Twister framework guide

Contents

1 - What is Twister?

Twister is an **open source** test automation framework.

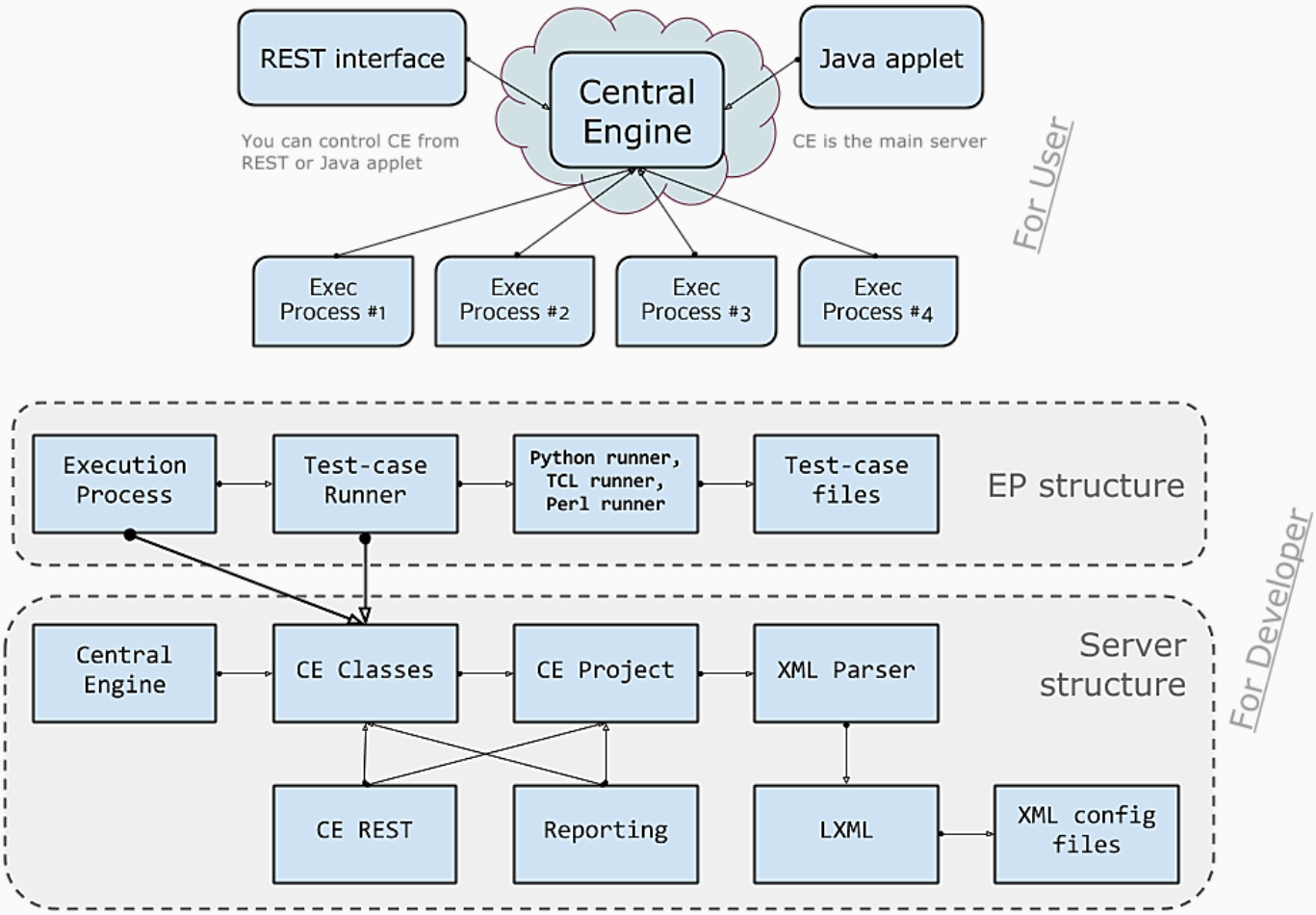
The code can be downloaded from: https://github.com/Luxoft/Twister.

Twister helps building functional, regression and load test suites. It was developed taking in account the specific needs of the enterprise telecommunication market to help in testing telecommunication devices like switches, routers or PBXs.

Key features of Twister:

- web based GUI intuitive & user friendly interface ;
- easy to manage tests/ suites/ projects ;
- multi-user architecture ;
- real time monitoring of execution ;
- distributed execution: SUT's can be tested in parallel ;
- against same or different set of tests ;
- flexible reporting mechanism: DB schema is not fixed and there is no need to change framework to fit with a new DB schema ;
- support for different scripting languages and for record & play GUI tools ;
- support for Continuous Integration, Source Revision Control, Bug tracking ;
- support for GUI/ backend plugins: specific functionality loaded dynamically ;
- OpenFlow 1.0 and 1.3 modules available for conformance testing.

Concept



2 - How to install the framework

In order to install the Twister Framework, a few requirements must be met:

- A **Linux** machine. All the services must run on **Linux** (tested on *Ubuntu* and *OpenSuse*);
- Python **2.7**. Python is installed by default, on most Linux systems; the framework is written and tested in Python 2.7;
- Python Tkinter. Required only if you need to run TCL tests. This is included by default in Python, but some Linux distributions don't have the `*python-tk*` lib, so it has to be installed with:`*sudo apt-get install python-tk*`;
- TCL Expect libraries. Required only if you need to run TCL tests with Expect. To test the functionality, open a Python 2.7 interpreter, then type:

```
from Tkinter import Tcl
t = Tcl()
t.eval('package require Expect')
# If this fails, you must install Expect from your package manager, or compile it from sources
# The sources are at: sf.net/projects/expect; download, extract, ./configure, sudo make install
exit()
```

- Perl Inline Python. This is required only if you need to run Perl scripts.

The Twister repository is located at: https://github.com/luxoft/twister.

The installer is located in the folder `*installer*` and it's also written in Python. It works only in **Linux**. When installing the server, must run as **ROOT**. When installing the client, ROOT is recommender, but not mandatory.

When installing Twister for the first time, it is STRONGLY RECOMMENDED to have an internet connection and run as **ROOT**, to allow the setup of all the dependencies, otherwise, you have to install the dependencies manually.

You might need to configure the **proxy** to access the internet. In this case, edit the file `*installer.py*`, locate the line with **`HTTP_PROXY`** and type: **`HTTP_PROXY`** `= 'http://UserName:PassWord@http-proxy:3128'`. If the username and password are not required for your proxy, you can omit them.

In order to serve the Java applet, you will also need ***Apache*** or ***Lighttpd*** server and ***Open-SSH*** server.

The recommended command for starting the installer:

```
sudo python2.7 installer.py
```

The *Twister Client* doesn't have any required dependencies. `*Scapy*` is optional, used only if you need to run the packet sniffer. Some tests and libraries will require additional dependencies! For example: `*paramiko*`, `*pExpect*`, `*RpcLib*`, `*Suds*`, `*Requests*`, or `*Gevent*`. If you need to run these tests or libraries, you can install `pip` (tool for installing and managing Python packages - www.pip-installer.org) and use `*sudo pip install <package>*`.

The installer will guide you through all the steps:

1. Select what you wish to install (*client* or *server*);
2. If the `*twister*` folder is already present, you are asked to backup your data in order to continue, because everything is DELETED, except for the `*config*` folder.

Twister Client will be installed in the home of your user, in the folder `*twister*`. The server will be installed by default in `/opt/twister`.

Any dependencies that are old, or missing, will be automatically downloaded and installed. If all the requirements are met, the client or server files are copied, nothing else is installed.

3 - Dependencies list

The dependencies will be installed automatically when you first install Twister, if you have a connection on the internet.

