

# OAR TESTSUITES DOCUMENTATION

## Contents

- What are we testing and How?
- RSpec Ruby Test Framework
- Regular RSpec Testing Method
- The Kameleon way of Testing: Idea
- Testsuite Installation
- The Testsuite Engine Design
- Advantages
- Sample Test Recipe snippet
- Schema
- List of rspectests\_lib methods
- List of oarrestapi\_lib methods
- Conclusion

## What are we testing and How?

The main aim of this project is to test the available OAR REST APIs to ensure that they are working as expected without any broken links while calling the APIs for implementing various functionalities.

A list of available OAR REST APIs is present in the Documentation which can be pulled from [http://wiki-oar.imag.fr/index.php/OAR\\_API](http://wiki-oar.imag.fr/index.php/OAR_API)

Both **unit testing** as well and **functional(Scenario)** testing will be done on the OAR REST APIs. So, we currently make available 2 ways of testing the APIs. They are as follows:

1. **Regular RSpec Testing**
2. **Testing using Kameleon Tool, IRB and RSpec**

So users of the APIs who are familiar with RSpec Test framework can use the first method of testing, while those developers who are fond of Kameleon tool and love to write recipes and steps can pick up the second option for testing the APIs.

In this Wiki, we cover the basics of how testcases can be written, by using both the methods mentioned above.

Before launching onto the 2 testing methods, let us understand about the RSpec Ruby Testing Framework which is common in both the methodologies. We cover that next.

## RSpec Ruby Test Framework

**Rspec** is a behavior driven test development framework developed for Ruby.

Behavior Driven Development(**BDD**) began its journey as an attempt to better understand and explain the process of Test Driven Development.

Behavior driven development is an agile software development technique .

RSpec uses simple plain English like statements to write test cases.

In Testsuites project, we use RSpec to carry out **unit/scenario testing** of the various OAR REST APIs.

**RSpec Documentation** with examples of simple scenario testing of OAR APIs can be pulled from [gforge scm - branch/gsoc2010/tests/testsuite/Documentation](https://github.com/gforge/scm-branch/gsoc2010/tests/testsuite/Documentation).

## Regular RSpec Testing Method

This is the most easiest and quickest way to test the OAR APIs.

Once testing using RSpec framework is clear, one can write very efficient testcases (testelements) for scenarios/unit functioning of APIs in a few minutes and run them on the fly using the **spec** command.

A Testsuite can be made by placing all the test files (must end with `_spec.rb`) in a spec folder.

By running the command: **spec path\_to\_spec\_folder --format specdoc**, each test spec file in the directory will get executed one by one displaying the result for each test file after it is run.

The RSpec Documentation I made, clarifies how this method of testing can be done with examples. But however for the sake of completeness, I will describe a small example of testing of APIs using simple RSpec testelements.

Following are testelements to submit a job successfully and check the queue if job has been submitted. Note: `jobid` is made global so that all testelements can test APIs that require this `jobid`. Also `submit_job` and `full_job_details` are methods of library `oarrestapi` that calls the POST/GET APIs which are called using the object of the library class. Read RSpec Documentation for more details. Assume the elements are from file `testspec.rb`.

```
#Submitting a job
it "should submit a job successfully " do
  jhash = { 'resource' => "/nodes=1/core=1" , 'script' => "ls;pwd;whoami;sleep
60" }
  begin
    @obj.submit_job(jhash)
    $jobid = @obj.jobstatus['id'].to_s
  rescue
    puts "#{!}"
    exit
  end
  $!.should == "" #Usually returns NULL or empty string
  @obj.jobstatus['status'].to_s.should == "submitted"
end

#Checking the queue (Can use GET /jobs to check) immediately.
it "should contain jobid in queue of created job" do
  begin
    @obj.full_job_details
  rescue
    puts "#{!}"
    exit
  end
  @obj.jobarray['items'] do |value|
    if value['job_id'] == $jobid
      @c=1
    end
  end
  $!.should == nil
  @c.should == 1
end
```

The testelement passes only if each `should` or `should_not` methods inside it are succeeding. If one fails, the testelement fails.

