DESY Summer Student Programme: Scientific Computing at dCache

Andy Martinez Nieto, University Of Antwerp, Belgium

7 September 2016

Abstract

In this work we present an exploration of the Robot Framework with regards to dCache. The Robot Framework is a generic testing framework which advertises itself for its readable and easy to write software tests. We were able to successfully port the complete existing test suites for the web interface to dCache and for testing the interaction between various grid tools. It can be noted that writing tests in the Robot Framework is easy when there is a premade library available but writing a custom library can be time consuming so considerations have to be made wether it is worth the initial time investment. Overall we saw that tests were more readable and the output of the Robot Framework is logically structured without extra configuration and easily integrated into the Jenkins continuous integration platform used in the development of dCache.

Contents

1 Introduction

This work will present the use of the Robot Framework with regards to functional testing of dCache. Functional testing is the practice of testing user scenarios instead of small units of code. dCache is the software that is present on the storage elements which are responsible for storing and retrieving data. These concepts will be explained in more detail in the coming sections. The Robot Framework is a general testing framework that produces readable and easy to write tests. This framework will be explained in more detail in the appendix where you can find a short guide on the Robot Framework which was written for the dCache developers to be used internally.

The goal of this internship was to obtain real life experience in working in a scientific computing group and to gain more familiarity with the inner workings of the tools that are used during big data analysis (for example data from LHC). This was also an oppurtunity for the dCache group to let someone explore a new framework without losing valuable time of one of the main developers. This allowed them to get an overview about what the framework could bring to the table and be able to decide if they want to continue working with it or not.

During this time at dCache a presentation was also given to the developers about the inner workings of the Robot Framework and a concise guide was written about working with the Robot Framework which got distributed to the developers. This guide will be attached in the appendix.

In this report we will briefly introduce the aforementioned concepts to be able to explain what was done in more precise terms. An overview of the projects which were completed and what can be improved is given and also a short discussion is had about how the Robot Framework fits into the dCache development cycle.

1.1 dCache

To explain what dCache is we will first explain the tiered system of the LHC computing grid [1] [2] [3].

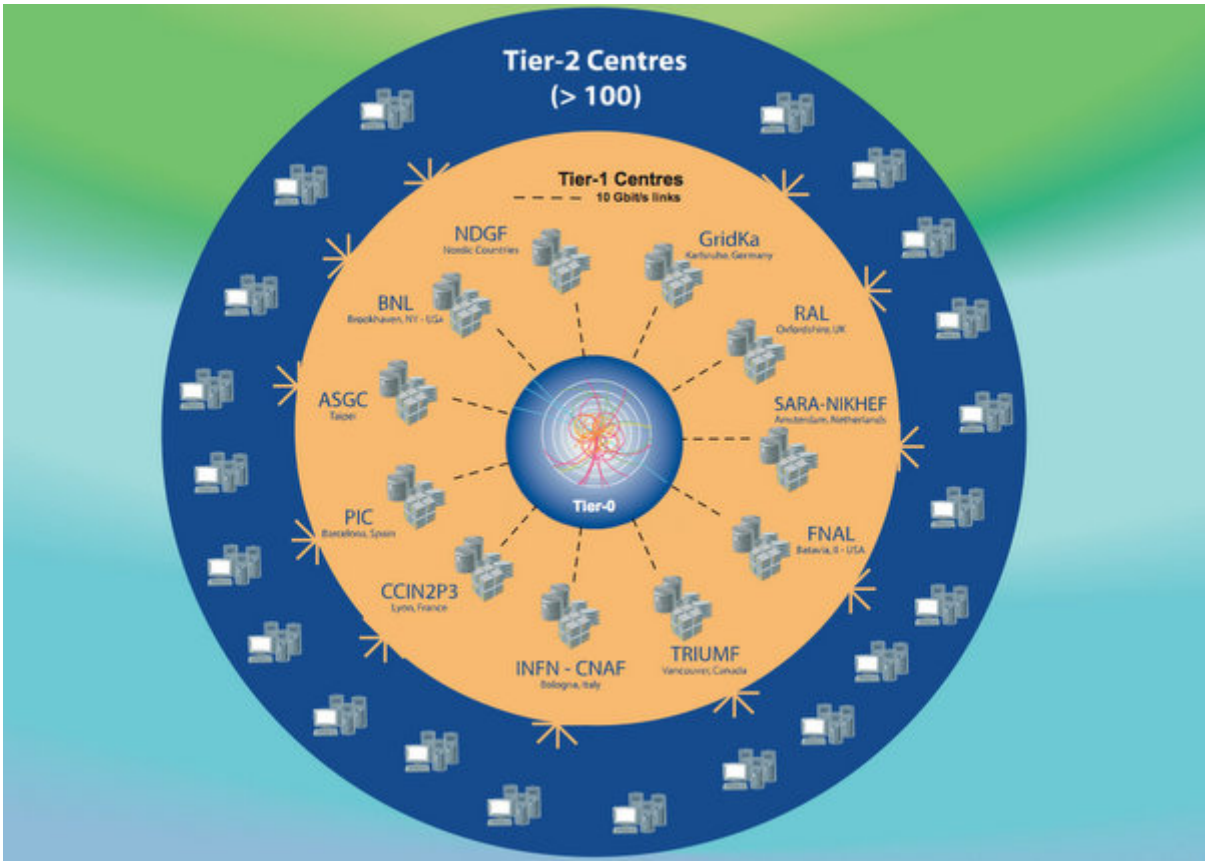

Figure 1: Scintific computing grid tiers

This is the computing grid of the LHC. There are three tiers and they all have a different purpose which we will briefly explain now.

Tier 0 is the experiment level. This is where all the raw data gets produced. CMS alone produces about 1 PB/s of raw data which is impossible to store. At this level there are hardware triggers that check if an event is worth keeping or not. These triggers will reduce the data flux to about 100 MB/s. This raw data gets stored on tape at the Tier 0 site. Besides storing the data a preliminary reconstruction of events will be made. The reconstructed events and the raw data will be distributed to Tier 1 sites.

On Tier 1 the received raw and reconstructed data gets stored on disk and tape. Besides storage there will also be analysis, calibration, re-reconstruction and skim making at this tier. Skims are a collection of reconstructed datasets according to some specifications. These skims are sent to Tier 2 sites.

Tier 2 sites will store the skims on tape or disks and create Monte-Carlo simulations. This is also the center where individual physicists can retrieve data from for analysis at their university and thus where discoveries are made.

This very short overview of the tiering system of the scientific computing grid shows that every tier will store and send data in some form. This is where dCache comes into play. dCache is the software that runs on the storage elements which handles storing data to disk or tape as well as retrieving this data by using various different protocols and clients [4]. dCache is used on Tier 1 and Tier 2 storage elements. The dCache software needs to be able to efficiently store and retrieve large amounts of data from across different nodes.

The way we interact with a storage element with dCache on (this isn't only for a dCache system) is by using so called grid tools. These are clients written specifically to do grid based actions, for example:

the client srmcp is a client that is designed to be able to copy a file (the "cp" part) using the srm protocol. There are various other grid tools that are used to interact with dCache and these are only mentioned here now because one of the projects was about testing the interaction between these grid tools and a dCache instance.

1.2 Robot Framework

The Robot Framework is a generic test automation framework. This means that the framework should be able to be used to write any kind of test for any kind of software. The main reason why we wanted to explore this framework and not some other testing framework is because tests in the Robot Framework are supposed to be easy to read and write. This will mean that developers won't have to spend a lot of time developing tests for their code which means that there will be more tests written.