- **LXML**: (*www.lxml.de/*)
  - XML and HTML documents parser;
  - LXML is included in Ubuntu by default. The other Linux distributions must install it;

- **MySQL-python**: (*mysql-python.sourceforge.net/*)
  - Connects to MySQL databases. It is only used by the Central Engine;
  - MySQL-python requires the *python2.7-dev* headers in order to compile;

- **CherryPy**: (*www.cherrypy.org/*)
  - High performance, minimalist Python web framework;
  - CherryPy is used to serve the Central Engine, Resource Allocator and Reports;

- **Mako**: (*www.makotemplates.org/*)
  - Hyperfast and lightweight templating for the Python platform;
  - Mako is used for templating the Central Engine REST and Report pages;

- **Beaker**: (*beaker.readthedocs.org/*)
  - Library for caching and sessions, in web applications and stand-alone Python scripts;
  - Beaker is *optional*; it is used by Mako, to cache the pages for better performance;

~ Optional ~

- Scapy: (*pypi.python.org/pypi/scapy-real/*)
  - Interactive packet manipulation tool;
  - Scapy, is used by the Execution Process to capture packets and send them to the applet;

- Paramiko: (*github.com/paramiko/paramiko/*)
  - Native Python SSHv2 protocol library;
  - Paramiko is *optional*; it is used by the Twister SSH Lib to connect to remote machines;

- *pExpect*: (*sourceforge.net/projects/pexpect/*)
  - Spawn child applications, control them, respond to expected patterns in their output;
  - pExpect is *optional*; it is used by some Python test cases to connect to FTP/ Telnet;

- *RpcLib*: (*https://github.com/arskom/rpclib/*)
  - Create web services in Python (soap, rpc, rest servers);
  - RpcLib is *optional*; it is used by some Python test cases;

- *Suds*: (*https://fedorahosted.org/suds/*)
  - Lightweight SOAP python client for consuming Web Services;
  - Suds is *optional*; it is used by some Python test cases;

- *Requests*: (*http://docs.python-requests.org/*)
  - Elegant and simple HTTP library for Python, built for human beings;
  - Requests is *optional*; it is used by some Python test cases to connect to HTTP servers;

- *Gevent*: (*http://www.gevent.org/*)
  - Co-routine-based Python networking library that provides a high-level synchronous API;
  - Gevent is *optional*; it is used by some Python test cases to create sockets and threads;

4 - Twister services

Twister framework has 2 services:

1. The **Central Engine** = central server for script and library files. It includes the Resource Allocator, Service Manager and Reporting Server. Must run as **ROOT**;
2. The **Execution Process** = one or more client services, that run the script files (Python, TCL, Perl, Java). It can run as a normal user, but it's recommended to also run as **ROOT**;

The executable scripts for starting the services are located in the `*bin*` folder:

- Central Engine executable is located in `*/opt/twister/bin*`;
- Execution Process executable is located in `*/$USER_HOME/twister/bin*`.

The EP service must be configured before run. You have to edit the file `*epname.ini*` from `*twister/config/*` folder; it contains the **name** of the EP, the **IP** and the **port** of the CE instance that it will run on.

To start the services, execute one of the following commands:

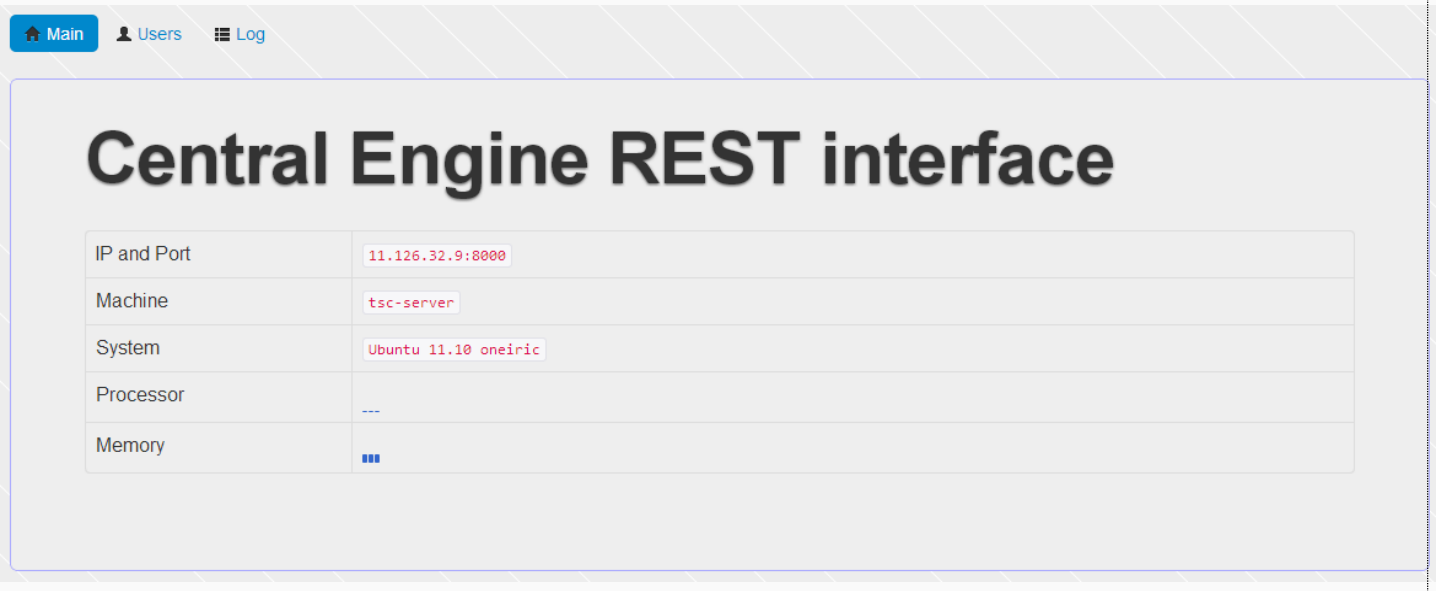
```
# Central Engine (ROOT!)
/opt/twister/bin/start_server

# For Linux Execution Process (ROOT recommended)
python /home/your_user/twister/bin/start_ep.py
```

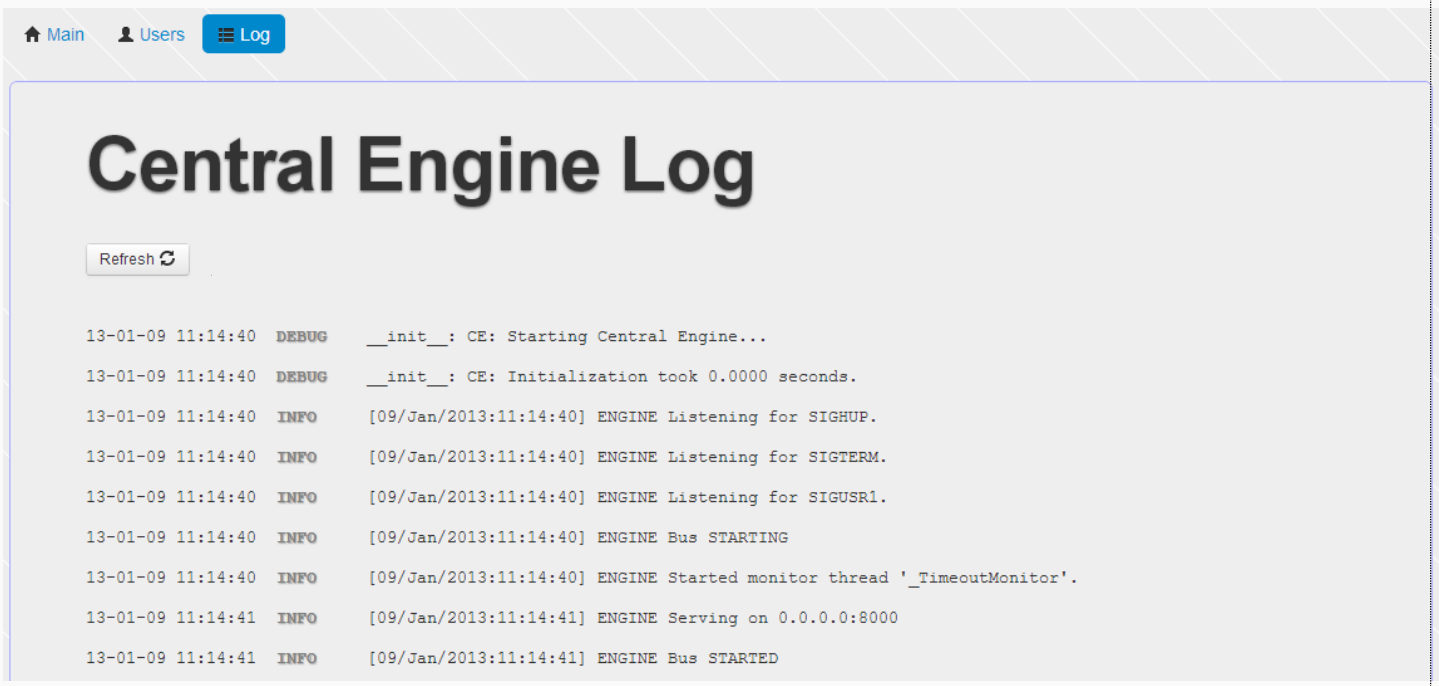
4.1 - Central Engine REST interface

While the **Central Engine** service is running, you can access a web interface that allows viewing some statistics, logs and users connected. You can also start and stop the processes.

