fail.

```
$ robot employee.robot
==============================================================================
Employee :: Test the employee Python script
==============================================================================
Empty Employees List :: Verify the output of an empty employees lis
['John Doe'] != []
------------------------------------------------------------------------------
Add Employee :: Verify adding an employee
['John Doe', 'John Doe'] != ['John Doe']
------------------------------------------------------------------------------
Employee :: Test the employee Python script
2 critical tests, 0 passed, 2 failed
2 tests total, 0 passed, 2 failed
==============================================================================
```

We assume in our tests that the list of employees is empty from the start on. This is not true anymore once we have ran the tests because during the tests we add an employee to the list. We do get clear messages why our tests fail. When the console output is not clear enough, just check the `log.html` file where more detailed information can be found of what went wrong.

We can solve our failed tests by adding a `Setup` to the tests. In the setup we can add all the prerequisites for running our test. We need to empty the list of employees and we also have a command for doing so. This brings us to the following problem: the `Setup` can only contain 1 keyword and calling the command and verifying the return code requires two lines. We can solve this by creating our own keyword in the **Keywords** section.

```
| *** Keywords ***                 |
| Clear Employees List             | [Documentation] | Clears the list
| | ${rc}                          | ${output} =     | Run and Return R
| | Should Be Equal As Integers    | ${rc}           | 0
```

We add the `Setup` to both tests and both tests execute successfully when running more than once.

```
| Empty Employees List             | [Documentation] | Verify the outpu
| | [Setup]                        | Clear Employees List
...
| Add Employee                     | [Documentation] | Verify adding an
| | [Setup]                        | Clear Employees List
```

4.3 Readability

Our tests do what they have to do and they execute successfully but they are not very readable and they contain duplicate items. It would be better if we create keywords in order to improve readability and in order to have more maintainable tests.

First, we create a keyword for retrieving the list of employees. We need the output of the list in our test and therefore we `Return` the output value, just like you would do in any other programming language.

```
| Retrieve Employees List        | [Documentation] | Return the list
| | ${rc}                        | ${output} =     | Run and Return R
| | Should Be Equal As Integers  | ${rc}           | 0
| | [Return]                     | ${output}
```

The `Empty Employees List` test case is changed as follows:

```
| Empty Employees List           | [Documentation] | Verify the outpu
| | [Setup]                      | Clear Employees List
| | ${output} =                  | Retrieve Employees List
| | Should Be Equal              | ${output}       | []
```

This test case looks a lot better now. We have a clear distinction between the setup of the test case and the test itself and because keywords are used, the test itself is very readable.

Something similar can be done for adding an employee. The only difference here, is that we need to be able to pass `Arguments` to the keyword. It is possible to add arguments next to each other separated with pipes, but it is also allowed to place them on a new line. Be aware that you must use ellipsis when a new line is being used.

```
| Add Employee                   | [Documentation] | Add an employee
|                                | [Arguments]     | ${first_name}
|                                | ...             | ${last_name}
| | ${rc}                        | ${output} =     | Run and Return R
| | Should Be Equal As Integers  | ${rc}           | 0
```

The `Add Employee` test case is changed as follows:

```
| Add Employee                   | [Documentation] | Verify adding an
| | [Setup]                      | Clear Employees List
| | Add Employee                 | first_name=John | last_name=Doe
| | ${output} =                  | Retrieve Employees List
| | Should Be Equal              | ${output}       | ['John Doe']
| | [Teardown]                   | Clear Employees List
```

We also added a `Teardown` to this test, in the `Teardown` you bring the application back to its state before the test. It is often used to restore configuration settings to its original value when they have been changed during the `Setup` .

5. Conclusion

In this post, we gave an introduction to Robot Framework for creating automated acceptance tests. It is an easy to learn, stable framework and moreover, when the test cases are set up by using best practices, they are even readable to your business users. This last remark about using best practices is very important. If you do not follow best practices, the test cases will become unreadable and unmaintainable and you will end up with spaghetti code. See also How To Write Good Test Cases and Robot Framework Do’s and Don’ts.

ACCEPTANCE TEST PYTHON ROBOT FRAMEWORK TESTING TUTORIAL

Published by mydeveloperplanet

View all posts by mydeveloperplanet

6 thoughts on “Automated Acceptance Testing With Robot Framework”

Add Comment

1. Pingback: How to Write Data Driven Tests With Robot Framework – My Developer Planet
2. Pingback: Create Custom Robot Framework Libraries – My Developer Planet
3. Pingback: Create Custom Robot Framework Libraries - DevOpExpo
4. Pingback: Parallel Testing With Robot Framework – My Developer Planet
5. Pingback: Improve Your Robot Framework Tests With Robocop – My Developer Planet
6. Pingback: Automated Visual Testing With Robot Framework – My Developer Planet

This site uses Akismet to reduce spam. Learn how your comment data is processed.

© 2022

BLOG AT WORDPRESS.COM.