A more extensive guide on using the Robot Framework was also written and is included in the appendix so as to not clutter the report.

1.3 Unit vs Functional Testing

There are a lot of different kinds of tests in software engineering but we will focus on two: unit and functional. Unit tests test a single piece of the code, for example: There is a function that adds two numbers together then unit tests would be to throw all kinds of arguments into this one function and checking if the results make sense.

Functional tests test a user scenario. If there is for example a website then a user test might be: Open the browser to that website, click the login button, fill in my credentials, click the login button, check that we are logged in, click the logout button, check that we are logged out and close the browser. It is clear that these kinds of tests use a lot of different pieces of the code. You are testing if pieces of code which work independently also work correctly together.

1.4 Continuous Integration

Continuous integration is a concept in software engineering which is the practice of adding additions to the source code to the main repository on a regular basis. This is usually combined with automatic build processes.

dCache uses Jenkins for their continuous integration platform. The continuous integration system of dCache development goes according to the following steps:

1. Someone commits changes to the dCache source code to one of the dCache version branches on github.
2. This will trigger dCache (one of the versions) to build.
3. Unit tests are run.
4. dCache is deployed on a machine.
5. Functional tests are run against that dCache machine.

By rebuilding both the test machine as well as dCache every time a change to the source code happens we can assure that dCache didn't break by running the tests against a fresh build every single time.

The Robot Framework can be easily integrated into Jenkins by use of a plugin. This will make it so that the output files of a Robot Framework test will be read by Jenkins and a graph of the successfull and failed tests will be shown on the project page on Jenkins.

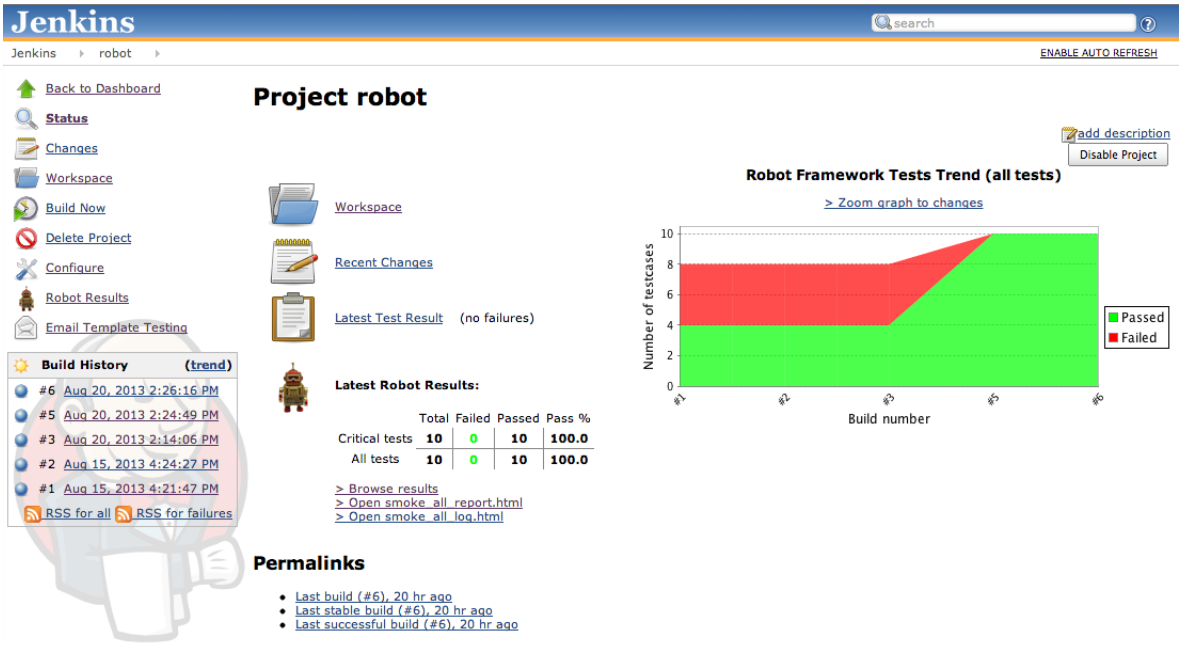

Figure 2: Jenkins project page with Robot Framework integration

2 First Project: Webadmin Tests

The first project was porting existing website tests written in Java to the Robot Framework. The website that we're testing is a web interface to dCache.

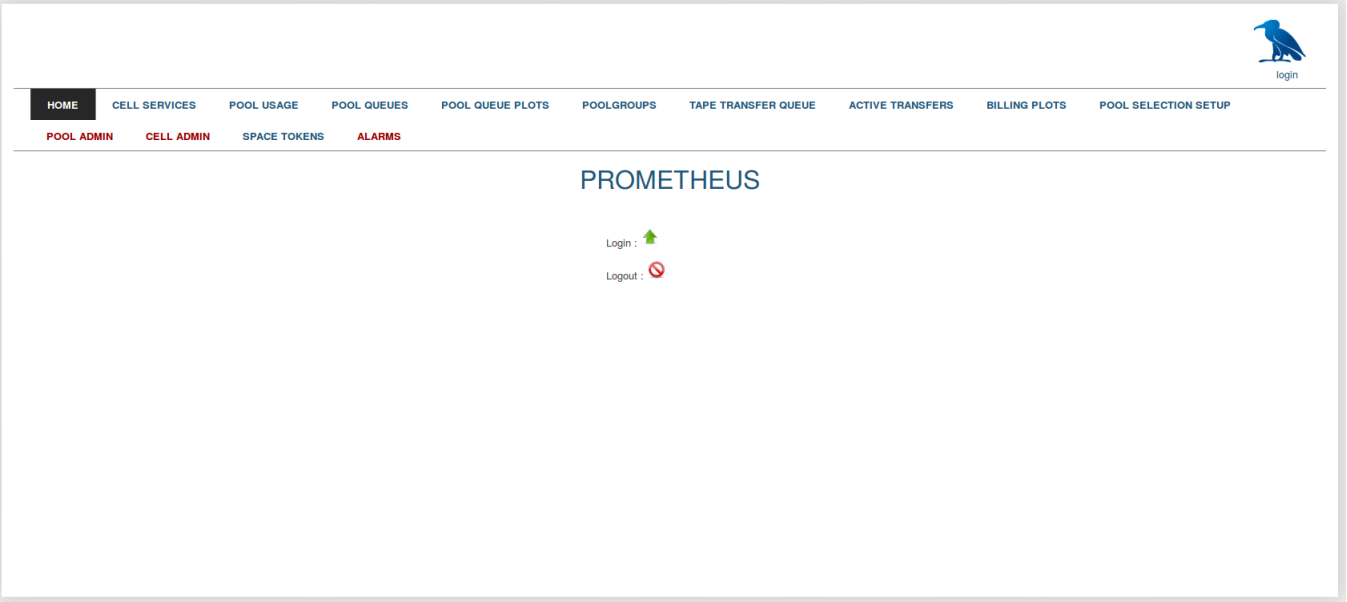

Figure 3: Web interface of dCache

The tests are separated by website page, for example we have a test suite with tests only for the Cell Admin page (which is a separate page on the web interface). Besides a pagewise separation of the test suites there are also a couple of general test suites: the ones concerning logging in and navigating to the

various pages. The login test suite will test if it can login via various different ways (different buttons that redirect to the login page) and if all the various extra buttons on the login form behave as they should (for example the reset button which should clear the username and password fields).