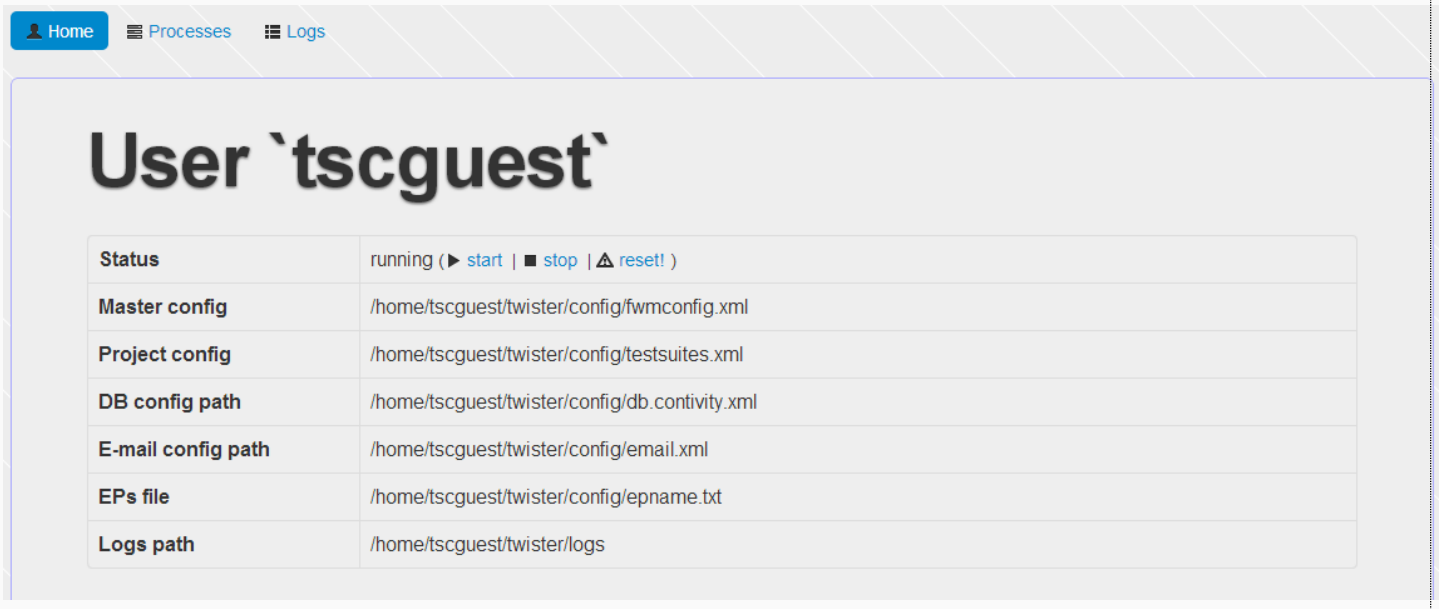
REST interface Home;



Central Engine Logs;



User interface;



Control the processes;

Check all user logs;

This web service can be accessed in a browser, by going to: `http://central-engine-IP:PORT/rest`, for **example**: `http://localhost:8000/rest`.

5 - How to compile the Java GUI

The Java Graphical User interface compiled version can be found at `*twister/**binaries**/applet*`. You have to copy the `applet` folder in `/var/www` and if you have an *Apache* or *Lighttpd* Server, you will be able to access it in the browser.

If you have changed the sources, or you want to compile the JAR files yourself, the sources are located at `*twister/client/userinterface/java*`. Some binary JAR files are already included in folders `*target*` and `*extlibs*`, respectively.

After compilation, you have to move the JAR files, so that a server can serve them.

Steps **1-2** require ***Oracle JDK 1.7*** (*Oracle Java Development Kit*).

Step **5** requires ***Apache*** *or* ***Lighttpd*** *Server* and your machine must have ***OpenSSH Server*** enabled on port 22.

Here are the steps:

1. Generate a key store, or import a certificate (*this is done only **the first time**!*);

```
PATH_TO_JDK/bin/keytool -genkey -keyalg rsa -validity 360000 -alias Twister -keypass password -storepass password
```

OR

```
PATH_TO_JDK/bin/keytool -import -alias Twister -file certificate_file.cer
```

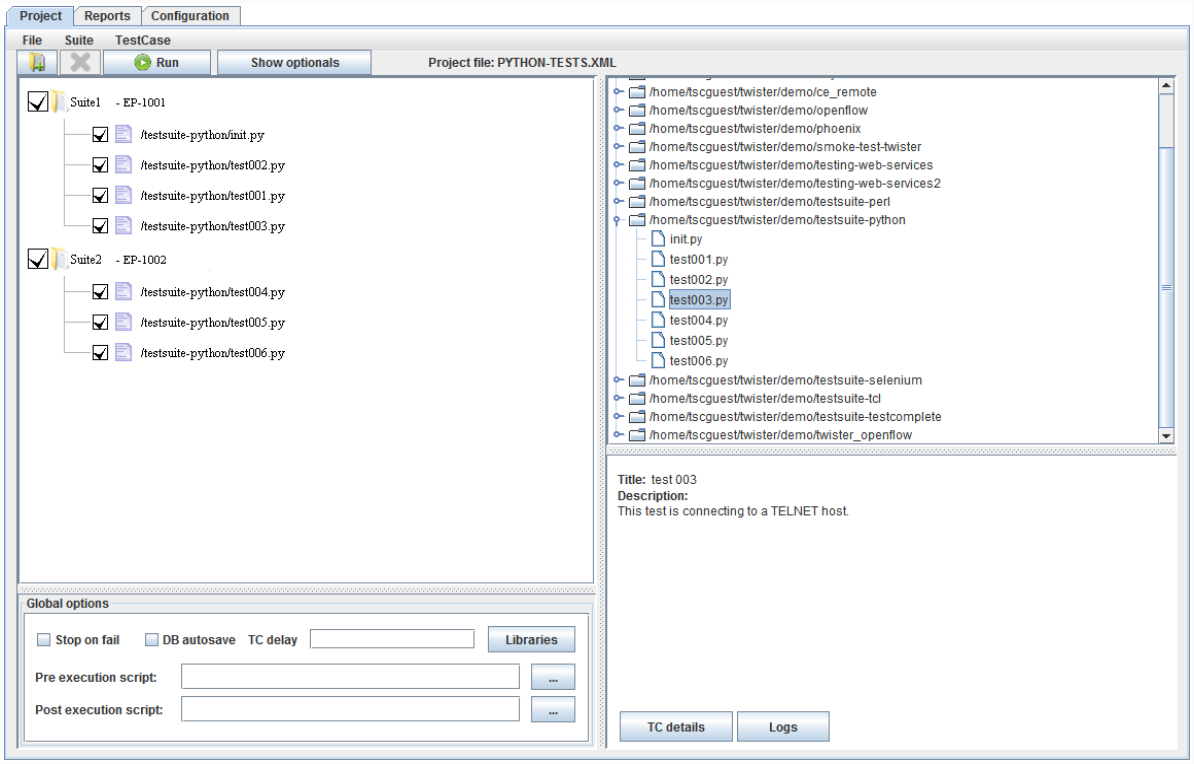
2. Go in `*client/userinterface/java*`. Then, if you are on **Windows**, run `*pack.bat*`, on **Linux** run `*./build.sh*`. You might need to edit these files, to change the path to **JDK_PATH**;
3. Move all files from `*target*` and `*extlibs*` in `*/var/www/twister*` (path for Apache, or other web servers);
4. Copy `*jquery.min.js*` from `*/opt/twister/server/static/js*` also in `*/var/www/twister*`;
5. Open a browser that supports Java Applets and go to: `http://localhost/twister`.

6 - Overview of the Java GUI

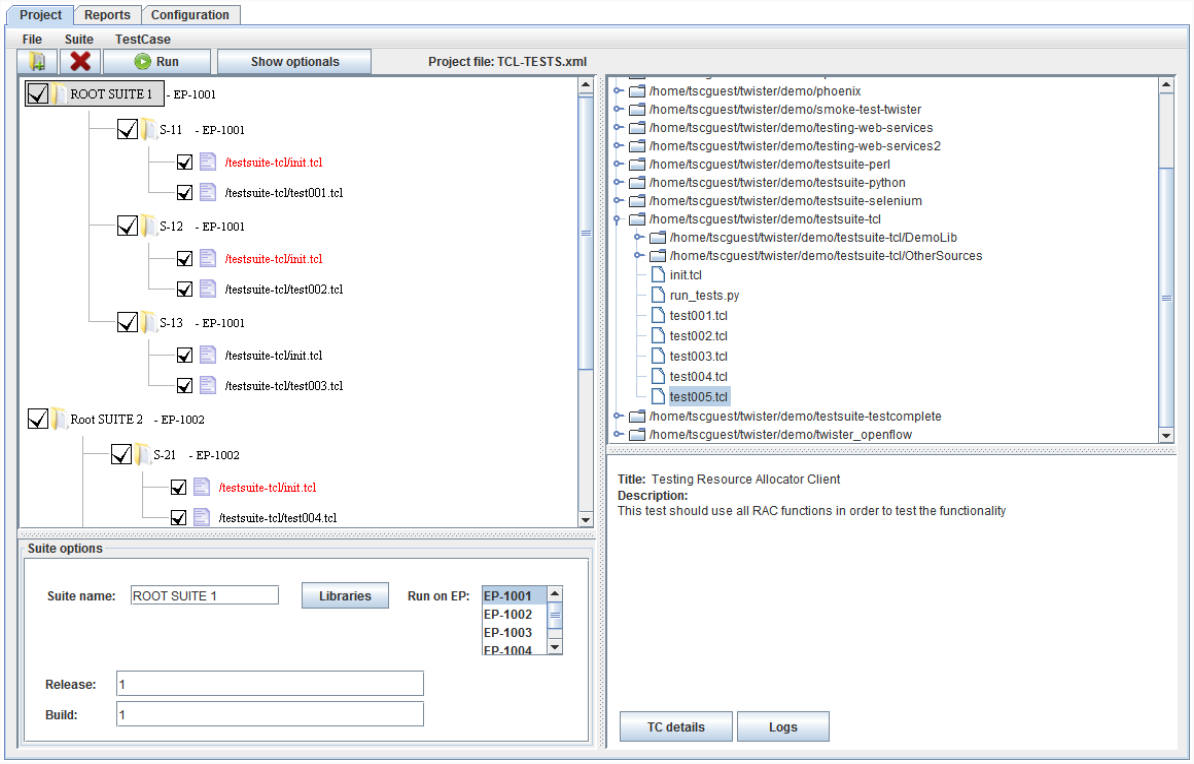
The **first tab (Suites)** is split in four panes:

- **Top left**, is where the test suites are defined. Any file from the right can be dragged in here. The files can be checked/ unchecked; the files that are not checked will not run;
- **Top right**, is where the test files are located. These files can be used in the suites;
- **Bottom left**, is where the project, suite and test information is added. The suite information is defined in the file `DB.xml`, section name `field_section` (*more about this in the configuration section*);
- **Bottom right**, you can see the title and description of the currently selected test file.

A configuration, with Python scripts:



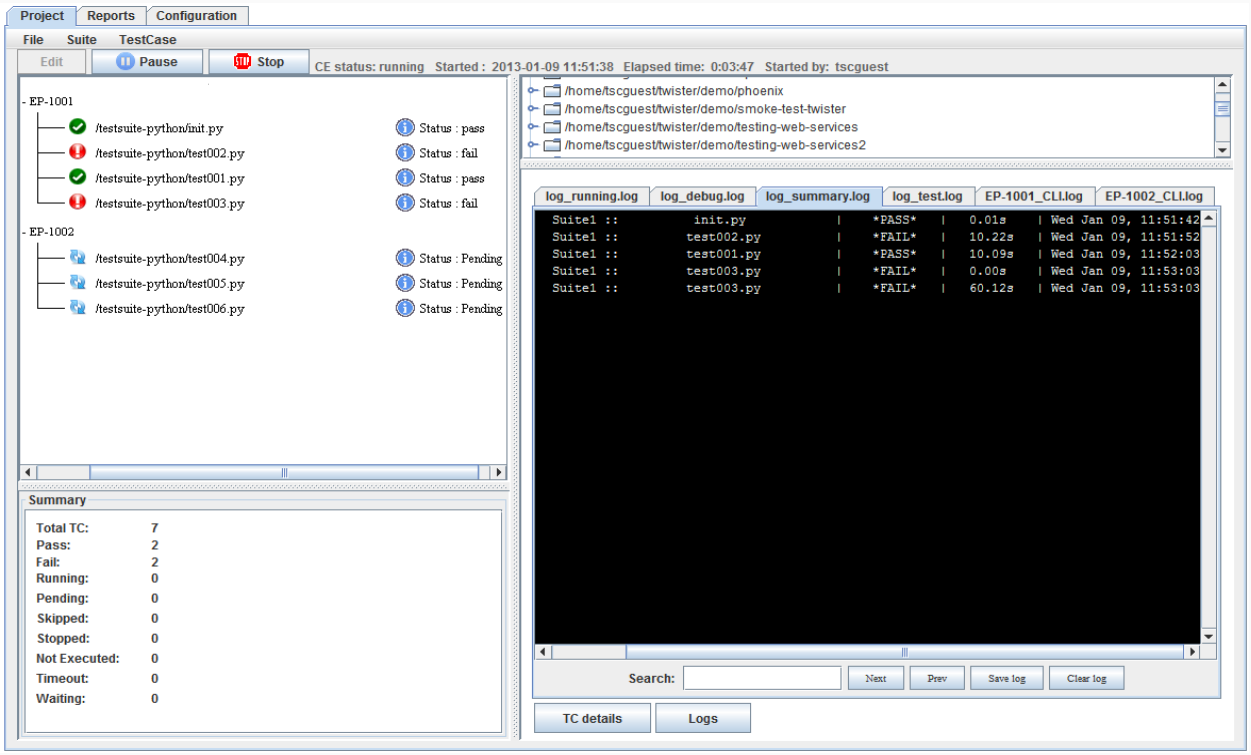
A different configuration, with TCL scripts:



While running:

- You can check test lists with their statuses. By default, all tests are in pending, unless they recently ran, in which case the most recent status is displayed;
- Logs for the tests. The logs can be cleaned, exported, or searched for keywords.

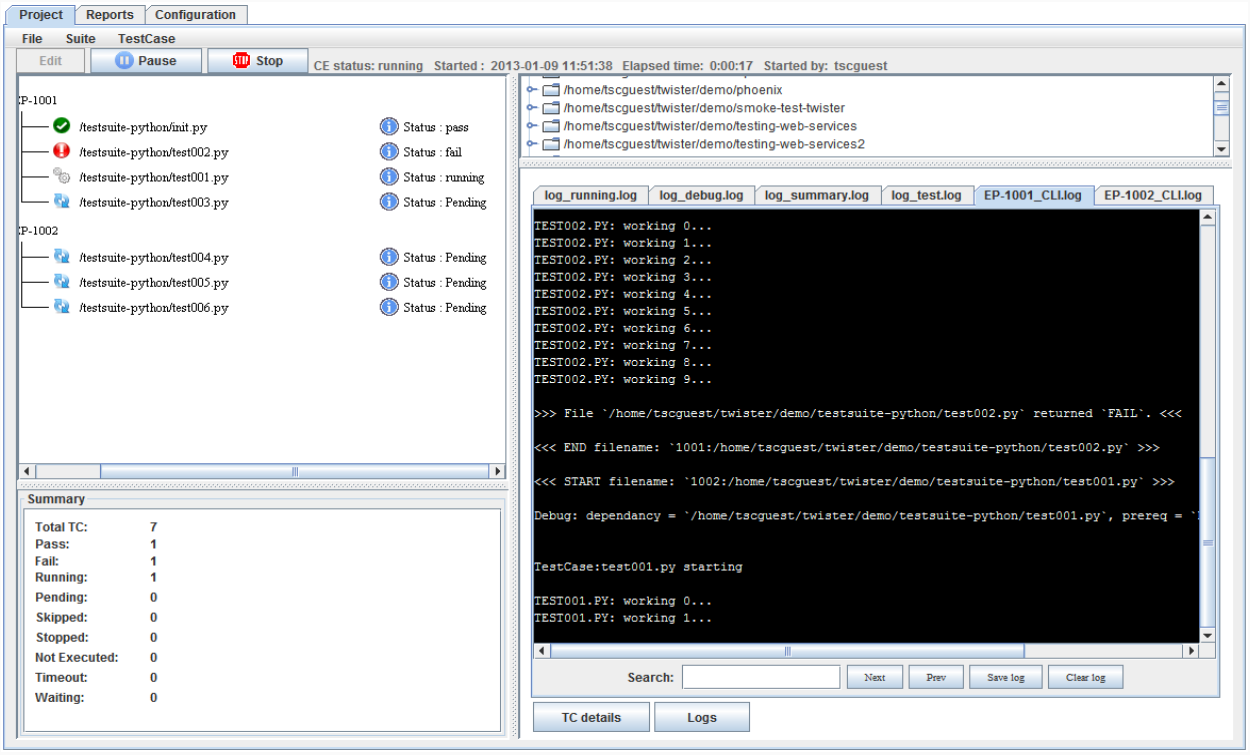
Here the Central engine is stopped; in this case you can see the most recent statuses:



At the top, there are two buttons, which control the Central Engine: **Run/ Pause** and **Stop**.

Also at the top, is the status of the Central Engine, the time of the last start, time elapsed and the user that started it.

