ART – Automated REST Testing

User Guide

Authors: Elena Dolinina, Lukas Bednar

Revision 0.2 19. Aug. 2012

Table of Contents

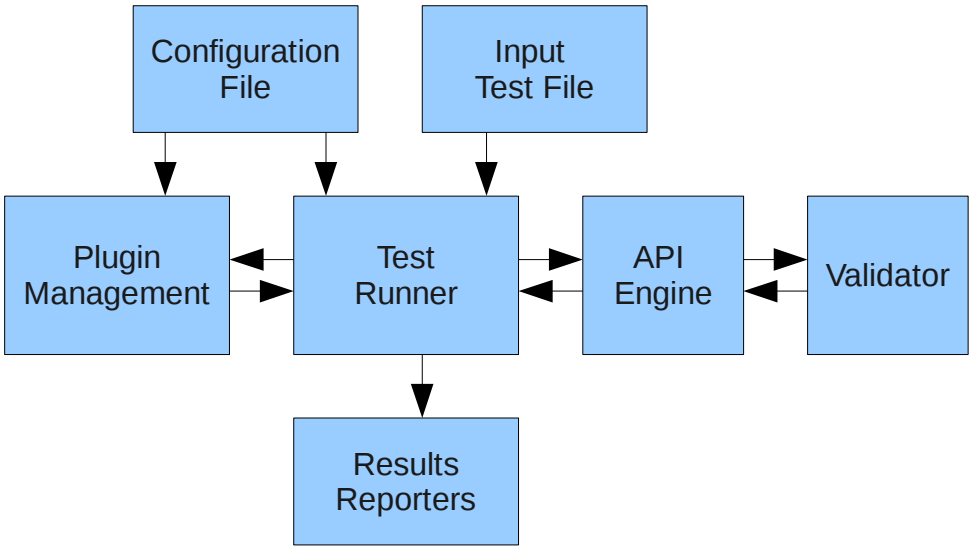
1 Introduction

1 ART

Automated REST Testing (ART) is a framework for automated tests of REST APIs. ART works via product xsd file (xml schema documents) that describe the structure of all possible *xml* requests and responses. You can run your REST API requests easily with ART and get them validated automatically. Your tests results are generated in *xml* and *junit* formats so it can be easily integrated with other tools.

ART is a plugin based so new features and functionality can be added easily by extending one of the existed plugins or writing your own.

2 ART structure



3 Features

- Builds python data structures based on XML Schema document. The data structures represent the elements described by the XML Schema. ART has parser that loads XML document into data structures and vice versa . *generateDS* tool is used for this purpose. For more information on *generateDS*, see: http://cutter.rexx.com/~dkuhlman/generateDS.html.
- Sends all types of REST API requests: GET, PUT, POST, DELETE.
- Maps all test functions to some known action in the configuration file. These actions can be accessed from input test scenario file.
- Supports 3 formats for input test scenario file: ODS Spreadsheet, XML file, Python script.
- Can use different placeholders in input file to get parameters from configuration file.
- Fetches output from one action to be reused in further test actions.
- Can group tests into test sets.
- Can run tests in loop and by conditions.
- Can easily add request headers via configuration file.

- Validates the responses via sent request content and .xsd file.
- Generates simple xml and junit tests results. Report can be configured to contain sub-tests (nested reports).
- Supports different REST based engines (sdk, cli, etc.).
- Supports parallel runs from the same configuration file or from different ones.
- Supports plugin management for new functionality to main engine (Test Runner). Based on Trac Component Architecture: http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture

4 System Requirements

- Red Hat Enterprise Linux Server release 6.2
- Mozilla Firefox 10 and higher

2 Installation

Clone the project from git repository:

git clone git://git.engineering.redhat.com/users/edolinin/ART.git

Also install the following Python packages:

yum install python-lockfile odfpy python-tpg pexpect python-lxml python-psycopg2

3 generateDS – Data Structures Auto-Generation

To run your REST requests the data structures should be generated. ART uses *generateDS* tool for this purpose. You just need to run the script *generateDS/generateDS.py* provided with your xml schema documents file:

python generateDS.py -f -o /tmp/data_structures.py --member-specs=dict ../conf/api.xsd

Where:

../conf/api.xsd – your input xml schema documents file

/tmp/data_structures.py – path to the output file where data structures will be written

As a result of this script you will get an auto generated Python module with all data structures required for testing your REST API application. To provide it to your main configuration file, see the next section.