The Selenium2 Library was used to write these tests. This library gives us keywords like "OPEN BROWSER" and "CLICK ELEMENT". They are easy to understand keywords which make it easy to write browser based tests. Selenium is a libarary written in Java for the purpose of interacting with a browser in a programmatic manner (There are also Python bindings for Selenium).

One of the tests is shown to give an example of the readability of the Robot Framework with the Selenium2 Library:

```
22  LOGIN VIA USER ACTION
23      [Documentation]     Login using the login button in the user panel.
24      OPEN BROWSER    %{BASE_URL} %{BROWSER}
25      CLICK ELEMENT   userpannel.action
26      TITLE SHOULD BE     Login
27      SUBMIT CREDENTIALS  ${ADMIN_USERNAME}   ${ADMIN_PASSWORD}
28      CLICK BUTTON    login.submit
29      ELEMENT TEXT SHOULD BE      ${USERNAME_ON_SITE} ${ADMIN_USERNAME}
30      CLOSE BROWSER
```

Figure 4: Login test written in the Robot Framework

The test is called "LOGIN VIA USER ACTION" and every line below this is a step in the test. As we can see, there is almost a one-to-one connection between explaining the steps of the test in english and writing the actual test. This is the biggest draw to the Robot Framework. Someone not familiar with programming can still read this and understand what is happening with minimal effort.

All the tests written in Java were successfully ported to the Robot Framework and can now be used in production instead of the Java tests.

All the code can be found at the robot-webadmin-tests project on my github[5].

3 Second Project: Grid Tools Functional Tests

The second project was porting all of the grid tools tests to the Robot Framework. The big difference between this project and the previous one is that there is no library already made to write these tests with.

The first step in completing this project was creating a library with which we could write the required tests in the Robot Framework efficiently and in a readable style. As mentioned in detail in the short guide that was written, and which can be found in the appendices, writing a library for the Robot Framework is the same as writing a class in Java or Python. Every public member function of the class is a callable keyword in the Robot Framework file.

Writing the library took most of the time during this project because a balance had to be found between having readable tests and having control over the tests. If too much customizability is abstracted away from the keywords to make it more readable it may reduce the ability to write different tests quickly while on the other hand if nothing is abstracted away the tests had to be written in the Robot Framework language and they won't be as readable.

Because we were porting tests from Java to the Robot Framework we already knew what the library should be capable of. This made it so that writing the library was easier than if we had to write a general library without knowing what kinds of tests were needed.

After about a week or two the library was done and tests were able to be written fairly quickly, one of the tests will be shown next:

```
56  COPY BAD CHECKSUM
57      [Documentation] Copies a file from which the content (and thus checksum) changes d
58      SET CLIENT  ${CLIENT}
59      SET PROTOCOL    ${PROTOCOL} ${PORT}
60      SET HOST    ${HOST}
61      SET EXTRA ARGUMENTS -${SRM_VERSION} -retry_num=0
62      ${FILE_NAME}=   REPLACE STRING  ${TEST_NAME}    ${SPACE}    ${EMPTY}
63      COPY LOCAL FILE     /proc/uptime    ${REMOTE_FILE}${FILE_NAME}
64      ERROR SHOULD CONTAIN    Checksum mismatch
```

Figure 5: Grid Tools Test

In this test a file is sent of which the checksum changes while sending it. This should result in an error. As we can see, the test is still readable but because of the different "SET" keywords it is clearly different than the web interface test that was shown. Because the tests will be written by people who have no problem with programming we chose to give up some readability for more controlability over the tests.

The libraries that were written for these tests are fully documented and can do all the current grid tools tests which means that also these test suites can be used in development instead of the current Java ones.

All the code for this project can be found at the Grid Tools Functional Tests project on my github[6].

4 Robot Framework vs Plain Python or Java

The obvious question that is asked when introducing a new framework is: Why should we use a new framework for testing when we can write these tests in plain Python or Java? We will show two tests that do the same thing but one is written with the Robot Framework and one in Python:

```
1   *** Settings ***
2    Variables  auth_vars.py
3    Library  CrlCpLib.py    ${USERNAME}     ${PASSWORD}
4   Suite Setup     CHECK CREDENTIALS   ${BASE_URL}         # Checks if we ca
5
6   *** Variables ***
7    ${BASE_URL}   https://prometheus.desy.de/Users/${USERNAME}/Private/
8
9    ${FILE_DIR}     /home/andy/
10  ${FILE_NAME}    CMSCalorimeter.png
11
12
13  *** Test Cases ***
14  FILE COPY
15      COPY FILE TO    ${FILE_DIR}${FILE_NAME}   ${BASE_URL}${FILE_NAME}
16      STATUS CODE SHOULD BE   201
```

Figure 6: Copy file test in the Robot Framework

```
5    prometheus_url = "https://prometheus.desy.de/Users/" + USERNAME + "/Private/"
6    curl_boy = CrlCpLib(USERNAME, PASSWORD)
7    curl_boy.check_credentials(prometheus_url)
8
9    # File copy test
10   curl_boy.copy_file_to("/home/andy/CMSCalorimeter.png", "CMSCalorimeter.png")
11   curl_boy.status_code_should_be(201)
```

Figure 7: Copy file test in Python

Both implementations use the exact same library so there are no differences there.

First of all, in the Python implementation, we have to deal with class instances and manage them ourselves. We have to make sure that different test cases do not influence each other. The Robot Framework will keep class instances separate per test case by default.

When using the Robot Framework you are forced to follow a specific structure for the test files. Every section of the Robot Framework file has to be denoted by it's specific table title (Settings, Variables, Test Cases, Keywords). In Python the programmer has to be trusted to create structure using comments and documentation. This can make it so that other programmers will write tests that have a different structure. Robot Framework ensures that all tests are written with the same structure which makes working with various different Robot Framework files, in general, easier than working with different Python files.

In the Robot Framework libraries written in Java and libraries written in Python can be used in exactly the same way. This has the benefit that people not well versed in one of the languages can still use all libraries to write their tests without having to deal with the syntax of the language that the library is written in. By using the Robot Framework you ensure that all tests are written in the same "language" (the Robot Framework language) and thus be readable and appendable by everyone regardless of their Python or Java skills.

A more general remark about using a separate framework for testing is that a nicely structured output is obtained without having to do anything extra. It is possible to create a custom output in Python or Java but then again different structures across different test suites can be the result which is not optimal for a professional software development environment. For the Robot Framework specifically we have the benefit that the output can be used by Jenkins to add the results to the project page which is something that isn't straightforward when making a custom output file.

In short, having a separate framework for testing ensures that all tests are structured in the same way, written in the same style and output the results in the same detailed structure. The Robot Framework has the added benefits that the output can be added into Jenkins projects.

5 Conclusion: Thoughts on the Robot Framework

During the Summer Student Programme we worked in the dCache group and had to explore the Robot Framework which is a generic testing framework. The dCache team wanted to know if it was applicable to their software and if it is a better alternative than their current testing method.

By porting the existing tests we have showed that tests can be readable and easy to write which was the main goal. The Robot Framework is also easily integrated in the current continuous integration platform of the dCache software.

Over the course of the programme we have encountered some positives and negatives about the Robot Framework which we will list and explain.

The positives are listed as follows:

- It is very easy to setup and start working with. The only requirement is python 2.6 or higher and then the command: pip install robotframework.
- When there are premade libraries tests are written easily.
- Making a custom library is as simple as writing a Java or Python class.
- Java and Python are both supported.
- Python scripts or Java programs can be used to create a variable with complex logic. This variable can be called inside the Robot Framework file with it's variable name.
- The output is detailed and logically structured.
- The Robot Framework comes packaged with various tools to help with documentation and combining of outputs which is very valuable.
- Easily integrated into Jenkins (continous integration platform).