Running using `spec` : **\$ spec testspec.rb --format specdoc**

*OarApis*

- should submit a job successfully
- should contain jobid in queue of created job

Finished in 03.062151 seconds

2 examples, 0 failures

Thus tests can be written and run easily as shown above.

The next method is for Kameleon fans and is described elaborately in the sections that follow.

## The Kameleon way of Testing: Idea

Main Tools used in this Idea:

- **Kameleon**
- **Interactive Ruby (IRB)**
- **RSpec Ruby Gem - A Ruby BDD Testing framework**

Tests will be written in simple yaml file format. Kameleon engine can be used to parse the yaml test files.

Recipe Test file will contain macrosteps and microsteps.

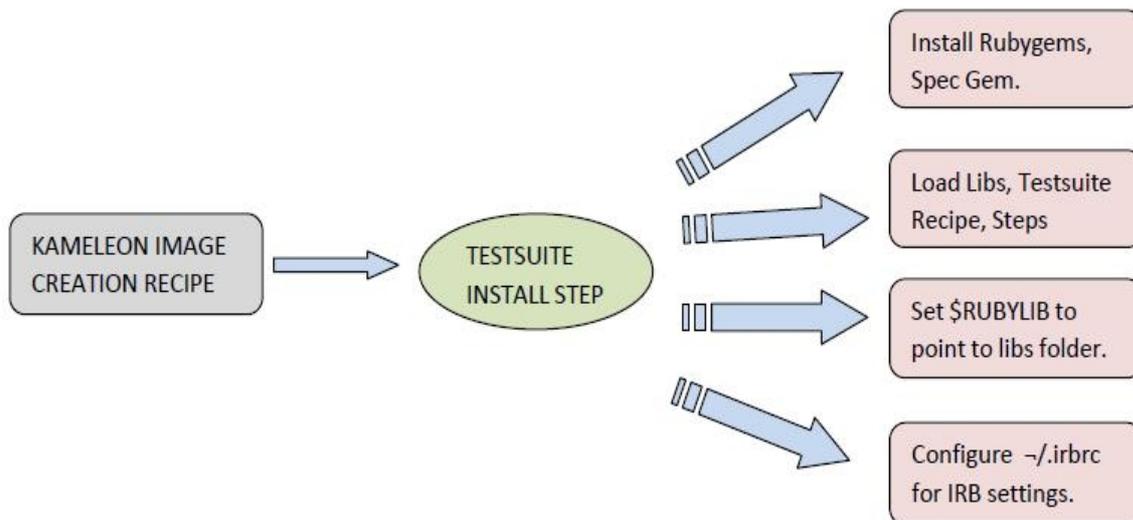
The actual testing is done using the RSpec Testing Framework internally.

An IRB Session is opened up in one of the microsteps. Named FIFO pipes are used for giving input and collecting back output/error to and from the IRB respectively. A library of RSpec test elements of the REST OAR APIs along with few scenario testing are made. This library is loaded in IRB in one of the microsteps and testings mapped with the corresponding macrosteps are run in IRB.

Ruby commands to load RSpec test library and to run the tests are given as input to the IRB via the named FIFO pipe. Similarly, output/errors of these tests run in IRB are collected back in other named pipes.