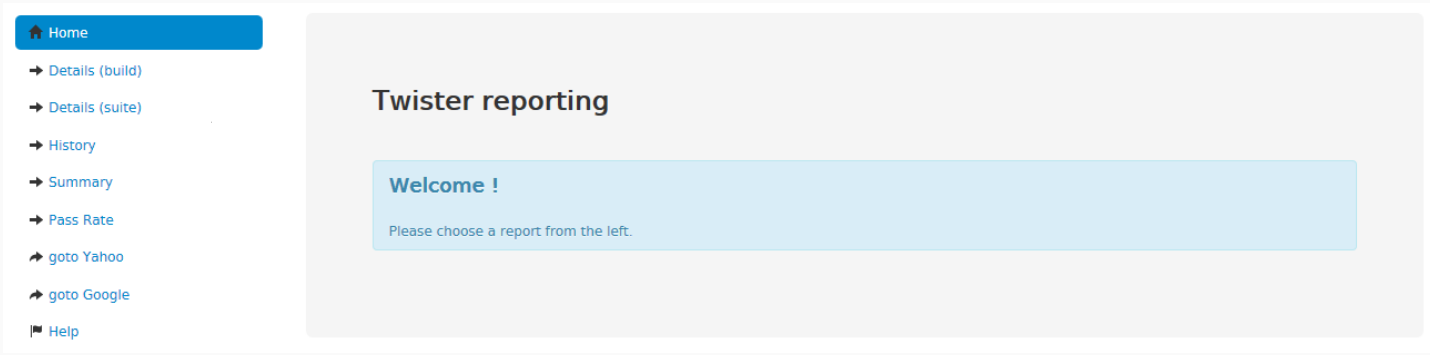
While the Central engine is running:



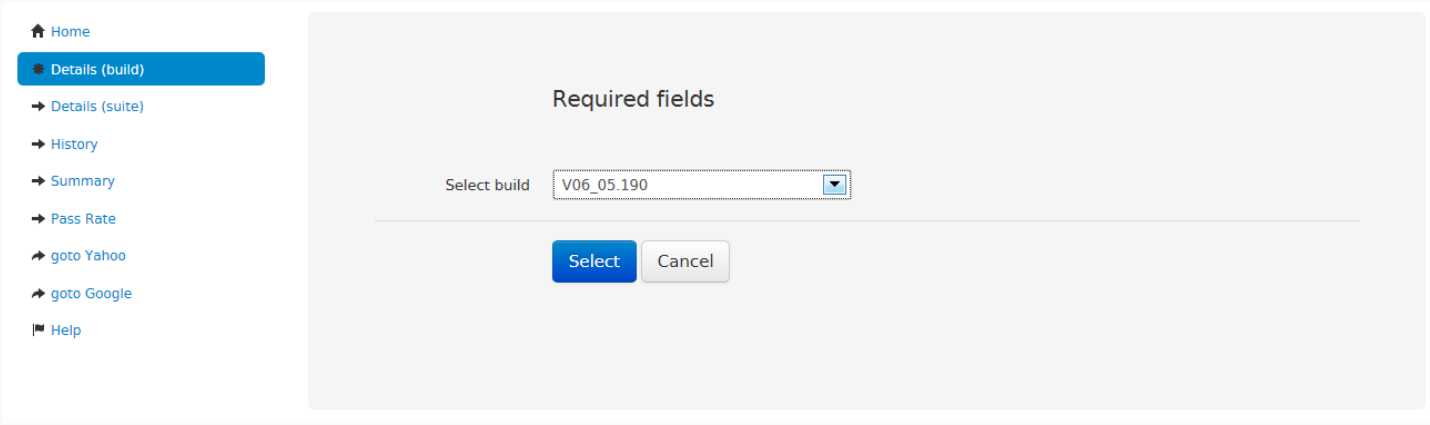
The **Reports Tab**

When clicking on it, the reports page will open in a new tab.

Reporting home (this will look different, depending on the configuration!)



A report with user chosen fields;



The same report, after the user chose the build;

The **configuration tab**

Here, you can configure:

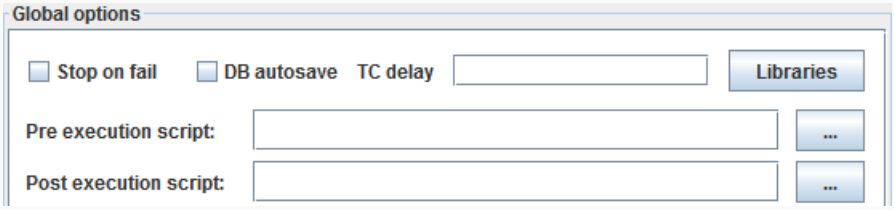
- Central Engine port (*default 8000*). This is the port the applet will use for connection;
- the path of the test files, logs files, user files;
- the path of the database xml, e-mail xml, EP names;
- the names of the log files;
- e-mail configuration, database configuration;
- devices configuration for Test Bed;
- global variables, injected in all Twister test files;
- panic-detect: checks errors in CLI logs.

More about this in `**Configure the framework**` section.

7 - How to define the suites and add tests

When starting the interface, you must first *select* or *create* a **project file**. This file will save your suites, script files and suites configurations.

Each project has a set of global settings:



- **Stop on fail** = if a test that is mandatory will fail, or will crash, all the project will fail ;
- **DB Autosave** = after all the tests from all suites finish execution, the Central Engine will automatically save the results into database, without asking the user ;
- **TC Delay** = after each test, the EP will wait X seconds, before starting to execute the next test ;
- **Pre execution script** = the script from this path will be executed by the Central Engine ***before*** running any test. The script can be written in any language (Perl, TCL, binary executable, etc.). You cannot use a normal Twister test as a Pre/ Post exec script! If you write a script in an interpreted language, don't forget to add `**#!/usr/bin/env ...***your_language*` on the first line. If the file is not executable, CE will automatically run `*chmod +x*` on the file ;
- **Post execution script** = the script from this path will be executed by the Central Engine ***after*** running all tests. It has the same specifications as the Pre execution script ;

After setting the project file, click on 

(Add suite). The required fields are: **the name** of the suite and **one or Test beds** (the workstation(s) where the tests from this suite will run).

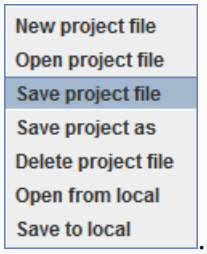
A suite is basically a folder, where one or more script/ test files can be added, that will be executed by one, or more Execution Processes.

Each suite can also have some properties attached to it, like `*release*`, `*build*`, `*comments*`, etc. These fields are defined in `DB.xml` file (*more about this in the database configuration section*) and will look different, depending on the configuration (text box, drop down list, path to a script, etc.) ; these `*meta*` properties are used while saving the results into database.

The script/ test files and the suites can be re-arranged anytime, using drag & drop, or can be deleted.

Each script/ test file has a few properties too:

- A test that is not *Runnable* will be sent to **EP**, but will never execute. You can use this to transfer configuration files on the EP ;
- *Optional* tests that fail, will not stop the EP when *Stop on Fail* checkbox is active ;
- *Setup* tests are always mandatory and if they *fail*, the entire Suite is considered *Failed*. Setup files are NOT saved into the database!

In order to save the **project file**, use the *File menu :* 

.

You can download the project file locally, to share it with your team members.

8 - How to run the test files

After all the suites and scripts are set, click on 

, to start the execution.

What will happen is that, all **checked** scripts from all suites will run, in order. The execution doesn't stop if one of the scripts fails, excepting the case when the test that fails is *mandatory* and the *Stop on Fail* checkbox is checked.

If the **Run** button is disabled, it means that the Central Engine service is not running, so the execution cannot start.
If the **Run** button is enabled, you can start the execution. At the same time, the **Stop** button will activate, allowing to stop the Central Engine and kill all the running processes and the **Run** button will become **Pause**, allowing to pause after the current tests finishes its execution.

If the Central Engine was started recently, by default all the files will be in state *pending* .
If a previous run was completed, the most recent status is displayed (*pass, failed, etc.*).

The states for the files and their respective icons are:

- (running) while the file is running;
- (pause) if the test is paused;
- (success) if the execution was successful;
- (failure) if the execution failed;
- (skip) if the file was marked as skip (*runnable=false*);
- (abort) if the file was stopped while running;
- (not executed) if the file was paused and was stopped instead of being resumed;
- (dependency) is the file depends on another file and the dependency didn't finish its execution, so this file is waiting.

While the tests are running, the logs from the left will update, showing the live output.

When a test is completed, the icon will change to Pass or Fail. All the history of result can be seen in the `*log_summary.log*`.

The logs can be cleaned, exported, or searched for keywords, by clicking the buttons from the bottom.

After all the tests are run, if the e-mail is configured, CE sends an e-mail with the report and then, all tests are saved into the database, excepting the *Setup* and *Teardown* files.

9.0 - Command line interface

The Command-line script can be found in twister/bin, both on server and client side.

This script can be used to:

- start, stop, pause the execution;
- show all users that are running tests;
- display what EPs are enabled for your user;
- show what is the start time for this run, suites list and tests list;
- show execution summary status: how many test cases are planned for execution, how many were executed, how may passed, how many failed;
- show execution details status: the same, plus status per test case;
- queue tests during run time.