The negatives are listed as follow:

- If there is no premade library it can take a lot of time to create a general custom library.
- If there is no need to mass produce tests for something that doesn't have a premade library, it might be more time efficient to not work in the Robot Framework.

Formally, the Robot Framework complies with the dCache internal requirements on the following points:

- The tests are easy to read/write.
- The framework can be integrated into the used continuous integration platform (Jenkins).
- The framework is OS independent.
- Test suites are easily integrated into source control (Test suites are just files and directories).
- When errors happen the output of the Robot Framework points at the specific part in the code that failed.

A Robot Framework Guide

A.1 Introduction

Robot Framework [7] is a generic test automation framework for acceptance testing and acceptance test driven development. Acceptance testing is the practice of conducting tests to determine if the requirements of a specification or contract are met. Tests are easy to read and write in this framework. The files are written in a tabular form and are split up in sections prefaced by a table header. In this short guide we will go over the structure of a Robot Framework file, explain the different keywords, analyze the output files and show what you need to do to start writing tests for your own software.

A.2 Structure of a Robot Framework file

A Robot Framework file looks like this:

```
1  *** Settings ***
2  Resource    Keywords.robot
3  Variables   auth_vars.py
4  Resource    Variables.robot
5  Library     Collections
6  Library     Selenium2Library
7  Suite Setup     WEBPAGE SHOULD BE REACHABLE     ${BASE_URL}     ${BASE_TITLE}   ${BROWSER}
8  Suite Teardown  CLOSE ALL BROWSERS
9
10 *** Variables ***
11 ${USERNAME_ON_SITE} xpath=//span[@id='userpannel.username']
12
13 *** Test Cases ***
14 LOGIN VIA SYMBOL
15     [Documentation]     Login using the symbol on the home page.
16     OPEN BROWSER    ${BASE_URL} ${BROWSER}
17     CLICK ELEMENT   home.login
18     TITLE SHOULD BE     Login
19     SUBMIT CREDENTIALS  ${ADMIN_USERNAME}   ${ADMIN_PASSWORD}
20     CLICK BUTTON    login.submit
21     ELEMENT TEXT SHOULD BE      ${USERNAME_ON_SITE} ${ADMIN_USERNAME}
22     CLOSE BROWSER
23
24
25
26 *** Keywords ***
27 WEBPAGE SHOULD BE REACHABLE
28     [Arguments]         ${URL}      ${EXPECTED TITLE}   ${BROWSER}
29     OPEN BROWSER        ${URL}      ${BROWSER}
30     TITLE SHOULD BE     ${EXPECTED TITLE}
31     CLOSE BROWSER
32
```

Figure 8: Example of a Robot Framework file

This is an example of a login test that was written for the dCache website. This example uses Selenium to test the dCache website so if you see words like home.login, login.submit or Login these are just css elements of the webpage and aren't important for the discussion of the Robot Framework. Currently we are only interested in how the Robot Framework file is built up and how it works, the exact details of this specific example aren't important.

Test data is structured in four types of tables: Settings, Variables, Test Cases and Keywords. These

are further explained in the test data tables section of the Robot Framework documentation. These are mostly surrounded by asterisks and are called table headers:

```
*** Variables ***
```

Figure 9: Table header

This denotes a section in the Robot Framework file. Everything in a test data table will have the properties of that table header. All of the table headers will be explained in separate sections because they all have very distinct features.

A.2.1 Settings

```
1  *** Settings ***
2  Resource    Keywords.robot
3  Variables   auth_vars.py
4  Resource    Variables.robot
5  Library     Collections
6  Library     Selenium2Library
7  Suite Setup     WEBPAGE SHOULD BE REACHABLE     %{BASE_URL}     %{BASE_TITLE}    %{BROWSER}
8  Suite Teardown  CLOSE ALL BROWSERS
```

Figure 10: Settings section of the example Robot Framework file

This example of a settings section in a Robot Framework file already contains a lot of different things we can do in this section. For a more complete set of settings configurations please refer to the settings section of the Robot Framework Docs.

When looking at the Settings in the example we see the tabular format pretty clearly. In the first column we will usually find a reserved word with it's arguments in the columns next to it. In these settings we find: Resource, Variables, Library and then some keywords (which are different from reserved words) like: Suite Setup and Suite Teardown which we will discuss later.

Library The Library setting is the usual import statement that is present in some form or another in standard programming languages. This gives you the ability to import functions or keywords as they are called in Robot Framework. We will have a more extensive discussion about these keywords later.

There are three types of libraries: *Standard*, *External* and $UserMade$. The *standard* libraries come with the Robot Framework installation and consist of the following libraries:

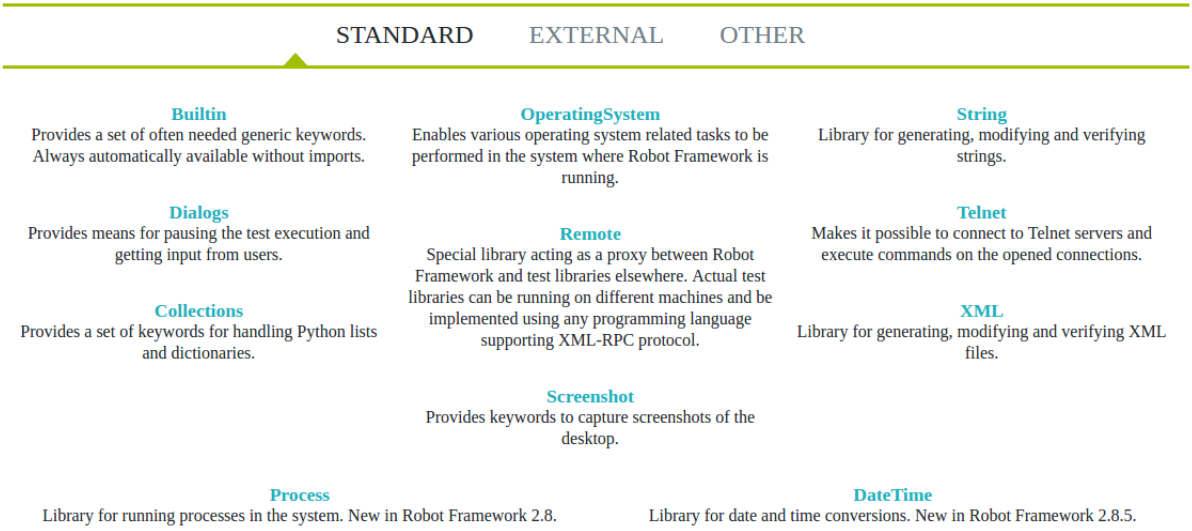

Figure 11: *Standard* libraries in the Robot Framework

These libraries contain keywords that are used in almost every Robot Framework file. Usually they are not enough to test your software on their own but they provide useful general keywords.

External libraries are libraries that you have to install next to the Robot Framework and which are/were developed by others and are big and well maintained enough to be considered the standard set of keywords for the particular use case that the libraries handle.

Figure 12: *External* libraries in the Robot Framework

These *external* libraries contain keywords for http requests (based on the python Requests library), Selenium2, SSH, FTP, Django etc. Before implementing your own keywords (which we will explain how to do later) always check if there isn't already an *external* library.