## Testsuite Installation

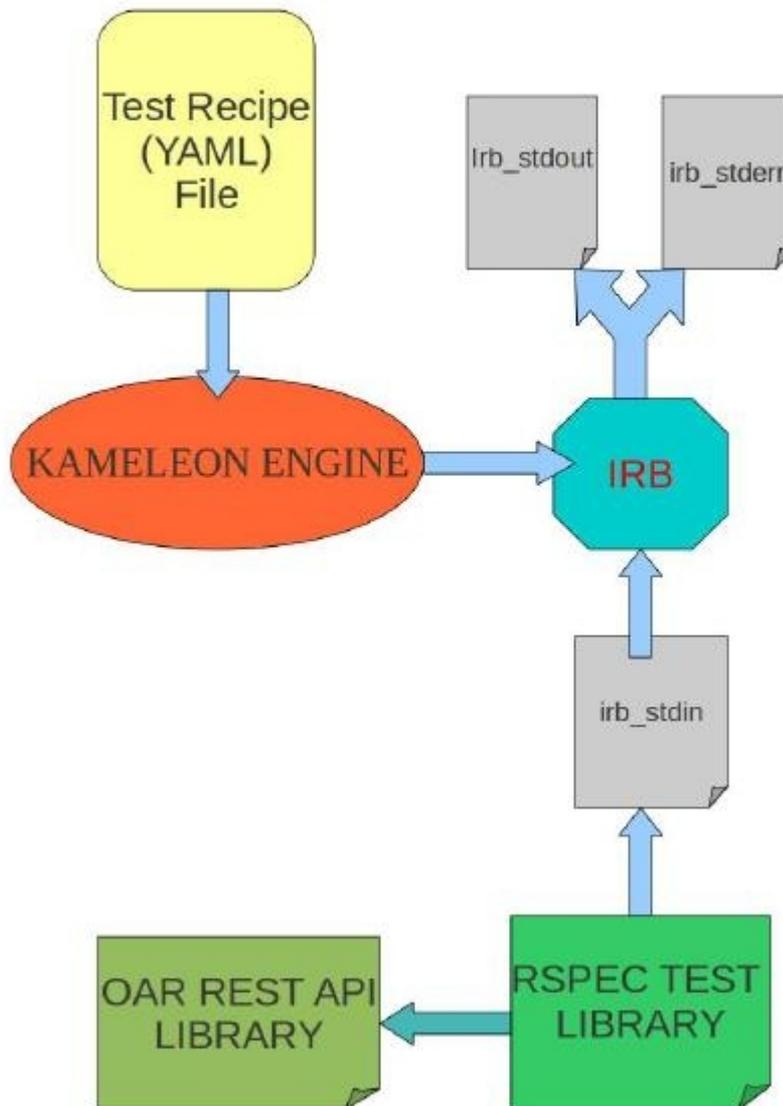
### TESTSUITE INSTALLATION IN APPLIANCE



## The Testsuite Engine Design

An overall tentative architecture of the testsuite has been drawn and committed to the Inria SVN Repository ([gsoc2010/tests/testsuite/Design](https://svn.inria.fr/svnroot/gsoc2010/tests/testsuite/Design)).

## TESTSUITE ENGINE DESIGN



### Advantages

- Test files can be easily written as it is in the standard YAML format.
- Usage of Kameleon engine will make parsing of test files quicker, powerful and efficient. All features of kameleon is derived here.
- Inner test cases made using RSpec, which is easier to learn, code; takes just few minutes to add a new spec test element to the rspec test library.
- Since each test is carried out in IRB through kameleon, debugging is easy at each step.

### Sample Test Recipe snippet

A sample steps of test recipe will tentatively look like:

**steps:**

- ```
- irb #Initializes IRB
- load_lib_create_object #Load rspec tests_lib and creates object of
```

```

the lib
- unit_tests:
  - test_submit_job      #Test post /jobs REST API submitted job successfully
  - test_get_job         #Test get /jobs API is getting details of job
  - test_delete_job      #Test delete /jobs API is fine
  - test_get_version     #Test GET /version is working fine
  - test_get_timezone    #Test GET /timezone is working fine
#
# Scenario Testing
#Test Scenario 1: Submit a job, check if submitted, Delete job, Check if deleted
- scenario1:
  - submit_job
  - test_if_submitted
  - delete_job
  - test_if_deleted
#- print_output
#Only 1 print_output should be there in a recipe. It should be put only after
all the tests/scenarios
#
#Test Scenario 2: Test to Submit job, Hold job, Resume job
- scenario2:
  - submit_job_running
  - test_if_submitted
  - hold_job_running
  - test_if_job_held
  - resume_job
  - test_if_job_resumed
- print_output          #Print output of tests

