Table of Contents

Test writing guidelines (work in progress)

In this twiki we collect some guidelines to write tests for the certification process of gLite components. The main goal is to have test reports as homogeneous as possible and to facilitate test maintenance and automation. These guidelines are particularly useful when the tests are integrated into Yaimgen and the automation scripts.

Test scripts

In our context we define as **test** an execution of a piece of code which is meant to verify the correct behavior of a software component. A test has a set of **pre-conditions**, an **expected outcome**, an **actual outcome** and a **result**. Pre-conditions are conditions that have to be true in order for the test to run, and the actual outcome be meaningful. If we run a test when its pre-conditions are not met, the actual outcome of the test has to be considered UNDEFINED. When the pre-conditions are met, the actual outcome has to be equal to the expected outcome (PASS or FAIL). The test result comes from comparing the actual outcome with the expected one, and can be:

- EXPECTED PASS: the test was expected to pass, and it passed.
- EXPECTED FAIL: the test was expected to fail, and it failed. This can happen for example when the tested code has a known bug which is not yet fixed at the moment the test was executed.
- UNEXPECTED FAIL: the test was expected to pass but it failed.
- UNEXPECTED PASS: the test was expected to fail but it passed. This can happen when the test executed is not up-to-date. In this case the test has to be updated expecting to pass, and its execution repeated.
- UNDEFINED: the tests pre-conditions are not met, therefore the test outcome is not meaningful.

Moreover, in practice a test also has a cleanup part, where some cleaning operations can be done on the system under test. Hence, four major parts can be identified:

- pre-test: where the precondition of the test are set and verified.
- test body: where the actual test is implemented.
- post-test: where the cleaning operations are performed.

It could be useful in some cases to know where (in which of these three parts) a test is failing. A failure in the test body can lead to one of the following outcomes: EXPECTED FAIL, UNEXPECTED FAIL. A failure in the pre-test always leads to the UNDEFINED outcome. For what concerns a failure in the post-test, in this case the test itself passes, but the post-tests failure could lead to problems in other tests. An additional outcome like PASSED WITH WARNING could be used in this case. PASSED WITH WARNING could be used also in cases where a threshold exists in the result, and the actual result passes but is "too close" to the threshold that is worth generating a warning.

Tests, as just defined, are run using scripts (in our case normally using Bash, Perl or Python). Often a script includes more tests. For example to test a CLI command, a single script could test the correct behavior using all the available combination of options, including wrong values. We therefore define as **testscript** a script which collects one or more tests.

The outcome of a testscript is a function of the outcomes of its tests, and has the following values:

- PASSED: when all its test outcomes are either EXPECTED PASS or EXPECTED FAIL. The script **must** have an exit code = 0 (Nagios complaint).
- FAILED: when at least one of its test outcomes is either UNEXPECTED FAIL, UNEXPECTED PASS, UNDEFINED. The script **must** have an exit code = 2 (Nagios will report it as 'Critical'). When the script is not called with the right arguments or when some preconditions are not satisfied, it has to

fail, exit code = 3 (Nagios 'Unknown'), and print the usage information and where the error was in the arguments or which precondition was not verified.
- WARNING: when at least one of it's test outcomes is PASSED WITH WARNING. The script must have an exit code = 1 (nagios 'Warning').

A testscript should have a **long and and short description**. The short description should appear on the testing report associated with a patch which has been certified, so it should give an idea on what is tested running that script with a few words. The short description should be the only thing which is printed in the test report together with the outcome of the testscript. The long description should be available only in rare cases (e.g. can be handy to describe your testsuite with something like *for testscript in `ls $testdir`; do $testscript --longinfo; done*), and not printed in the test report, therefore can explain more than the short description. The short and long descriptions can be printed for example by calling the scripts with a special argument, like --shortinfo and --longinfo. A testscript should require a minimum number of arguments.

The test and testscripts outcome has to be reliable: a testscript outcome of PASS just because some tests are skipped due to a bad setup, or some return values are not well checked will provoke serious damages!

The tests have to be run using test user proxies. Prepare your tests to accept the VOs 'dteam' and 'org.glite.voms-test'.

Metascripts

Metascripts are scripts that are used to call a set of testscripts. Normally each metascript comes with a configuration file that will be sourced to set some specific variables needed by the metascript, like the name of the host to be tested and the location of log files.

A file containing functions that can be re-used by multiple scripts and metascripts should be available to minimize code duplication. For example, checking the validity of a proxy is a 10 lines of bash that is present in many scripts.

From our current experience the metascripts are standard, and a single one could be used to minimize the effort to maintain it. A typical metascript is like the following:

```
 Node-certtest.sh  [-f <config file>] [--node <NODE HOST>] [--arg <ADDITIONAL PARAMETER>]
```

The metascript has to be able to run without option, using the default configuration file usually named **Node-certconfig**. Using the options one has to be able to use a different configuration file, and provide parameters to override the variables defined in the configuration file. Apart from parsing the arguments, setting the correct variables, and creating a directory to store log files, the main part of the metascript is to launch in sequence a set of testscripts, redirecting its output to a log file, and printing its short description and its outcome. Tests that are failed should be collected in an array and at the end the metascript should list which tests are failed with the location of the relative log file. The configuration file should contain only important variables that define the behavior of the tests being run, and behavior that a tester might want to modify.