4 Configuration

Open *conf/settings.conf* file. This is an example of your main configuration file, it contains the following sections:

1 [RUN] section

- engine – if you have several engines supported in project and based on REST – sdk, cli, etc. - you can switch between the engines by setting the proper value of this parameter. By default the 'rest' engine is used.
- tests_file - name of the file which contains your test scenario. If this file is located in the *conf/* folder, you only need to specify its relative name; otherwise specify the absolute path to the file. You can list here several files that should be run with comma separation. The input file can be in XML format, ODS Spreadsheet or a python script.
- tests_sheets - the sheets from your test scenario file which should be run, relevant only for ODS files.
- in_parallel – optional parameter for parallel runs. List of the sheets or xml files which should be run in parallel. You can run different sheets/xml files or the same sheet/xml file with different configurations files in parallel. See the

example below.

- parallel_configs - optional parameter for parallel runs. Paths to configuration files can be set here (separated by commas) for each of the sheet or xml file running in parallel. If this parameter is omitted - default configuration file of test is taken.
- debug - if the test should be run in debug mode or no, if this parameter set to 'no', only INFO and ERROR messages will be printed to the log file
- data_struct_mod – path to data structures file generated by *generateDS* (see the previous section how to create it)
- api_xsd – path to xml schema documents file
- media_type – application/xml for *xml* format, application/json for *json* format

Example for RUN section simple configuration:

[RUN]

engine = rest

tests_file = /tmp/rest.xml

data_struct_mod = data_struct.data_structures

api_xsd = /data_struct/api.xsd

debug = yes

media_type = application/xml

Example for the same xml file running in parallel with different configuration files:

[RUN]
tests_file = /tmp/test.xml, /tmp/test.xml
in_parallel=/tmp/test.xml, /tmp/test.xml # same or different xml files can be set here
parallel_configs = conf/settings.conf, conf/settings2.conf # paths to config files, if omitted - default config is set

2 [REST_CONNECTION] section

This section defines parameters for connecting to REST API application:

- scheme - 'http' or 'https'
- host - your REST client machine
- port - port which is used by REST APIs
- entry_point - REST APIs main url suffix (for an example for <host>:<port>/api/ entry_point=api)
- user – user name to connect to REST APIs client
- user_domain - user domain

Example for the secured http connection:

[REST_CONNECTION]
scheme = https
host = host.redhat.com
port = 443
user = admin
password = 123456
entry_point = api

user_domain = qa.lab.redhat.com

3 [PARAMETERS] section

This section defines global parameters used by the test. They are optional and can be removed or extended depending on test scenario definition (in your input test scenario file). Any of these parameters could be a single value or a list or values (separated by commas). All these parameters can be accessed from test scenario file by different place holders.

4 Customized sections for parallel run

You can run the same test file in parallel with different parameters from one configuration file. For this purpose you should define new configuration sections with required parameters for each of the parallel threads. Here is an example:

[RUN]
tests_file = /tmp/test.xml,/tmp/test.xml
in_parallel=/tmp/test.xml,/tmp/test.xml
parallel_sections = test1, test2
...

[TEST1]
name=test1

[TEST2]
name=test2

5 [REPORT] section

This section defines parameters required for tests results reporting.

- has_sub_tests - 'yes' if each test should be reported as a sub-test of its group/testing module, 'no' if each test should be reported as independent
- add_report_nodes - list of nodes names if additional nodes besides the default ones (defined in input xml nodes) should be added to report, 'no' otherwise. Example of added nodes format string: add_report_nodes=version, name.

  Note: If you want to add additional nodes to the report make sure that the function you use in your test scenario returns these properties in its output dictionary

5 Input Test Scenario File

This file defines the test scenario. Each test case in this file should contain the following attributes (xml nodes for *.xml* file, column for *.ods* file, TestCase object attributes for *.py* file):

test_name - mandatory

- Name of the test which will appear in reports

test_action - mandatory

- Action which will run to implement the test, each action is mapped to the function in conf/actions.conf file

parameters - mandatory

- Parameters received by the test. Parameter names should correspond to the names of test function parameters. You can put parameters names from conf/settings.conf file here as place holders. Just put parameter name in {} brackets and it will be replaced with the relevant value during the run time. If you want to get product constants from conf/elements.conf section call it e{param_name}. If you have a comma separated list in conf/settings.conf you can fetch its value in a way of array indexing - {name_of_param[param_index]}. If you want to get these listed as a string you can call it as [name_of_param]. If you are using loop in 'run' column (see below) you can concatenate loop iteration index to any of your parameters. Here are a few examples for values of parameters column:

Get parameter 'compatibility_version' from settings

name='Test1',version='{compatibility_version}'

Get parameter 'storage_type_nfs' from conf/elements.conf

name='Test1',storage_type='e{storage_type_nfs}'

Get first parameter from list of 'vm_names' from settings

name='{vm_names[0]}'

Get list of 'vm_names' names as a string from settings

name='[vm_names]'

Add iteration index to name (relevant when running in loop)

name='testVM#loop_index'

positive - mandatory

- TRUE if the test should succeed, FALSE otherwise. You can set NONE if your function should not use this parameter at all.

run - mandatory

- 'yes' if the test should be run, 'no' if you want to skip the test. Also, you can specify a Python condition here:

Simple condition:

if(<condition>)

Action condition:

ifaction(<action_name>,<action parameters>) # test will run if action returns True
not ifaction(<action_name>,<action parameters>) # test will run if action return False

Loop and forkfor:

`Loop` can be used to make several same operations. While `loop` does all iterations one after another , `forkfor` executes them simultaneously. Both can be used with groups so the contents of the group will get executed several times.

loop(<number_of_iterations>) loop(<iterations_range>) loop({<parameter_name_to_iterate_on>})

forkfor(<number_of_iterations>) forkfor(<iterations_range>) forkfor({<parameter_name_to_iterate_on>})

Note that Python might not be able to create more than 600 threads, therefore `forkfor(700)` may fail. In addition, executing too many requests can lead to load problems on remote side.

'if' and 'loop' together:

if(<condition>);loop(<number_of_iterations>)

Examples:

Simple condition:

if('{compatibility_version}'=='2.3')

if(len({hosts})>2) # will run if number of values in 'hosts' parameter in configuration file is greater than 2

Action conditions:

ifaction(activateHost,'TRUE',name='{host}')
not ifaction(activateHost,'TRUE',name='{host}')

Loop statements

loop(5) loop(5-10) loop({os_name})

'if' and 'loop' together:

if('{compatibility_version}'=='2.3');loop(5)

You can iterate over several parameters at once. It can be useful for an example for host installation. If you want to install several hosts which all have different passwords, define the following parameters in the settings.conf file:

hosts = host1,host2,host3
password = pass1,pass2,pass3

Then in your input file put the following in the 'parameters' field:

host={host},password={password}

And in 'run' field:

loop({host},{password})

Your test will run for 3 times and each time the required action will be run with the hostname and password relevant to the current iteration.

report – optional (default is 'yes')

- 'yes' if the test results should be reported (in the results.xml file), 'no' if the test results should not be reported. If this column is omitted all tests are reported.

fetch_output - optional

- If your function returns some value that you want to use in further tests, specify it in this column. It assumed that the function will return additional values (besides status) in dictionary format. Specify the key name related to the desired output value and the parameter name of where the key will be put. The format of this value should be the following:

<fetch_output_from_key_name>-><fetch_output_to_parameter_name>

- Examples:

  osName->myOsName

You can use parameters place holders in <fetch_output_to_parameter_name> (can be useful in parallel runs)

osName->osName{index}

Then you can use this fetched value as parameter in your further tests:

vm='MyVm',os_name=%myOsName%

or with parameters place holders:

vm='MyVm',os_name=%osName{index}%

or to concatenate fetched output to another string:

vm='MyVm',os_name='test' + %osName{index}%

You can fetch several output parameters in the same manner, just separate them with commas. For example:

osName->myOsName, osType->myOsType

If the function returns a Python list type object, it's possible to reference the individual items like this later on:

name=%out%[1]

vital – optional (default is 'no')

- 'yes' if the test is vital and further tests cannot continue without its success, 'no' otherwise. If a vital test fails all further tests will not be run.

tmcs_test_case - optional

- The numeric identifier of the TCMS test case related to your REST-Framework test case.

test_description - optional

- Detailed description of your test.

conf – optional

- Option to change the global configuration settings for each test case. For example if you want to change user and http headers for one of the tests you can set the following:

  headers={'Filter': true}, user='new_user'

Grouping tests in input file into test sets

You can group your tests according to their functionality, flows, testing purpose, and so on. To define a custom group in input file, add a new test case before the tests that should be grouped. Fill this test case as follows:

- test_name=START_GROUP: <name_of_tests_group>

- All other fields except 'run' are not applicable and can be set as 'n/a' or skipped.
- The 'run' value should be set if you want to run the whole group of tests in a loop or by certain conditions. See 'run' possible values above. All test case ‘run’ possible values are applicable also in scope of tests groups.