```

The irb.yaml macrostep:

**irb:**

```

- create_pipes:
  #Named Fifo pipe to store the output produced after testing in IRB
  - exec_appliance: mkfifo irb_stdout
  #Named Fifo pipe which accepts input for IRB
  - exec_appliance: mkfifo irb_stdin
  #Named Fifo pipe to log the errors produced after testing in IRB
  - exec_appliance: mkfifo irb_stderr
- start_irb:
  - exec_appliance: irb <irb_stdin >irb_stdout &
  - exec_appliance: IRB_PID=$!
- bash_function:
  - exec_appliance: function write_irb(){ echo "$1" > irb_stdin; }
- clean:
  - exec_appliance: echo "kill $IRB_PID" >> $$workdir/clean.sh
  - exec_appliance: echo "rm -f $$workdir/chroot/irb_stdout" >> $
$workdir/clean.sh
  - exec_appliance: echo "rm -f $$workdir/chroot/irb_stdin" >> $
$workdir/clean.sh
  - exec_appliance: echo "rm -f $$workdir/chroot/irb_stderr" >> $
$workdir/clean.sh

```

This macrostep creates named pipes for input, output and error and initializes a session of IRB. The clean step adds rm -f commands that delete pipes into clean.sh

The **load\_lib\_create\_object** step contains the following microsteps:

**load\_lib\_create\_object:**

```

- load_lib_create_object:

```

- load\_libs:
  - exec\_appliance: write\_irb "require '\$\$rpsec\_lib\_path'"
- create\_objects:
  - exec\_appliance: write\_irb "obj = Tests.new(\$\$apiuri)"

Here, the microsteps uses 2 Global variables declared in the Kameleon recipe file which are referenced using \$\$ .They are:

RSpec Library Path and Apiuri Path.

rpsec\_lib\_path: /home/kameleon/lib/rspectests\_lib

apiuri: "<http://kameleon:kameleon@localhost/oarapi-priv>"

The **test\_submit\_job** macrostep will contain the following microsteps:

#### test\_submit\_job:

- test\_submit\_job:
  - exec\_appliance: write\_irb "jhash = { 'resource' => \"/nodes=1/core=1\" , 'script' => \\"ls;pwd;whoami\" }"
  - exec\_appliance: write\_irb "obj.test\_submit\_job(jhash)"

Here, the load\_lib\_create\_object step imports the RSpec Test library file and creates an object of RSpec class, Test. It uses this object to explicitly call the test\_submit\_job() ruby method which contains the RSpec test element and runs the test element separately.

The **scenario1** macrostep will contain the following microsteps:

#### scenario1:

- submit\_job:
  - exec\_appliance: write\_irb "jhash = { 'resource' => \"/nodes=1/core=1\" , 'script' => \\"ls;pwd;whoami;sleep 60\" }"
- test\_if\_submitted:
  - exec\_appliance: write\_irb "obj.test\_submit\_job(jhash)"
  - exec\_appliance: write\_irb "jid = obj.jobid"
  - exec\_appliance: write\_irb "puts \"Submitted job \"+jid.to\_s"
  - exec\_appliance: write\_irb "obj.test\_job\_in\_queue(jid)"
- delete\_job:
  - exec\_appliance: write\_irb "puts \"Deleting job \"+jid.to\_s"
  - exec\_appliance: write\_irb "puts jid.to\_s"
- test\_if\_deleted:
  - exec\_appliance: write\_irb "obj.test\_jobs\_delete(jid)"
  - exec\_appliance: write\_irb "obj.test\_job\_notin\_queue(jid)"
  - exec\_appliance: write\_irb "puts \"Scenario1 successfully tested \""