You can use `./cli.py --help` anytime, to see the usage information.

```
tscguest@tsc-server:/home/tscguest/twister/bin$ python cli.py --help
Usage: cli.py --server <ip:port> --command [...parameters]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --server=SERVER       Central engine server IP and Port.
  -u, --users           Show active and inactive users.
  --eps                 Show active and inactive Eps.
  --stats               Show stats.
  --status              Show stats.
  --details             Show detailed status for All files.
  --status-details=STATUS_DETAILS
                        Show detailed status for running, finished, pending,
                        or all files.
  -q QUEUE, --queue=QUEUE
                        Queue a file at the end of a suite. Specify queue like
                        `suite:file`.
  -s SET, --set=SET     Set status: start/ stop/ pause. (Must also specify a
                        config and a project)
  -c CONFIG, --config=CONFIG
                        Path to FWMCONFIG.XML file.
  -p PROJECT, --project=PROJECT
                        Path to PROJECT.XML file.
```

Commands:

- **./cli.py –u** - Show active and inactive users ;
- **./cli.py –eps** - Show active and inactive Eps ;
- **./cli.py –stats** - Show minimal stats ;
- **./cli.py –details** - Show detailed stats, per ep + suite + test ;
- **./cli.py --status-details** <running| finished| pending| all> ;
- **./cli.py --queue** Suite1:testsuite-python/test_py_resources.py
  - Queue a file at the end of a suite. Must specify queue in the form of `suite:file_path` ;
- **./cli.py -s stop** | **./cli.py -s start** -p ~/twister/config/testsuites.xml --config ~/twister/config/fwmconfig.xml
  - start, pause, or stop the central engine.

10.0 - Twister configuration files

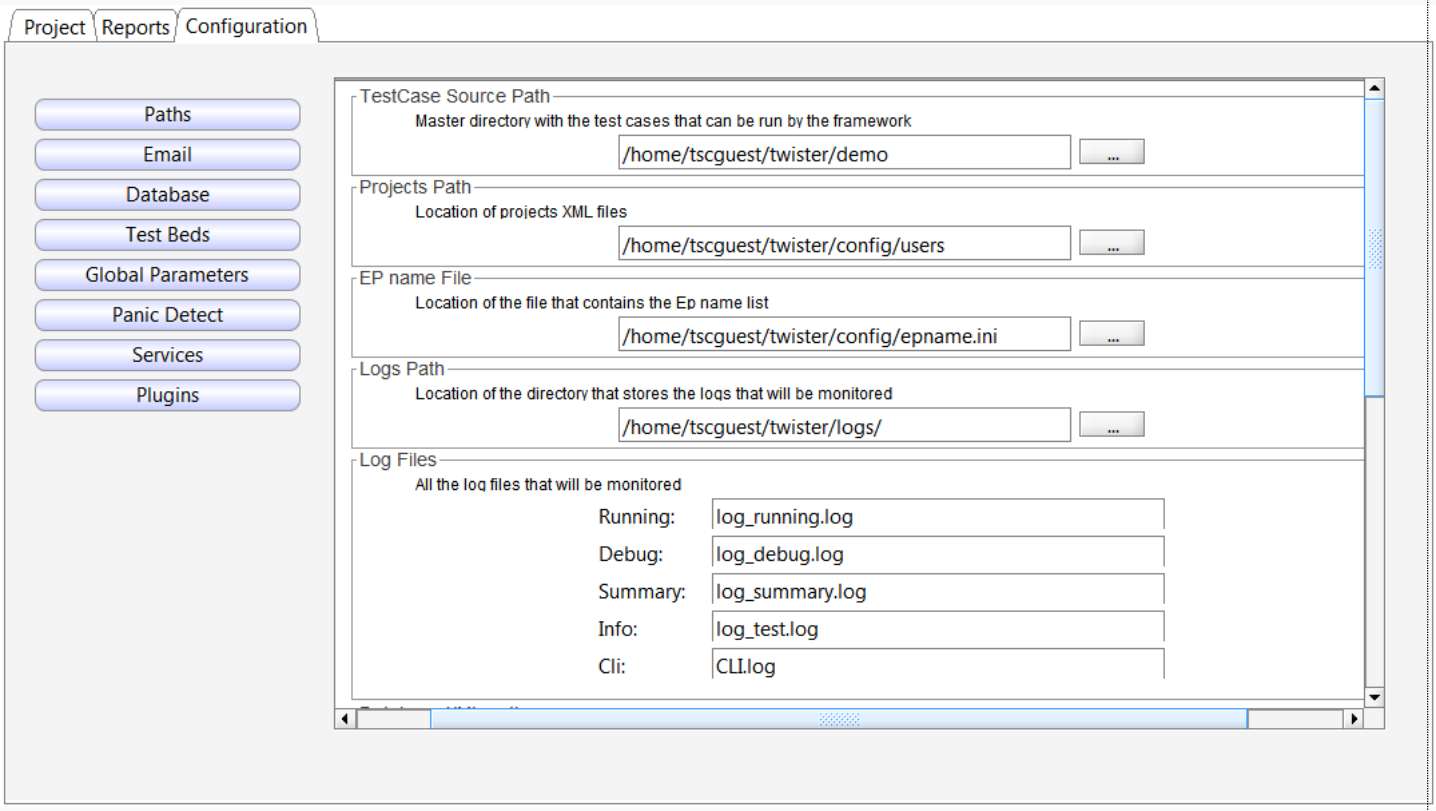
Twister has a few configuration files: XML, ini and Json.

There are 2 types of configurations: per user (located at */$USER_HOME/twister/config*) and global for all users (located at */opt/twister/config*).

- fwmconfig.xml: the **master framework config**. Contains the paths to all the other config files. Config saved per user;
- epname.ini: contains all the EPs for the current user. This must be edited manually. Config saved per user;
- email.xml: contains all the information necessary to send an e-mail from Twister. Config saved per user;
- db.xml: contains all the information about saving and reading from the MySQL database. Config saved per user;
- globals.xml: contains all global variables, per user;
- plugins.xml: contains information about all plugins, per user;
- testsuites.xml: contains all suites and all files for the current run. Saved per user;
- resources.json: contains all the resources from Resource Allocator server, all testbeds and devices. It's a global config;
- services.ini: contains all the services from Service Manager. It's a global config;

Most of the files are generated automatically from the Java applet. They **should not** be edited manually, unless specified otherwise.

10.1 - Configure the paths



★*All the paths below refer to the computer where the **Central Engine** is running.*

Test case source path represents the folder where all test files are located. The files here can be dragged inside suites, in the first tab (suites).

User's path is the folder where the profiles are saved, in the first tab. Usually this doesn't need to be changed.

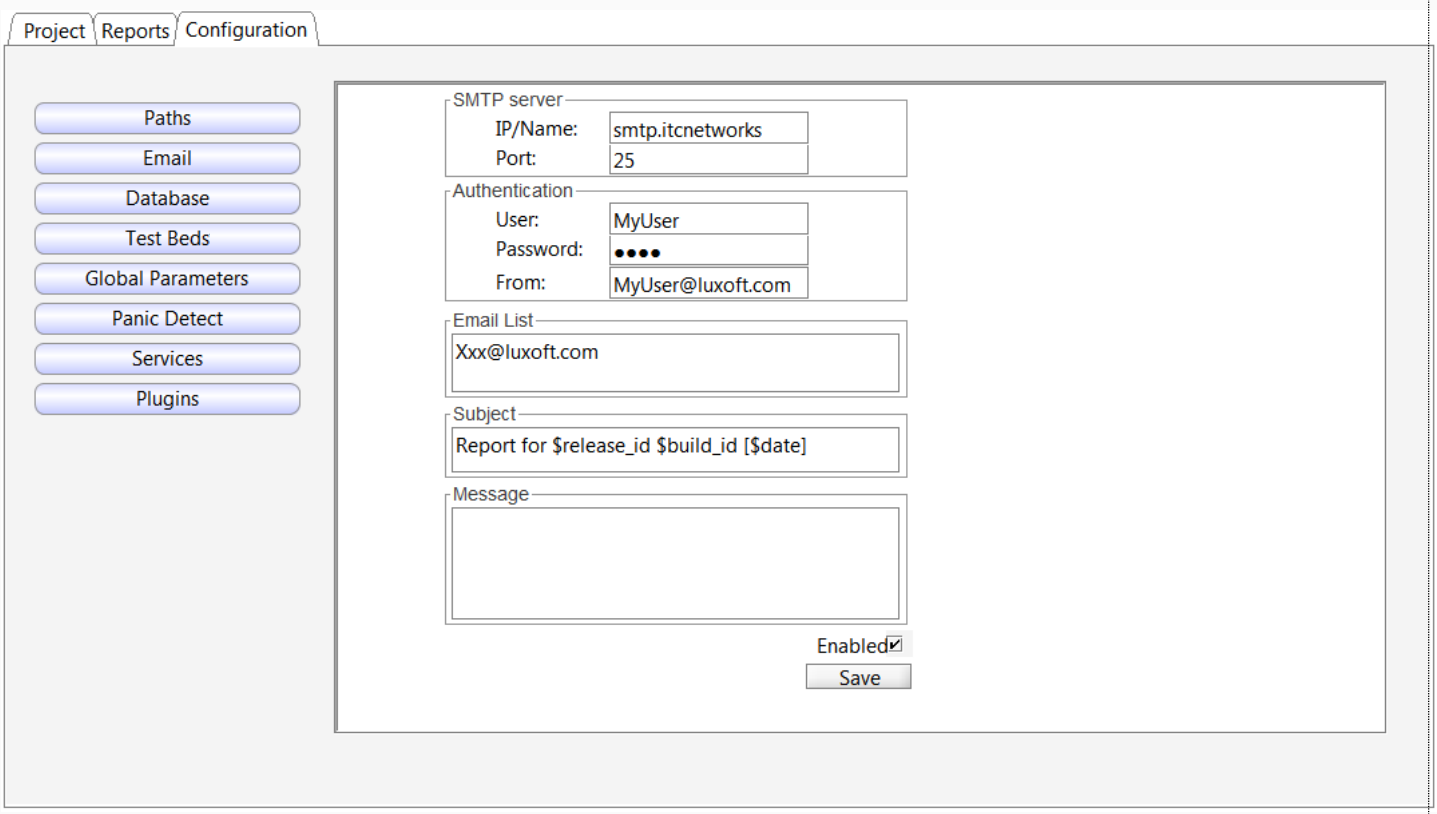
EP Names file stores the list of EPs (the workstations where tests will run). An `*EP*` is just a name to uniquely identify a computer, it can be any string.