To mark where the test group is finished add a new test case after the last test of the group and fill it as follows:

- test_name=END_GROUP:<name_of_tests_group>. Name of the test group here must be the same as the value specified in START_GROUP .

Test Templates

You can find samples of test scenarios files at tests/xml_templates/ folder.

6 Actions configuration

Each action you are going to use in your input test scenario file should appear in the *conf/actions.conf* file. This file is a mapping of action name to its function path. When you add a new testing function, add its mapping in the *conf/actions.conf* so this function can be accessed by Test Runner. Here is an example of a *conf/actions.conf* file :

[ACTIONS]

firstTest=root_module.test_module.firstTestFunction

secondTest=root_module.test_module.secondTestFunction

utilFunction=util_module.utilFunction

7 Running with different input files

Your test scenario can be created in 3 possible formats: XML file, ODS Spreadsheet or Python script. The examples below show how to define test cases for all of these formats.

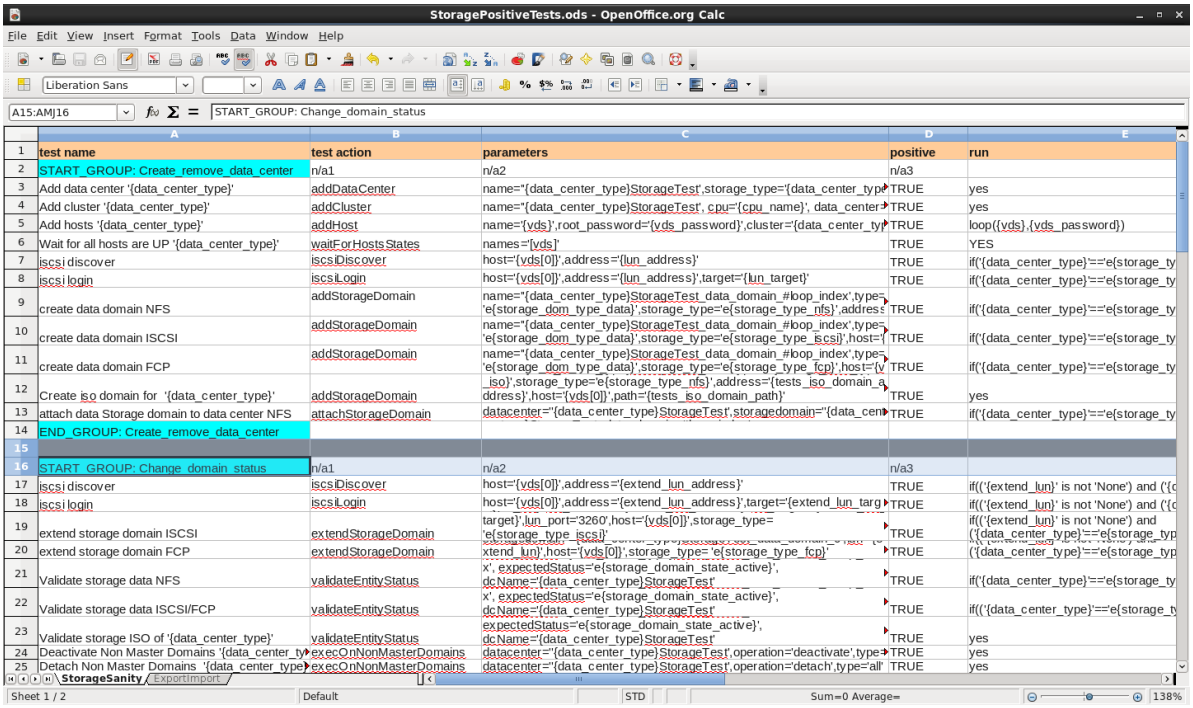
1 Running from ODS file

Open a spread sheet of *.ods* file and define the following columns headers:

test_name, test_action, parameters , positive, run, report, and all other required tests attributes.