The **print\_output** macrostep contains the following microsteps:

#### print\_output:

- print:
  - exec\_appliance: write\_irb "exit"
  - exec\_appliance: sleep 1
  - exec\_appliance: cat irb\_stdout|tee output\_file
  - exec\_appliance: echo "Results of the Testsuite Run"
  - exec\_appliance: grep -C 1 "OarApis" output\_file
  - exec\_appliance: grep failure output\_file>numexamples
  - exec\_appliance: cat numexamples|wc -l|sed 's/^/Total Number of examples run:/'
  - exec\_appliance: awk -F '[ ,]' '{static s; if(\$4 == 0) s++;} END { print s }'
  - numexamples|sed 's/^/Number of Successes:/'
  - exec\_appliance: echo "Time taken to run each tests"
  - exec\_appliance: grep seconds output\_file|tee testtime
  - exec\_appliance: echo "Total Time taken:"
  - exec\_appliance: awk '{ static timesum; timesum+=\$3 }; END { print timesum }'

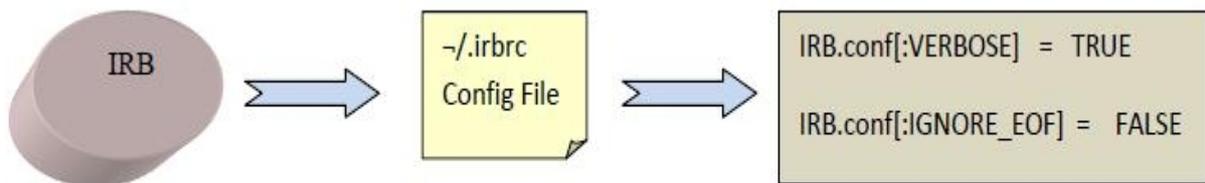
testtime

Only 1 print\_output microstep should be present in the recipe file. If 2 scenario testings needs to be done, place print\_output step after 2nd scenario. This will contain the output of tests of both the scenarios.

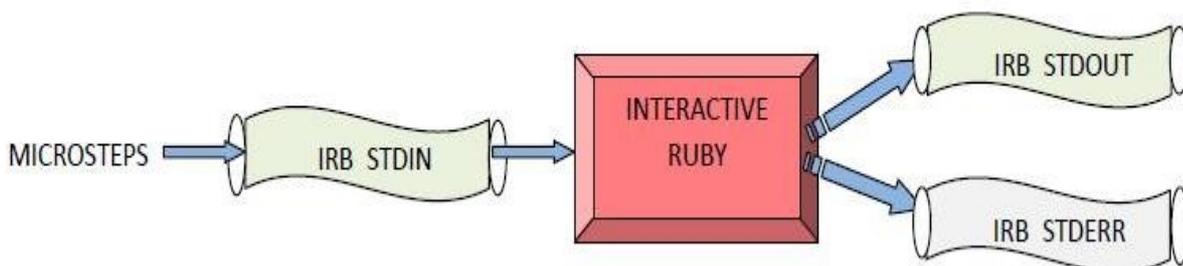
Notice here that print\_output microstep is rather big. The intention was to add more creativity in the output after the test run. It can also be rather small and simple; depends on the requirements of the tester.