Libraries can be found at `http://robotframework.org/#test-libraries`.

The last kind of library are the *usermade* libraries. If there is no library for your specific usecase you can always make your own library and import it with the same statement as the others.

Variables Instead of creating variables inside the Robot Framework file, another way (and most of the time this is the preferred way) is to create a Python module which contains variables. This gives you the advantage that you can call any Python function while creating your variables or that you can remove the data from any Robot Framework file for flexibility or safety. The variables are called in the Robot Framework file by their variable name in the Python module. For example in our auth_vars.py file we have the following variables:

```
3    ADMIN_USERNAME = "GuideUsername"
4    ADMIN_PASSWORD = "1234"
```

Figure 13: auth_vars.py example

and they can get called in the Robot Framework file like:

```
SUBMIT CREDENTIALS  ${ADMIN_USERNAME}    ${ADMIN_PASSWORD}
```

Figure 14: Call of auth_vars.py variables in Robot Framework file

Another common way to extract variables from a Python module is to have functions in that module that take arguments and return variables. You can also implement these variables in a Python or Java class.

If you don't want to work with Python or Java to implement variables you can also use YAML files.

For more information on Variable files please see the Variable File section in the Robot Framework Docs.

Resource Resource files resemble the variable files but these files are Robot Framework files. They have the exact same structure as any other Robot Framework file but do not contain Test Cases. These files are great to keep user defined keywords in so that they don't clutter up the actual testing Robot Famework files.

An example of such a resource file is the following:

```
*** Keywords ***
WEBPAGE SHOULD BE REACHABLE
    [Arguments]          ${URL}      ${EXPECTED TITLE}   ${BROWSER}
        OPEN BROWSER         ${URL}      ${BROWSER}
        TITLE SHOULD BE      ${EXPECTED TITLE}
        CLOSE BROWSER

ABLE TO LOGIN TO PAGE
    [Arguments] ${PAGE_NAME}    ${USERNAME} ${PASSWORD}
    OPEN BROWSER    %{BASE_URL} %{BROWSER}
    TITLE SHOULD BE     %{BASE_TITLE}
    LOGIN TO PAGE   ${PAGE_NAME}    ${USERNAME} ${PASSWORD}
    CLOSE BROWSER


LOGIN TO PAGE
    [Arguments] ${PAGE_NAME}    ${USERNAME} ${PASSWORD}
    ${ELEM_ID}= GET FROM DICTIONARY ${PageName_ElemID_dict} ${PAGE_NAME}
    ${WE_ARE_LOGGED_OUT}=   RUN KEYWORD AND RETURN STATUS   ELEMENT SHOULD NOT BE VISIBLE
    RUN KEYWORD IF  ${WE_ARE_LOGGED_OUT}    LOGIN   ${USERNAME} ${PASSWORD}
    CLICK ELEMENT   ${ELEM_ID}
    ${EXPECTED_TITLE}=  GET FROM DICTIONARY ${ElemID_Title_admin}   ${ELEM_ID}
    TITLE SHOULD BE     ${EXPECTED_TITLE}
    ELEMENT TEXT SHOULD BE      ${USERNAME_ON_SITE} ${USERNAME}

```

Figure 15: Example of a Resource file

As you can see there is only a Keywords section in this Resource file but it can have a Settings and Variables section as well. You can think of these kinds of files as Python modules that contains all the functions that you want to use in the main Python file but don't want to keep there.

For more information on Resource files please see the Resource File section in the Robot Framework Docs

Settings Defined Keywords In the settings section we can also implement some other useful keywords like: Suite Setup, Suite Teardown, Test Setup, Test Template, Test Teardown. For the full list of settings defined keywords check the Robot Framework Settings.

We will discuss Setup, Teardown and Template type keywords here. In the example shown above we have the Suite Setup and Suite Teardown in our Robot Framework file. A Suite is basically the Robot Framework file, it's the collection of tests in one Robot Framework file.

The Suite Setup keyword defines the keyword which gets executed at the very beginning of a Suite. In our case it's the "WEBPAGE SHOULD BE REACHABLE" keyword. This is a prerequisite test for the entire suite which means if this keyword returns with a fail none of the tests in the Suite will get executed. In our case it doesn't make sense to try and login to the site when we can't even reach the site.

The Suite Teardown keyword defines the keyword that gets executed at the very end of a Suite. This is usually the cleanup keyword. If we have a cleanup keyword in every test case then there is a chance that the cleanup doesn't get executed when something before that fails in the test. The Suite Teardown keyword always gets executed at the end of a suite regardless of how the test cases went.

The Suite Setup and Suite Teardown keywords are also available on the Test Case level which are

aptly named Test Setup and Test Teardown. These keywords define the keyword that gets executed at the start and end of every test case in the suite.

Another important keyword is the Test Template keyword. The Test Template keyword defines what keyword each and every test will call. So for every test case you only need to input the arguments that need to be fed into this keyword which you defined with the Test Template keyword. It's clear how it works if we look at the Test Cases section which is coloured in pink in the following example:

```
*** Settings ***

Documentation   A test suite containing tests related to invalid login. These
...             tests are data-driven by they nature. They use a single
...             keyword, specified with Test Template setting, that is called
...             with different arguments to cover different scenarios.
Suite Setup     Open Browser To Login Page
Test Setup      Go To Login Page
Test Template   Login With Invalid Credentials Should Fail
Suite Teardown  Close Browser
Resource        resource.txt


*** Test Cases ***              User Name       Password

Invalid Username                invalid         ${VALID PASSWD}
Invalid Password                ${VALID USER}   invalid
Invalid Username And Password   invalid         whatever
Empty Username                  ${EMPTY}        ${VALID PASSWD}
Empty Password                  ${VALID USER}   ${EMPTY}
Empty Username And Password     ${EMPTY}        ${EMPTY}


*** Keywords ***

Login With Invalid Credentials Should Fail
    [Arguments]  ${username}  ${password}
    Input Username  ${username}
    Input Password  ${password}
    Submit Credentials
    Login Should Have Failed

```

Figure 16: Test Template example

In this example the Test Template is "Login With Invalid Credentials Should Fail" which is a self made keyword (which we will discuss later). This keyword takes a username and password as arguments.

We see now that we can write a test case in one line because we just need to pass arguments to the template keyword. This allows us to very easily implement more tests by just adding an extra line to the Test Cases section with the arguments that the template function requires. This eliminates a lot of code duplication and increases readability a lot.

A.2.2 Variables

```
*** Variables ***
${USERNAME_ON_SITE} xpath=//span[@id='userpannel.username']

```

Figure 17: Example of a variable in a Robot Framework file

Defining and working with the variables in the Robot Framework syntax is a pretty extensive subject and for a more detailed explanation refer to the Variables Section in the Robot Framework Docs.

In general you should keep variables in separate files so that when you want to change the variables

you can't compromise the integrity of the tests. Sometimes you only need one variable that shouldn't be changed anytime soon and then you can implement that directly into the Robot Framework.

As always the Robot Framework has a tabular structure so the variable names are defined in the first column with it's value in the following columns. We have scalars, lists and dictionaries to our disposal which you can access by naming your variables

${SCALAR_NAME}
@{LIST_NAME}
&{DICT_NAME}

respectively.