Logs path is the folder where all the logs are written. There are 5 major logs: log running, log debug, log summary, log info, and log CLI. Each of the logs will be saved in the logs path, with the name defiled in the configuration. Usually, the logs don't need to be changed.

E-mail XML path, ***Database XML path*** and ***Globals XML path*** are the files that store the information for the next 3 tabs. You can have multiple files, and switch between configurations.

The *Central Engine port* is, of course, the port where the applet connects to the server. The default value is *8000*.

10.2 - Configure the e-mail



Here you can configure the parameters required to connect to a SMTP server and send an e-mail.

The Central Engine will send the e-mail every time the execution finishes for ALL the test files.

The most important fields are: SMTP *IP* and *port*, *username*, *password*, *from* and the *e-mail list*.

Optionally, you can change the subject and add a few lines in the message body.

Both the subject and the message, can contain template variables from `DB.xml` fields, from `*<insert_section>*`.

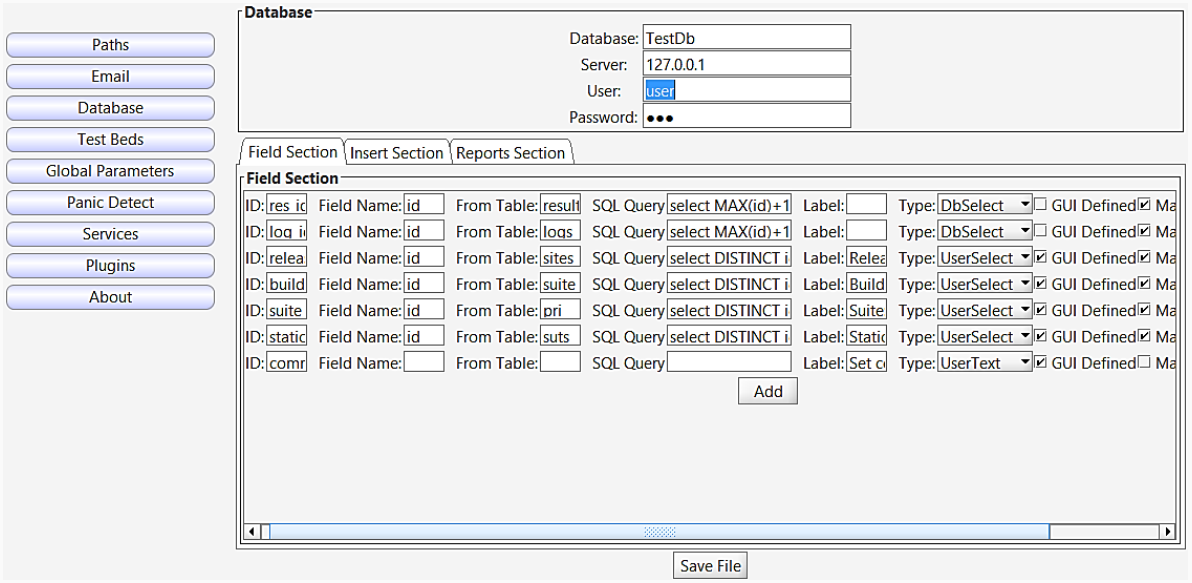
For example, if you defined the fields with IDs `*release_id*`, `*build_id*`, `*suite*`, you can write the subject like :

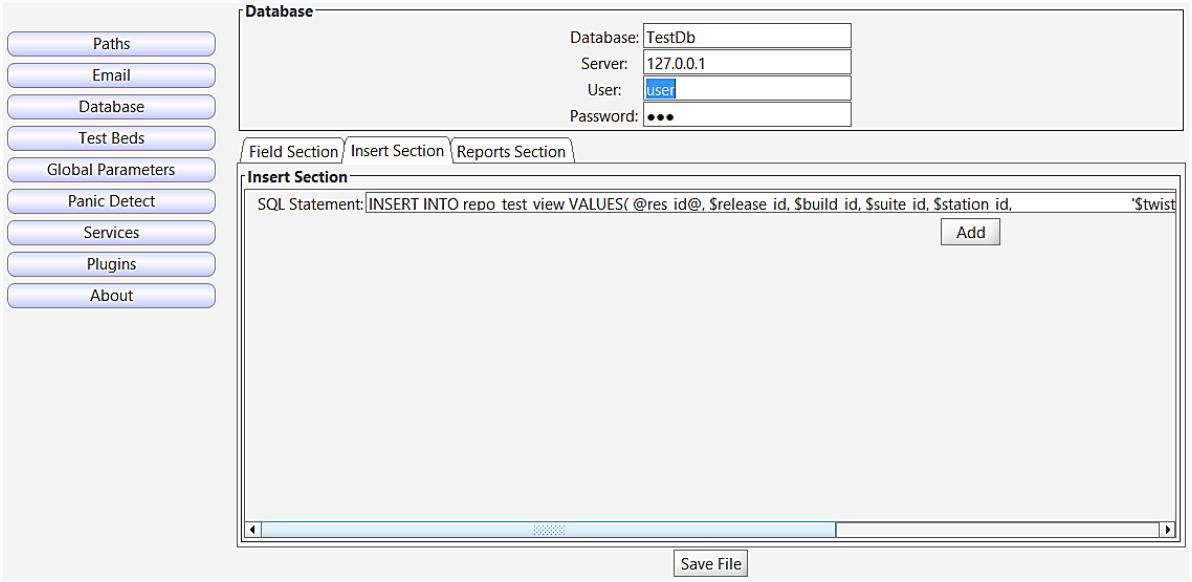
```
E-mail report for R{release_id} B{build_id} - {suite} [{date}]
```

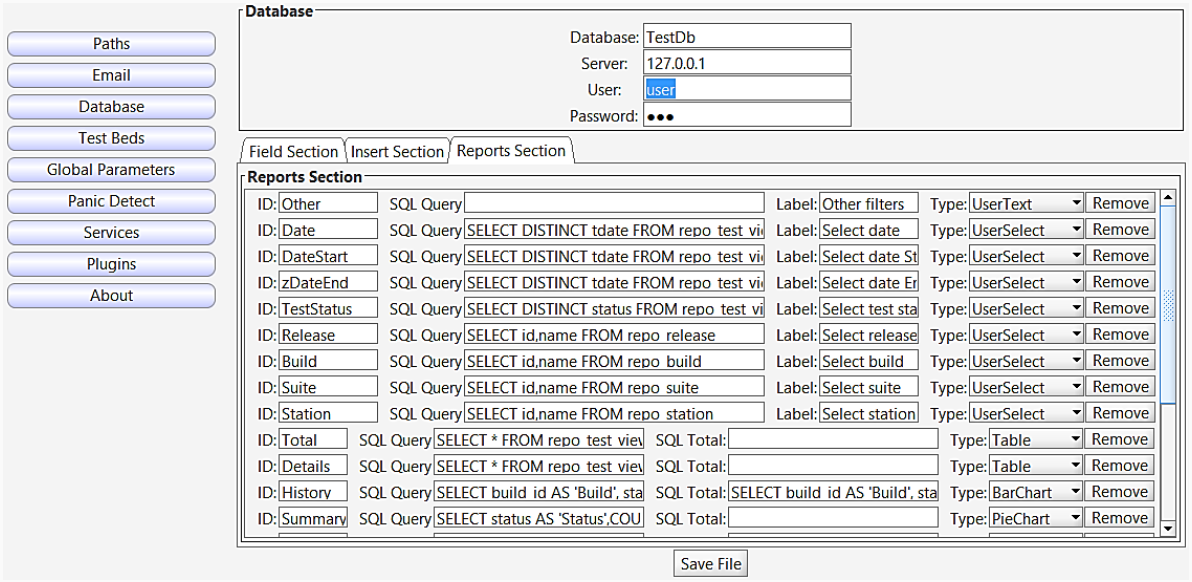
If your release number is `2`, build number is `15` and suite is `Branch Test1`, the subject will be generated like:

```
E-mail report for R2 B15 - Branch Test1 [2012.03.23 13:24]
```

10.3 - Configure the database







All the database information is stored in `*DB.xml*` file, by default. This file can be changed from the interface, in the **Paths tab**. You can have multiple configurations and switch between then.