## Schema

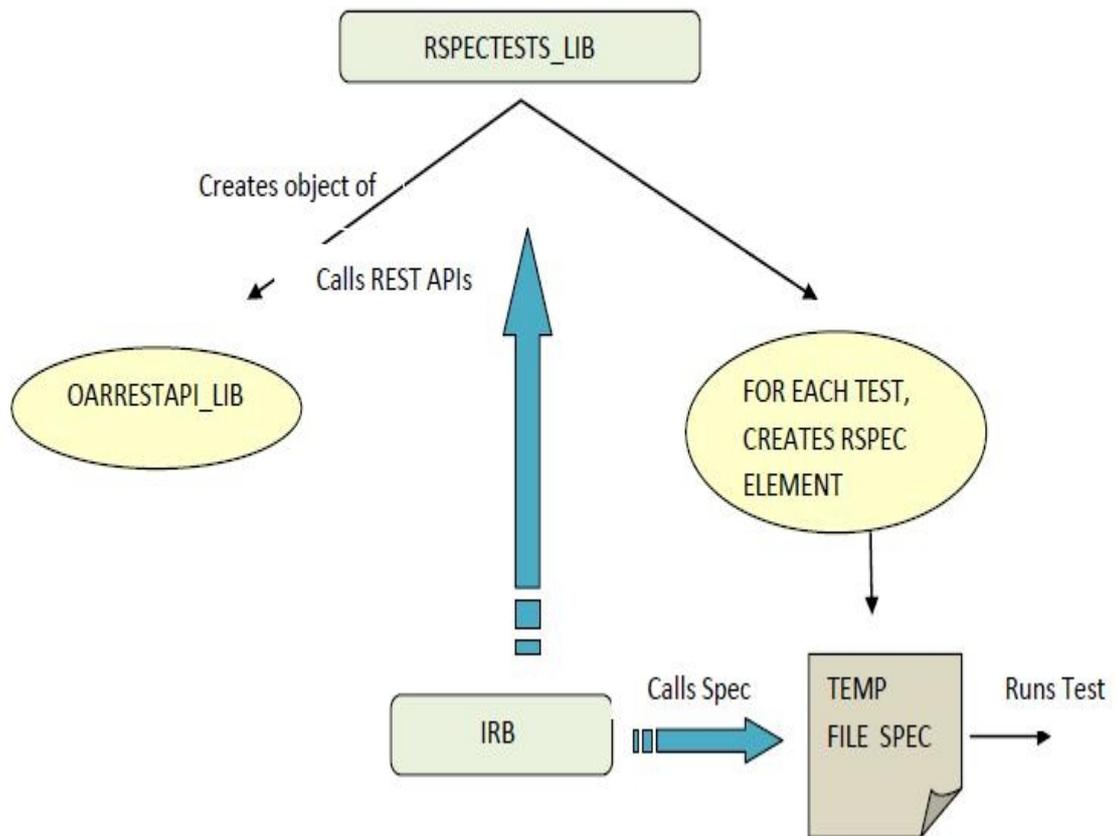
### IRB INITIALIZATION CONFIGURATION SETTINGS



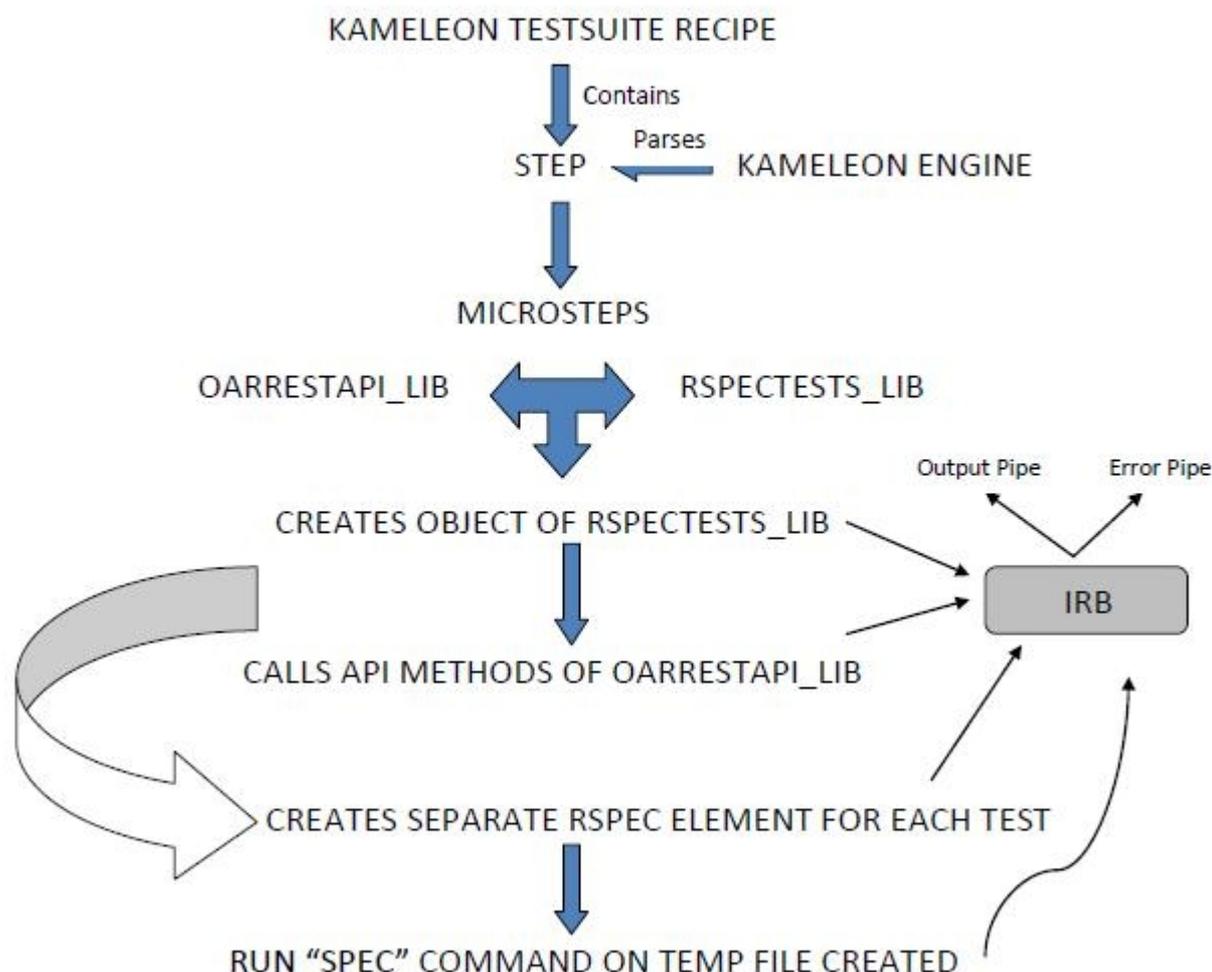
### IRB INPUT/OUTPUT



## RSPECTESTS\_LIB CONTROL FLOW



## OVERALL\_FLOW



## List of rspectests\_lib methods

The **rspectests\_lib** currently contains the following methods/rspec test elements. As and when more lib methods are required, it can be easily added to the library. The OAR REST APIs are called by this library using the **oarrestapi\_lib** library. For writing tests, one has the flexibility to require either **rspectests\_lib** or **oarrestapi\_lib** based on their specific requirements.

### Class: Test

#### Methods:

- |                                   |                                          |
|-----------------------------------|------------------------------------------|
| 1. test_get_version               | - Testing the GET /version REST API      |
| 2. test_get_timezone              | - Testing the GET /timezone REST API     |
| 3. test_get_jobs_details          | - Testing the GET /jobs/details REST API |
| 4. test_get_running_jobs          | - Testing the GET /jobs REST API         |
| 5. test_get_jobs_id (jid)         | - Testing the GET /jobs/<ID> REST API    |
| 6. test_job_in_queue(jid)         | - Test to check if job is there in queue |
| GET /jobs/details API             |                                          |
| 7. test_job_notin_queue(jid)      | - Test to check if job is deleted from   |
| queue using GET /jobs/details API |                                          |