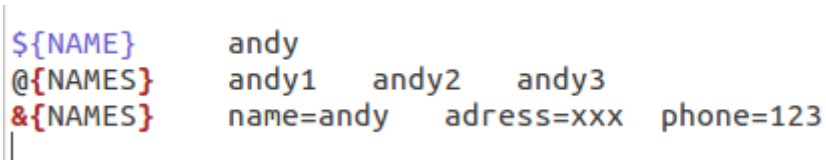

Figure 18: Example creation of different variable types

When you call a variable with:

${EXAMPLE}

the variable will be placed exactly like it was created (you can do this with any variable, not just scalar variables) but if you have a list variable and you want to pass each item in the list as a separate argument to the keyword you are calling then you should call it as:

@{EXAMPLE}

Accessing elements from lists is the same as in Python:

@{EXAMPLE}[index]

where *index* can be negative as well.

Dictionaries can also be passed using their special sign & so that now you can pass the items in the dictionary as named arguments. Accessing specific values that belong to a key is done in the usual way as:

&{EXAMPLE}[key]

Something that the Robot Framework also allows is the use of environment variables. Let's say you have an environment variable called $BROWSER = phantomjs$ then you can access this environment variable from within the Robot Framework file with the following notation:

%{BROWSER}

A.2.3 Test Cases

```
11  *** Test Cases ***
12  LOGIN VIA SYMBOL
13          [Documentation]     Login using the symbol on the home page.
14      OPEN BROWSER    %{BASE_URL} %{BROWSER}
15      CLICK ELEMENT   home.login
16      TITLE SHOULD BE     Login
17      SUBMIT CREDENTIALS  ${ADMIN_USERNAME}   ${ADMIN_PASSWORD}
18      CLICK BUTTON    login.submit
19      ELEMENT TEXT SHOULD BE      ${USERNAME_ON_SITE} ${ADMIN_USERNAME}
20      CLOSE BROWSER
21
22  LOGIN VIA USER ACTION
23      [Documentation]     Login using the login button in the user panel.
24      OPEN BROWSER    %{BASE_URL} %{BROWSER}
25      CLICK ELEMENT   userpannel.action
26      TITLE SHOULD BE     Login
27      SUBMIT CREDENTIALS  ${ADMIN_USERNAME}   ${ADMIN_PASSWORD}
28      CLICK BUTTON    login.submit
29      ELEMENT TEXT SHOULD BE      ${USERNAME_ON_SITE} ${ADMIN_USERNAME}
30      CLOSE BROWSER
```

Figure 19: Example of test cases

In the figure you will see two test cases. Every test case is prefaced by it's title which will show up in the log files. This title should be descriptive enough but documentation can always help. After the title we need to go to the next line and indent the line.

Below the title are all the keywords that need to be executed for that specific test. Tests should be easy to read, anyone should be able to describe step by step what the test does simply by looking at the keywords.

In the columns next to the keywords is where the arguments go for the specific keyword that was called.

There is not much more to it but for additional and more extensive information refer to the Test Case section in the Robot Framework Docs.

A.3 Keywords

Keywords are the main entity in the Robot Framework. These are simply the functions that get called during the execution of the Robot Framework file. There are three ways of geting keywords: Using already made libraries (*standard* or *external*), Combining keywords to get new keywords (User Keywords) or creating your own library (Library Keywords).

Using premade libraries is straightforward. You import the library in the Settings section as described earlier and you look at the documentation for that library to see which keywords are available. The other two keywords will be examined more closely.

A.3.1 User Keywords

There was one section in the example Robot Framework file that we didn't discuss. This was the Keywords section and looks like the following:

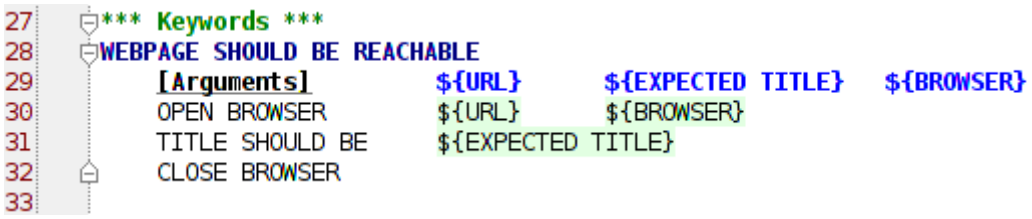

Figure 20: User defined keywords

User keywords are simply functions that call other functions. When you look at the example given you see that the "WEBPAGE SHOULD BE REACHABLE" keyword simply calls the keywords "OPEN BROWSER", "TITLE SHOULD BE" and "CLOSE BROWSER" which are keywords that are implemented in the Selenium2 Library. This gives us the power to execute an entire test in one keyword so that we can use it in the Suite/Test Setup/Teardown keywords. This is also handy to remove code duplication and to abstract away any details.

For instance if I want to login to a page i just want to put my login details in and nobody should be exposed to where the text gets input and which buttons i press. A login user keyword would look like the following:

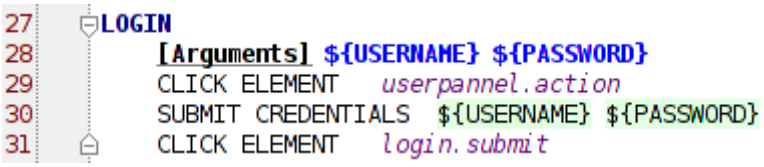

Figure 21: Login User Keyword

This makes reading login tests a lot more natural because only [$LOGIN username password$] will be written in the tests instead of describing every webpage button that gets clicked.

A.3.2 Library Keywords

Library keywords is a very important concept. This is the name of the keywords that come from libraries that you've written yourself. This will be necessary when there is no library out there that does what you need it to do and the *standard* library doesn't suffice.

When making a library yourself you start off with a Java or Python class. You can use either language to extend the Robot Framework. For each test (as default but you can change this) a new instance of the class will be instantiated. It's member functions are the keywords that you can use in your tests. Let's look at a concrete example for sending, downloading and deleting files via http requests. It has to be noted that for http requests there exists a premade library which anyone who does http request testing should use but for this example I've made my own library using the same Python library (Requests) as the premade library that is available on the Robot Framework homepage.

```
*** Settings ***
Variables  auth_vars.py
Library  CrlCpLib.py    ${USERNAME}    ${PASSWORD}
Suite Setup     CHECK CREDENTIALS   ${BASE_URL}          # Checks if we can lo

*** Variables ***
${BASE_URL}   https://prometheus.desy.de/Users/${USERNAME}/Private/

${FILE_DIR}     /home/andy/
${FILE_NAME}    CMSCalorimeter.png


*** Test Cases ***
FILE COPY
    COPY FILE TO    ${FILE_DIR}${FILE_NAME}   ${BASE_URL}${FILE_NAME}
    STATUS CODE SHOULD BE   201

FILE DELETE
    DELETE FILE  ${BASE_URL}${FILE_NAME}
    STATUS CODE SHOULD BE  204

FILE DOWNLOAD
    DOWNLOAD FILE AS    ${BASE_URL}notice    ${FILE_DIR}notice
    STATUS CODE SHOULD BE   200

```

Figure 22: Robot Framework file of http requests testing

There is an extra thing that I want to put the focus on first. If we look at the Library import in this example we see that next to the library filename CrlCpLib.py there are two arguments. These are the arguments that you would pass to the class constructor. So everytime a class of the library gets created these arguments get passed to the constructor of that class.