After you have defined the columns headers, each test case is represented as a new line in your *.ods* file. To create a test case add a new line and fill each its cell according to the column it belongs to.

Here is an example of a test scenario in an *.ods* file:



	test name	test action	parameters	positive	run
2	START_GROUP: Create_remove_data_center	n/a1	n/a2	n/a3	
3	Add data center '{data_center_type}'	addDataCenter	name='{data_center_type}StorageTest',storage_type='{data_center_type	TRUE	yes
4	Add cluster '{data_center_type}'	addCluster	name='{data_center_type}StorageTest', cpu='{cpu_name}', data_center	TRUE	yes
5	Add hosts '{data_center_type}'	addHost	name='{vds}',root_password='{vds_password}',cluster='{data_center_ty	TRUE	loop({vds},{vds_password})
6	Wait for all hosts are UP '{data_center_type}'	waitForHostsStates	names='{vds}'	TRUE	YES
7	iscsi discover	iscsiDiscover	host='{vds[0]}',address='{lun_address}'	TRUE	if('{data_center_type}'=='e{storage_ty
8	iscsi login	iscsiLogin	host='{vds[0]}',address='{lun_address}',target='{lun_target}'	TRUE	if('{data_center_type}'=='e{storage_ty
9	create data domain NFS	addStorageDomain	name='{data_center_type}StorageTest_data_domain_#loop_index',type= '{storage_dom_type_data}',storage_type='e{storage_type_nfs}',address	TRUE	if('{data_center_type}'=='e{storage_ty
10	create data domain ISCSI	addStorageDomain	name='{data_center_type}StorageTest_data_domain_#loop_index',type= '{storage_dom_type_data}',storage_type='e{storage_type_iscsi}',host='{	TRUE	if('{data_center_type}'=='e{storage_ty
11	create data domain FCP	addStorageDomain	name='{data_center_type}StorageTest_data_domain_#loop_index',type= '{storage_dom_type_data}',storage_type='e{storage_type_fcp}',host='{v	TRUE	if('{data_center_type}'=='e{storage_ty
12	Create iso domain for '{data_center_type}'	addStorageDomain	_iso}',storage_type='e{storage_type_nfs}',address='{tests_iso_domain_a ddress}',host='{vds[0]}',path='{tests_iso_domain_path}'	TRUE	yes
13	attach data Storage domain to data center NFS	attachStorageDomain	datacenter='{data_center_type}StorageTest',storagedomain='{data_cen	TRUE	if('{data_center_type}'=='e{storage_ty
14	END_GROUP: Create_remove_data_center				
15					
16	START_GROUP: Change_domain_status	n/a1	n/a2	n/a3	
17	iscsi discover	iscsiDiscover	host='{vds[0]}',address='{extend_lun_address}'	TRUE	if('{extend_lun}' is not 'None') and ('{
18	iscsi login	iscsiLogin	host='{vds[0]}',address='{extend_lun_address}',target='{extend_lun_targ	TRUE	if('{extend_lun}' is not 'None') and ('{
19	extend storage domain ISCSI	extendStorageDomain	target}',lun_port='3260',host='{vds[0]}',storage_type= '{storage_type_iscsi}'	TRUE	if('{extend_lun}' is not 'None') and ('{data_center_type}'=='e{storage_typ
20	extend storage domain FCP	extendStorageDomain	xtend_lun}',host='{vds[0]}',storage_type='e{storage_type_fcp}'	TRUE	('{data_center_type}'=='e{storage_typ
21	Validate storage data NFS	validateEntityStatus	x', expectedStatus='e{storage_domain_state_active}', dcName='{data_center_type}StorageTest'	TRUE	if('{data_center_type}'=='e{storage_ty
22	Validate storage data ISCSI/FCP	validateEntityStatus	x', expectedStatus='e{storage_domain_state_active}', dcName='{data_center_type}StorageTest'	TRUE	if('{data_center_type}'=='e{storage_ty
23	Validate storage ISO of '{data_center_type}'	validateEntityStatus	expectedStatus='e{storage_domain_state_active}', dcName='{data_center_type}StorageTest'	TRUE	yes
24	Deactivate Non Master Domains '{data_center_ty	execOnNonMasterDomains	datacenter='{data_center_type}StorageTest',operation='deactivate',type	TRUE	yes
25	Detach Non Master Domains '{data_center_type	execOnNonMasterDomains	datacenter='{data_center_type}StorageTest',operation='detach',type='all'	TRUE	yes