- 8. test\_get\_jobs\_table - Testing the GET /jobs/table REST API
- 9. test\_submit\_job (jhash) - Testing the POST /jobs REST API
- 10. test\_jobs\_delete\_post (jid) REST API - Testing the POST /jobs/id/deletions/new REST API
- 11. test\_jobs\_delete (jid) - Testing the DELETE /jobs/<id> REST API
- 12. test\_get\_resources - Testing the GET /resources REST API
- 13. test\_get\_resources\_full - Testing the GET /resources/full REST API
- 14. test\_job\_rholds (jid) REST API - Testing the POST /jobs/<id>/rholds/new REST API
- 15. test\_job\_hold (jid) REST API - Testing the POST /jobs/<jobid>/holds/new REST API
- 16. test\_job\_resumption (jid) REST API - Testing the POST /jobs/<id>/resumption/new REST API
- 17. test\_job\_update (jid, actionhash) - Testing POST /jobs/<id>/ API (deleting use when browsers dont support DELETE)
- 18. test\_if\_job\_delete\_updated (jid) - Testing POST /jobs/<id>/ to see if job is updated with actionhash, Call this after calling above method
- 19. test\_job\_checkpoint (jid) REST API - Testing the POST /jobs/<jobid>/checkpoints/new REST API
- 20. test\_job\_running (jid) - Testing if job is currently running

## List of oarrestapi\_lib methods

The rspectests\_lib call the OAR REST APIs through the **oarrestapi\_lib** library.

### Class: OarApi

#### Methods:

- 1. get(api,uri) - GET REST OAR API; Function to get objects from the api
- 2. post(api,uri,j) - POST REST OAR API; Function to create/delete/hold/resume objects through the api
- 3. delete(api, uri) - DELETE REST OAR API; Function to Delete objects through the api
- 4. oar\_version - Gives version info & Timezone about OAR and OAR API/Server.
- 5. oar\_timezone - Gives the timezone of the OAR API server.
- 6. full\_job\_details - List the current jobs & some details like assigned resources
- 7. run\_job\_details - List currently running jobs
- 8. specific\_job\_details(jobid) - Get Details of a specific job
- 9. dump\_job\_table - Dump the jobs table (only current jobs)
- 10. submit\_job(jhash) - Submits job
- 11. del\_job(jobid) - Delete job - POST /jobs/id/deletions/new
- 12. send\_checkpoint(jobid) - Send checkpoint signal to a job
- 13. hold\_waiting\_job(jobid) - Hold a Waiting job
- 14. hold\_running\_job(jobid) - Hold a Running job
- 15. resume\_hold\_job(jobid) - Resume a Holded job
- 16. send\_signal\_job(jobid, signo) - Send signal to a job with signalno.
- 17. update\_job(jobid, actionhash) - Update a job
- 18. resource\_list\_state - Get list of Resources and state
- 19. list\_resource\_details - Get list of all the resources and all their details
- 20. specific\_resource\_details(jobid) - Get details of resources identified by an ID
- 21. resource\_of\_nodes(netaddr) - Get details about the resources belonging to the node identified by network address
- 22. create\_resource(rhash) - Create Resource
- 23. statechange\_resource(jid,harray) - Change the state of resources of a job

- 24. `delete_job(jobid)` - Delete or kill a job using DELETE API
- 25. `delete_resource(resid)` - Delete the resource identified by id
- 26. `delete_resource_cpuset(node,cpuid)`- Delete the resource corresponding to cpuset id on node node.

## Conclusion

Thus, the Testsuites are very flexible and highly scalable. Any type of test cases can be thought of and coded straightaway. The Kameleon method helps in writing highly readable testcase steps , macrosteps and microsteps and the Regular Rspec Method of testing helps in writing easy testcases and scenarios quickly.

The `oarrestapi_lib` as well as `rspectests_lib` can be further enhanced as and when required introducing new test functions.

---

---

SUBMITTED BY NARAYANAN K AS PART OF GSOC-2010 TESTSUITES PROJECT

---

---