If we take a look at the CrlCpLib.py file we see the following:

```python
import requests

class CrlCpLib:

    def __init__(self, username=None, password=None):
        self.status_code = None
        self.status_text = None
        self.user = username
        self.password = password

    def check_credentials(self, base_url):
        r = requests.get(base_url, auth=(self.user, self.password))
        self._set_status_vars(r.status_code, r.reason)
        self.status_code_should_be(200)
        return self.status_code

    def open_file(self, filename):
        r = requests.get(filename, auth=(self.user, self.password))
        self._set_status_vars(r.status_code, r.reason)
        return self.status_code

    def copy_file_to(self, filename, destination):
        with open(filename, 'rb') as data:
            r = requests.put(destination, data=data, auth=(self.user, self.password))
        self._set_status_vars(r.status_code, r.reason)
        return self.status_code

    def delete_file(self, filename):
        r = requests.delete(filename, auth=(self.user, self.password))
        self._set_status_vars(r.status_code, r.reason)
        return self.status_code

    def download_file_as(self, file_to_download, local_filename):
        r = requests.get(file_to_download, stream=True, auth=(self.user, self.password))
        with open(local_filename, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1024):
                if chunk:  # filter out keep-alive new chunks
                    f.write(chunk)
```

Figure 23: Piece of the CrlCpLib class

As you can see these are just regular Python functions. If a function starts with _ it will be a "private" function and you won't be able to call that function from Robot Framework. Any _ inside a function name can be replaced by a space in the keyword call in the Robot Framework file. For example the function "open_file" will be the keyword "open file" (or "OPEN FILE" as is the convention).

When a function fails to execute the test will be a fail but let's say you want to download a file but the status code indicates that the file isn't there, the function still executed correctly but in the test this should be flagged as a fail. When writing a library yourself you need to make sure that you raise errors when there is behaviour that you want to be interpreted as a failed test or to change some class variable when executing functions and have a function that checks this variable and throws the correct errors.

All of this can be implemented in Java classes in the same way. You can even mix Python and Java libraries in the Robot Framework file. If someone has written a great library in Python to manipulate files you can still use it while writing your own classes, which are responsible for something else, in Java.

For more information about extending the Robot Framework please refer to the Extending Robot Framework section in the Robot Frameworks Docs.

A.4 Executing and Examining the Results of a Robot Framework Test

To run a Robot Framework file we just need to execute a simple shell command: "robot robot_file_name.robot". When executing this command in a shell you will receive the following output:

```
andy@Andy-Lap:~/PycharmProjects/RobotTest$ robot OrthoTest.robot
==============================================================================
OrthoTest
==============================================================================
Orthogonal Vectors (Functional)                                       | PASS |
------------------------------------------------------------------------------
Non Orthogonal Vectors (Functional)                                   | PASS |
------------------------------------------------------------------------------
Zero Vector (Unit)                                                    | PASS |
------------------------------------------------------------------------------
Dimension Mismatch (Unit)                                             | PASS |
------------------------------------------------------------------------------
Machine Epsilon Stability (Functional)                                | PASS |
------------------------------------------------------------------------------
OrthoTest                                                             | PASS |
5 critical tests, 5 passed, 0 failed
5 tests total, 5 passed, 0 failed
==============================================================================
Output:  /home/andy/PycharmProjects/RobotTest/output.xml
Log:     /home/andy/PycharmProjects/RobotTest/log.html
Report:  /home/andy/PycharmProjects/RobotTest/report.html
```

Figure 24: Shell output after Robot Framework execution

We will get the name of the suite and then the name of each testcase with their description (if it's implemented) and the status of the test (PASS or FAIL). Next to this output you get three output files which explain the tests in more detail.

There are a number of extra arguments that you can pass to the robot command. One that was particularly useful was to change the names of the three output files. When running several Robot Framework files you don't want the output files of one Robot Framework file to overwrite those of the others so the best way to handle this is to change the names of the output files with the -o "output˙filename", -l "log˙filename" and -r "report˙filename" tags which change the names of the output.xml, log.html and report.html files respectively.

After executing several Robot Framework files individually you probably want one big log file that tells you how each suite (Robot Framework file) went. We can combine all of the output.xml files from each suite by calling the rebot tool. This is a tool that comes with the Robot Framework and is designed specifically for this purpose. By calling "rebot output1.xml output2.xml" you will combine all the logs of suite 1 and 2. This is also something that has to be done when integrating Robot Framework into Jenkins but this will be discussed later.

For more information about executing test cases please refer to the Executing Test Cases section of the Robot Framework Docs.

The output file which contains the most information is the log.html file and will look like the following:

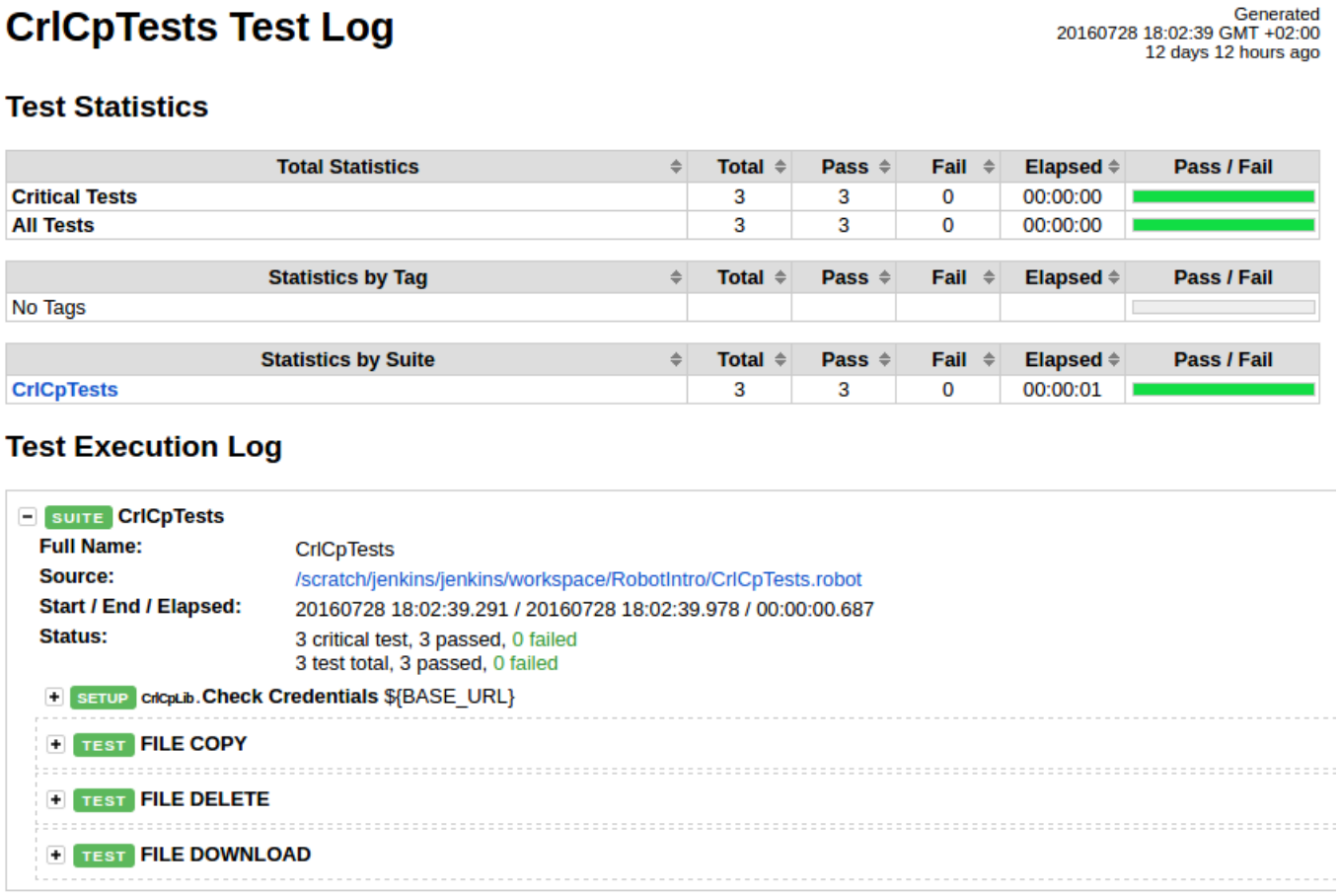

Figure 25: Log.html output file

Every test is expandable into it's component keywords and you can investigate how the execution of a specific keyword went.