2 Running from XML file

Here is an example of an *xml* input file:

```
<input>
<test_case>
  <test_name>START_GROUP: Test1</test_name>
  <test_action/>
  <parameters/>
  <positive/>
  <run>loop(5)</run>
  <report/>
</test_case>
<test_case>
  <test_name>Create NFS Data Center</test_name>
  <test_action>addDataCenter</test_action>
  <parameters>name='dc1',storage_type='NFS',version='2.2'</parameters>
  <positive>TRUE</positive>
  <run>yes</run>
</test_case>
<test_case>
  <test_name>END_GROUP: Test</test_name>
  <test_action/>
  <parameters/>
  <positive/>
  <run/>
  <report/>
</test_case>
</input>
```

3 Running from Python script

Here is an example of running a test case from a Python script:

```
from test_handler.python_runner import TestCase
def addCluster():
    test = TestCase()
    test.name = 'Add Cluster'
    test.action = 'addCluster'
    test.positive = True
    name = test.config['PARAMETERS'].get('cluster_name')
    version = test.config['PARAMETERS'].get('compatibility_version')
    test.params = "name='{0}', version='{1}', cluster='Test',\
            wait=False".format(name, version)
    test.run()
    test.logger.info(test.status)
    test.logger.info(test.output)
```

8 Run The Test

1 Compile *ods/xml* file for local run

To check there are no errors in your *ods* or *xml* files, run a compilation beforehand. It will validate your input file but not run the tests:

python run.py ---compile

In case there are errors in your ods/xml file you will see them in your console. Fix them as required.

2 Run test

When your configuration is ready run the test from your working folder. The test requires several parameters:

- configFile (conf) – Mandatory path to the *settings.conf* file .
- resultsXmlFile (res) – Optional path to the *results.xml* file, the default is *results/results.xml.*
- log – Optional path to the log file, default is */var/log/restTest<timestamp>.log*

See run.py – help for other possible options.

If you want to run the test with default parameters run the main script as:

python run.py -conf=<config_path>

3 Run only specific lines or test groups from input file

You can run only specific lines from your *.ods* file or specific test cases from *xml* file. In this case all other lines or test cases will be skipped. Lines numbers should be greater than 1 as the first line in *.ods* file is actually a header.

python run.py --lines=50-60,70,80-90,100

You can run only specific test groups and skip all other groups :

python run.py --groups=Test1,Test3,Test5

The test will run and report all its actions to your console. Test results will be reported to *results/results.xml* and *junit_results.xml* files . The log file can be found at */var/tmp/restTest_<timestamp>.log* or at the location specified at -log cli option.

4 Run externally (without settings, ods and run.py)

If you want to use REST APIs functions in your own code independent of the whole framework you can do it with the RestTestRunnerWrapper instance. For example:

from utils.restutils import RestTestRunnerWrapper
restWrapper = RestTestRunnerWrapper('10.35.113.80') # provide ip of your rest client server
try:
status = restWrapper.runCommand('rest.datacenters.addDataCenter', 'true', name='test',storage_type='NFS', version='2.2') # run the function via wrapper, first parameter is a function path, then a list of function's parameters

except RestApiCommandError:
...

9 Reports

Test reports are located by default in *results/results.xml.* You can also specify a customized location using the *--resultXmlFile/--res* parameter during a test run. The test report header has the following attributes:

- logfile – path to a test log file
- testFile – input test file name

All tests results appear as sub nodes of the related group or just 'test' node in case of independent tests. Names of test statistics nodes depend on parameters set in the REPORT section of your *settings.conf* file, .The following default nodes are always added:

- start_time - time stamp when test started
- end_time - time stamp when test finished
- status - test status (Pass/Fail)
- iter_num - test iteration number
- all test cases attributes you used in your input file (test_name, parameters, positive, etc.)

10 Currently Available Plugins

1 Hosts Cleanup

This plugin removes Storage and Network leftovers from your VDS hosts machines as defined in your configuration file.

CLI Options

--cleanup enable cleanup plugin

Configuration File Options

RUN.auto_devices – Enabling of Auto Device Plugin. If this is set to "yes" the storage cleaning will be skipped.

PARAMETERS.vds – List of VDS machines which will be cleaned up.

PARAMETERS.vds_user – User name for root account.