The metascript output should be compact so that it can be used as a test report to attach to a certification report. It should contain at least the following output:

- Date (start and end)
- Log location
- Important configuration settings
- VO and proxy used
- Test run (with short description), and outcome (PASSED, FAILED, PASSED WITH WARNING)
- Summary (if some tests are failing the location of the logs).

It is important also that the metascript runs a test to check that the rpm present in the patch being certified are actually installed. This rpms could be listed in the metascript output.

CLI testing

These section contains practical hints on developing tests for Command Line Interfaces. These are personal suggestions, if you don't like them and you have something better to propose please do not hesitate in dropping an email to Gianni.Pucciani@cernNOSPAMPLEASE.ch.

- Before starting writing the first test, have a look at all commands you are planning to test, and start thinking from the beginning for possible automation, common function etc...Use uniform structure and terminology in all the test scripts, prepare a template to create new test scripts.
- When testing the CLI of a service, you might want to test some common arguments (e.g --verbose, --help) in a single script, to avoid code duplication. The name of the command will be passed to this script that will check these common arguments once for all the commands in the CLI.
- Check the functionality that are explained in the man pages, or in a user guide. Try to check the use of all the available parameters.

Bash tips

- Calling commands in a uniform way

It is always useful to have a utility function to call a command. It avoids code duplication, reduce sources of errors, and allows for uniform reporting. The detailed output of a test script should show the command executed, in a uniform way, a function like the following one does the job.

```
# ... Run command successfully or return 1
function run_command() {
  echo " $" $@

  OUTPUT=$(eval $@ 2>&1)
  if [ $? -ne 0 ]; then
    echo " --> ${OUTPUT}"
    myecho "$1 failed"
    echo ""
    return 1
  fi
  echo " --> ${OUTPUT}"
  export OUTPUT
  return 0
}
```

This is an improved version which allows also to check the return code against an expected value (last argument):

```
function run_command_status() {
  command=$1
  expstatus=`echo $@ | awk '{print $(NF)}'`
  if ! [[ "$expstatus" =~ ^[0-9]+$ ]] ; then
     echo "Error: expected status not a number: $expstatus"
     return 1
  fi
  declare -a arguments
  for item in `seq 2 $(($#-1))`
  do
    arg=`echo $@ | awk -F' ' '{print $v1}' v1=$item`
    arguments=("${arguments[@]}" "$arg")
  done
  echo " $" $command ${arguments[*]}
  OUTPUT=$(eval $command ${arguments[*]} 2>&1)
  status=$?
  if [ $status -ne $expstatus ]; then
    echo " --> ${OUTPUT}"
    myecho "$command failed"
    myecho "Exit status was $status while expecting $expstatus"
    return 1
  fi
  echo " --> ${OUTPUT}"
  export OUTPUT
  return 0
}
```

- Returning values within a function.

A function should return 0 when executed successfully, non-zero otherwise. Bash does not allow to call a function with an "out" argument, this article explains a useful workaround: http://www.linuxjournal.com/content/return-values-bash-functions

```
function myfunc()
{
    local  __resultvar=$1
    local  myresult='some value'
    eval $__resultvar="'$myresult'"
}

myfunc result
echo $result
```

API testing

These section contains practical hints on developing tests for Application Programming Interfaces. These are personal suggestions, if you don't like them and you have something better to propose please do not hesitate in dropping an email to ...?

General hints for test developers

The following hints are meant to help tests developers to write long lasting and usable scripts.

- Document your code. In the testscript or in a README file explain how to use the test, and what are the requirements needed to run the test if any. You will not be the only one using your scripts, so make other people's life easier!
- Don't use fancy programming styles and complicated commands, we all know you are a good programmer but a newbie might need to put his hands is your code.
- Testscripts should prepare their own environment, and check for the existence of external requirements. Example: if you want to test an lcg-del that removes a file on a SE put the file on the SE and check for it's existence as a pre-condition of the test.
- Avoid as much as possible to make your test depend on other services. Example: when the SE storage area path is retrieved from the BDII the test depends on the BDII working correctly. Define the same path as a variable of the test (in the script or in the configuration file) so that the test can be run without contacting the BDII.
- The testscript has to print all the sensitive commands executed to help debugging in case of failures: looking at the output a tester has to be able to reproduce the failure manually.
- Make sure your tests execute a repeatable work flow. Failures has to be reproducible. A simple function like this to execute shell command can be useful for this purpose:

```
function run_command() {
  echo " $" $@
  OUTPUT=$(eval $@ 2>&1)
  if [ $? -ne 0 ]; then
    echo "${OUTPUT}"
    myecho "$1 failed"
    myexit 1 $1
  fi
  echo "${OUTPUT}"
  export OUTPUT
  return 0
}
```

- When testing CLIs or APIs you might want to have some common functions to be used by different testscripts: this will help the maintainability of the tests. As soon as you start cut and paste code from different tests consider putting this code in a common function.

-- GianniPucciani - 11-Dec-2009

This topic: Main > TestWritingGuidelines
Topic revision: r13 - 2010-06-16 - unknown