Figure 26: Expanded Log.html file

If we made the keywords ourselves we could add extra info to the output files by simply using print in

Python or the corresponding standard output command in Java. There are also basic "LOG" keywords to add extra information to the output files. Let's say that we want the status code and message of our HTTP request displayed in the logs. Because we made the library ourselves we can just add a simple print statement in our "STATUS CODE SHOULD BE" keyword like this:

```
42      def status_code_should_be(self, expected_status_code):
43          if not int(expected_status_code) == self.status_code:
44              raise AssertionError("Expected status code: " + str(expected_status_code) + " but
45          else:
46              print "Status Code: " + str(self.status_code)
47              print "Status Message: " + str(self.status_text)
```

Figure 27: Adaptation to status keyword

and this will be represented in the log files as:

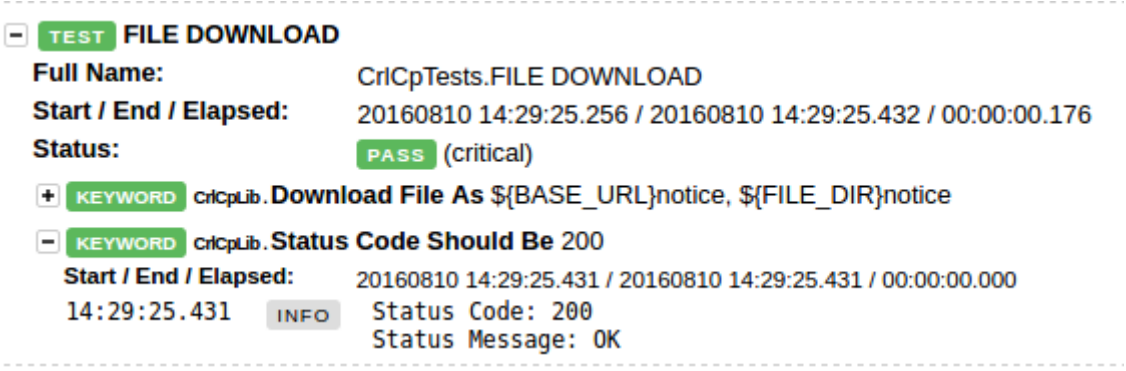

Figure 28: Adapted log info

The output files are pretty easy to understand, for a more extensive dissection of the output files please refer to the Created Outputs section of the Robot Framework Docs.

A.5 Jenkins Integration

Intagrating Robot Framework results into Jenkins is pretty straightforward but there are a couple of caveats. The first thing that you need to do is install the Robot Framework plugin:

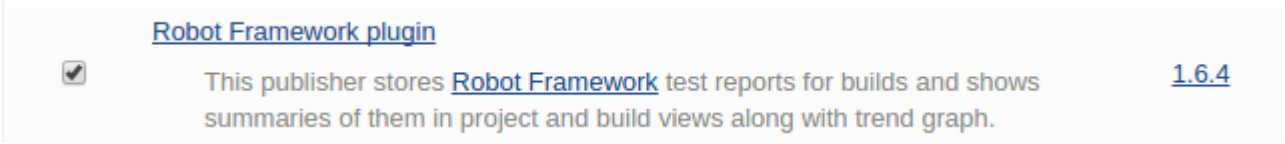

Figure 29: The Robot Framework Plugin installed

after the installation you will have to configure a job to use this plugin according to the following steps:

1. Configure your project
2. Select: Build → Add build step → execute shell
3. Add the execution command discussed earlier. Note that you should rename your output files when executing multiple Robot Framework files.

4. Force your Robot Framework script to return successfully from the shell with "exit 0" because when a test fails the Robot Framework will return an error code and none of the other commands in the shell script will get executed. We did this by adding a "|| true" at the end of our Robot Framework execution command

5. If executing several Robot Framework files combine the output files with rebot and be sure to use "−−output output.xml" as an extra tag for rebot because the Robot Framework plugin requires an output.xml

6. Select: Post-build Actions → Publish Robot Framework test results

7. Set path where your results are located

8. If your output files are named differently than default, specify the filenames by pressing the "Advanced..." button and write the names in the relevant fields

9. If you have other artifacts such as screenshots that you want to persist for viewing later in the logs you must configure them under "Advanced... → Other files to copy"

10. Set thresholds and optionally disable thresholds for critical tests only to count every test in the pass percentage

Besides having to force a succesful exit of the shell after executing the robot command you will notice that you won't be able to open the log files from within Jenkins. There is a conflict between the Robot Frameworks output html files (which contain javascript, images and css) and Jenkins' CSP which doesn't allow this by default. If you want to find more about this you can refer to bug JENKINS-32118. To work around this issue log into Jenkins, Manage Jenkins → Script Console and run the following script:

```
System.setProperty("hudson.model.DirectoryBrowserSupport.CSP","sandbox allow scripts; de
```

This will update the CSP Jenkins is returning in the response headers of GET requests so that html files may contain javascript, images and css.

A.6 Steps to take to start working with the Robot Framework

After this short introduction of the Robot Framework I will outline the necessary actions to start testing your own software using the Robot Framework.

1. Search for a keyword library that fits your needs.

2. If there is no library for you, you'll have to make one yourself which is simply a Python or Java class from which the public member functions are the keywords.

3. Make a new .robot file and divide it into the 4 sections: Settings, Variables, Test Cases and Keywords.

4. Optionally: Remove Variables and Keywords sections from the .robot file that contains the Test Cases and put them in a separate .robot, Java or Python file.

5. Import all necessary Libraries in the Settings section of the .robot file.

6. Set the prerequisite tests in the Suite/Test Setup/Teardown keywords in the Settings section of the .robot file.

7. Write your test cases avoiding code duplication and abstracting away as much of the unnecessary code details as possible. The test cases should be understandable with very limited coding knowledge.

8. Execute the .robot file using the command "robot robot_filename.robot".

9. Examine the output files and hope everything is green.

References

[1] Günther Quast, Karlsruhe Institute of Technology, From raw data to Pbytes on disk: the world wide LHC computing grid.

[2] A. Scheurer et al. Institute for experimental nuclear physics, University Karlsruhe, Challenges of the LHC computing grid by theCMS experiment

[3] Lucas Taylor, CMS Experiment, Triggering and Data Acquisition, `http://cms.web.cern.ch/news/triggering-and-data-acquisition`

[4] Patrick Fuhrmann, DESY, dCache, the Overview, `https://www.dcache.org/manuals/dcache-whitepaper-light.pdf`

[5] Andy Martinez Nieto, robot-webadmin-tests, `https://github.com/AndyMN/robot-webadmin-tests`

[6] Andy Martinez Nieto, GridTools-Functional-Tests, `https://github.com/AndyMN/GridTools-Functional-Tests`

[7] The Robot Framework, User Documentation, `http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html`