PARAMETERS.vds_password – List of passwords for the root account of each host.

PARAMETERS.mgmt_bridge – Name of network bridge, the default is 'rhevm'.

2 Bugzilla Plugin

This plugin provides access to the Bugzilla site. Tests can be skipped accordingly to 'bz' attribute of each test case.

CLI Options

--with-bz – Enables the Bugzilla plugin.

--bz-user BZ_USER – User name for Bugzilla , the default is 'bugzilla-qe-tlv@redhat.com'.

--bz-pass BZ_PASS – Password for the Bugzilla , the default is '2kNeViSUVO'.

--bz-host BZ_HOST – URL address for Bugzilla , the default is https://bugzilla.redhat.com/xmlrpc.cgi

Configuration File Options

BUGZILLA.enabled – true/false; equivalent to with-bz CLI option

BUGZILLA.user – equivalent to bz-user CLI option

BUGZILLA.password – equivalent to bz-pass CLI option

BUGZILLA.url – equivalent to bz-host CLI option

BUGZILLA.constant_list – list of bug states which should be not skipped

3 TCMS Plugin

Plugin provides access to TCMS site. TODO add description

CLI Options

--with-tcms – Enables Bugzilla plugin.

--tcms-user TCMS_USER – User name for the TCMS site.

--tcms-url TCMS_URL – URL address for the TCMS site.

--tcms-realm TCMS_REALM – Kerberos realm for the TCMS site.

Configuration File Options

TCMS.user – equivalent to tcms-user CLI option

TCMS.keytab_files_location – path to directory contains KRB keytabs per each user

TCMS.realm – KRB realm

TCMS.kt_ext – keytab extension, default '.keytab'

TCMS.placeholder_plan_type – placeholder plan type, default 23

TCMS.tcms_url – equivalent to tcms-url CLI option

4 RHEVM Storage devices allocation

This plugin creates storage devices before the test starts and remove all these devices after the test finishes.

To enable this plugin add the following parameter to the RUN section:

auto_devices=yes

In your settings.conf file add a section called STORAGE and fill it with the following parameters:

[STORAGE]

- host_group = <host_group_name> # the only global parameter of this section.
- SECTION_NAME.parameter_prefix # sub-section name is a configuration path to the target device in the configuration file.

possible keys for nfs:

- nfs_server = <host_name>
- nfs_devices = <number_of_nfs_devices>

possible keys for export:

- export_server = <host_name>
- export_devices = <number_of_export_devices>

possible keys for iso:

- iso_server = <host_name>
- iso_devices = <number_of_iso_devices>

possible keys for iscsi

- iscsi_server = <host_name>
- iscsi_devices = <number_of_iscsi_devices>
- devices_capacity = <devices_capacity>

possible keys for local

- local_devices = /tmp/rest_test_domain, /tmp/rest_test_domain2
- local_server = <host_name> # optional, default is first vds

Here is an example:

[STORAGE]
host_group = indigo

[[REST.data_domain]] # will replace data_domain_address and data_domain_path in REST section
nfs_server = wolf.qa.lab.tlv.redhat.com
nfs_devices = 2

[[REST.export_domain]] # will replace export_domain_address and export_domain_path in REST section
export_server = wolf.qa.lab.tlv.redhat.com
export_devices = 2

[[REST.tests_iso_domain]] # will replace tests_iso_domain_address and tests_iso_domain_path in REST section
iso_server = wolf.qa.lab.tlv.redhat.com
iso_devices = 2

[[REST.local_domain]] # will replace local_domain_path in REST section
local_devices = /tmp/rest_test_domain, /tmp/rest_test_domain2

[[REST.lun]] # will replace lun_address, lun_target and lun in REST section
iscsi_server = tiger.qa.lab.tlv.redhat.com
iscsi_devices = 2
devices_capacity = 10

[[REST.lun1]] # will replace lun1_address, lun1_target and lun1 in REST section
iscsi_server = wolf.qa.lab.tlv.redhat.com
iscsi_devices = 2
devices_capacity = 10

5 Disabling vital tests

This plugin removes vital marks from all test cases. It means even if a vital test fails all further tests will continue running.

CLI Options

--vital-disable - Enables plugin.

6 VDSM code coverage

This plugin enables VDSM Code Coverage Reports on your VDS hosts. It requires the vdsmtests, vdsm and ruth repositories. Make sure that ruthAgent is installed on each host.TODO: add urls to repositories

CLI Options

--with-vdsm-code-coverage [VDSM_CODE_COVERAGE] – Enables the VDSM code coverage plugin.You can specify the directory for storing the results The results stored in each host uses this convention: VDS_ADDRESS.tar.gz

Configuration File Options

PARAMETERS.vds – List of VDS machine.

PARAMETERS.vds_password – List of passwords for the root account of each host.

VDSM_CODE_COVERAGE.vdsm_repo – Path to vdsm repository, must contain debugPluginClient.py

VDSM_CODE_COVERAGE.vdsmtests_repo – Path to the vdsmtests repository.

VDSM_CODE_COVERAGE.vdsm_root_path – Path to vdsm installed on VDS.

VDSM_CODE_COVERAGE.ruth_repo – Path to the ruth repository.

REST_CONNECTION.scheme – TODO

7 MAC to IP converter

Plugin captures DHCP leases on VDS hosts. It means you are able to get IP address of VM according to MAC address. ART provides function convertMacToIp which somehow maps MAC to IP itself. Plugin rebinds this function to converter, so you don't need to use two different functions and don't need to change your tests.

CLI Options

--with-mac-ip-conv - Enables plugin.

8 Host's NICs names resolution

Plugin takes defined hosts from configuration file and retrieves their NICs names. Which are stored into configuration file (PARAMETERS.host_nics).

CLI Options

--with-host-nics-resolution enable plugin

Configuration File Options

HOST_NICS_RESOLUTION.enabled – true/false enable plugin

PARAMETERS.vds – list of hosts

PARAMETERS.vds_pasword – list of root's passwords

9 Log Capturing

This plugin collects log messages when executing test cases . When a test case is finished, this plugin groups and sends logs to the related test case.

CLI Options

--log-capture - Enables plugin.

Configuration File Options

LOG_CAPTURE.enabled –Enables plugin. Set to either 'true' or 'false'.

LOG_CAPTURE.fmt – A string which describes the log message format.

11 Creating testing functions

1 API Initialization

```
from rhevm_api.utils.test_utils import get_api
ELEMENT = 'my_object_name'
COLLECTION = 'my_objects_collection_name'
my_api = get_api(ELEMENT, COLLECTION)
```

2 Fetching REST Objects

There are 2 possible ways:

using getDS:

```
from core_api.apis_utils import getDS
objectName = getDS('ObjectName')
```

directly:

```
from core_api.apis_utils import data_st
objectName = data_st.ObjectName
```

3 Object creation example:

```
def addNewObject(positive, **kwargs): # path to this function should appear in actions.conf
    majorV, minorV = kwargs.pop('version').split(".")
    objVersion = Version(major=majorV, minor=minorV)
    newObject = objectName(version=objVersion, **kwargs) # build new object
    obj, status = my_api .create(newObject, positive) # call for POST method and send new object
   return status  # must return status for test reports
```

4 Object update example:

```
def updateObject(positive, object_name, **kwargs):
    objForUpdate = my_api.find(object_name)
    newObject = objectName()
    if 'name' in kwargs:
        newObject.set_name(kwargs.pop('name'))
    if 'description' in kwargs:
        newObject.set_description(kwargs.pop('description'))
    newObject, status = my_api.update(objForUpdate, newObject, positive)
    return status
```

5 Object delete example:

```
def removeObject(positive, object_name):
        obj = my_api.find(object_name)
        return my_api.delete(dc, positive)
```

6 Get element from an element collection:

```
def getObjectFromOtherObjectCollection(parent_obj_name, get_obj_name):
    objAPI = get_api(object_type, object_collection_name)
    parentObj = objAPI.find(parent_obj_name)
    return my_api.getElemFromElemColl(parentObj, get_obj_name)
```

7 Get element sub-collection:

```
def getObjCollection(obj_name, collection_name, collection_elem_name):
        object = my_api.find(obj_name)
        return util.getElemFromLink(object, link_name=collection_name, attr=collection_elem_name,       get_href=True)
```

8 Add element to an element sub-collection:

```
def addElementToObjCollection(positive, parent_obj_name, add_object_name):
    parentObjColl = getObjCollection(parent_obj_name, collection_name, collection_elem_name)
    addObj = my_api.find(add_object_name)
    obj, status = my_api.create(addObj, positive, collection=parentObjColl)
    return status
```