The ***field section*** contains all the information that was defined in the **Suites tab** for each and every suite, things like: release, build, station, logs, comments, etc.

This information is used when saving the execution results into the database and when sending the report e-mail.

Each field must contain the following tags:

- ID: represents the name of the field and MUST be unique;
- Type: there are 3 types of fields: UserSelect, DbSelect (where you must define an SQL query that will generate a list of value in the interface; the user will select 1 value and that will be saved; the difference between them is that DbSelect will not be shown in the interface) and UserText (free text, you can write anything);
- SQLQuery: this is required for UserSelect and DbSelect fields. The query must be defined in such a way that the values will be unique (eg: by using SELECT DISTINCT id, name FROM …) and should select 2 columns. The first column will be the ID and second will be the description of the respective ID;
- GUIDefined: if a field is not GUI defined, it will be visible in the **Suites tab**, when editing suites;
- Mandatory: if a field is mandatory, each suite from the **Suites tab** must have a value for this field. If the user doesn't choose a value, he will not be able to save the profile, or generate the Suites XML;
- Label: a short text that describes the field, in the interface; it's not necessary for DbSelect fields, because they are not visible in the interface.

***Examples** of fields*:

```
<field ID="res_id"  Type="DbSelect"
SQLQuery="select MAX(id)+1 from repo_test_view"
Label="-"  GUIDefined="false"  Mandatory="true" />

<field ID="release_id"  Type="UserSelect"
SQLQuery="select DISTINCT id, release_name from t_releases"
GUIDefined="true"  Mandatory="true"  Label="Release:" />

<field ID="build_id"  Type="UserSelect"
SQLQuery="select DISTINCT id, build_name from t_builds"
Label="Build:"  GUIDefined="true"  Mandatory="true" />

<field ID="comments"  Type="UserText"  SQLQuery=""
Label="Set comments:"  GUIDefined="true"  Mandatory="false" />
```

The ***insert section*** defines a list of SQL queries that will execute every time the execution finishes for ALL the test files. All queries are executed for each and every test file.

The insert queries use the information from the fields described above. A file can only access the fields defined in his parent suite.

Other than that, the queries can access a list of variables passed from the Central Engine, which describe how the execution was completed. Here are the variables:

- **$twister_ce_os** = the operating system of the computer where Central Engine runs;
- **$twister_ep_os** = the operating system of the computer where Execution Process runs;
- **$twister_ce_ip** = the IP of the Central Engine;
- **$twister_ce_hostname** = the host name of the Central Engine;
- **$twister_ep_ip** = the IP of the Execution Process;
- **$twister_ep_hostname** = the host name of the Execution Process;
- **$twister_ep_name** = EP name, defined in **Suites tab**;
- **$twister_rf_fname** = the path to Twister resources file (default is `resources.json`);
- **$twister_pf_fname** = the path to Twister project file (default is ` project_users.json`);
- **$twister_ce_python_revision** = python version from Central Engine;
- **$twister_ep_python_revision** = python version from Execution Process;
- **$twister_suite_name** = suite name, defined in **Suites tab**;
- **$twister_tc_name** = the file name of the current test;
- **$twister_tc_full_path** = the path + file name of the current test;
- **$twister_tc_title** = the title, from the **Suites tab**;
- **$twister_tc_description** = the description, from the **Suites tab**;
- **$twister_tc_status** = the final status of the test: pass, fail, skip, abort, etc;
- **$twister_tc_crash_detected** = if the file had a fatal error that prematurely stopped the execution;
- **$twister_tc_time_elapsed** = time elapsed;
- **$twister_tc_date_started** = date and time when the running started;
- **$twister_tc_date_finished** = date and time when the running finished;
- **$twister_tc_log** = the complete log from execution.

These variables can be used in the query like `$variable_name`, or `@dbselect_field_name@`.

Only the fields of type **DbSelect** are surrounded by @.

***Examples** of database inserts*:

```
<sql_statement>
INSERT INTO gg_regression
(suite_name, test_name, status, date_start, date_end, build, machine)
VALUES
( '$twister_suite_name', '$twister_tc_name', '$twister_tc_status',
'$twister_tc_date_started', '$twister_tc_date_finished', '$release.$build',
'$twister_ep_name' )
</sql_statement>
```

Or:

```
<sql_statement>
INSERT INTO results_table1
VALUES
( @res_id@, $release_id, $build_id, $suite_id, $station_id,
'$twister_tc_date_finished', '$twister_tc_status', '$comments' )
</sql_statement>
```

★In this last example, `*res_id*` is a DbSelect field with the query defined as:
`SELECT MAX(id)+1 FROM results_table1`.

The ***reports section*** defines all the information exposed to the reporting framework.

In this section you can define the *fields*, the *reports* and the *redirects*.

The **fields** must have the following properties:

- ID: represents the name of the field and MUST be unique;
- Type: there are 2 types of fields: UserSelect (where you must define an SQL query) and UserText (free text, you can write anything);
- SQLQuery: this is required only for UserSelect fields. The query should select two columns: the first is the ID and the second is a name, or a description of the respective ID. If the table where you have the data doesn't have any description associated with the ID, you can use only the ID;
- Label: a short text that describes the field, when the user is asked to select a value.

***Examples** of report fields*:

```
<field ID="Dates" Type="UserSelect" Label="Select date:"
SQLQuery="SELECT DISTINCT date FROM results_table1 ORDER BY date" />

<field ID="Statuses" Label="Select test status:" Type="UserSelect"
SQLQuery="SELECT DISTINCT status FROM results_table1 ORDER BY status" />

<field ID="Releases" Label="Select release" Type="UserSelect"
SQLQuery="SELECT DISTINCT SUBSTRING(build, 1, 6) AS R FROM results_table1 ORDER BY R" />

<field ID="Other" Type="UserText" Label="Type other filters:" SQLQuery="" />
```

The **reports** must have the properties:

- ID: represents the name of the report and MUST be unique;
- Type: there are 4 types of reports: Table (an interactive table is generated; the table can be sorted and filtered dynamically), PieChart, BarChart and LineChart (they show both the chart and the table; for PieChart report, the SQL query must be defined in such a way that the first column is a string describing the data, and the second column is an integer or float data; BarChart and LineChart must also have the query generate 2 columns, the first is a number and the second is a label or another number);
- SQLQuery: all reports must define an SQL query. If the type of report is Table, it can select any number of fields (although it's recommended to use a maximum of 10, to fit on the screen without having to scroll to the right). If the report is a chart, you must select only 2 columns. The query can use any, or none of the fields described above. When a field is used in the query, the reporting framework will require the user to choose a value, before displaying the report.

Examples or reports:

```
<report ID="Details (build)" Type="Table"
SQLQuery="SELECT * FROM  results_table1  WHERE build='@Build@' ORDER BY id" />

<report ID="Details (suite)" Type="Table"
SQLQuery="SELECT * FROM  results_table1  WHERE build='@Build@' AND suite_name='@Suite@' " />

<report ID="Summary" Type="PieChart"
SQLQuery="SELECT status AS 'Status', COUNT(status) AS 'Count' FROM  results_table1  WHERE build=
'@Build@' group by status " />

<report ID="Pass Rate" Type="LineChart"
SQLQuery="SELECT Build, COUNT(status) AS 'Pass Rate (%)' FROM  results_table1  WHERE Build LIKE
'@Release@%' AND status='Pass' GROUP BY Build"
SQLTotal="SELECT Build, COUNT(status) AS 'Pass Rate (%)' FROM  results_table1  WHERE Build LIKE
'@Release@%' GROUP BY Build" />
```

The **redirects** must have the properties:

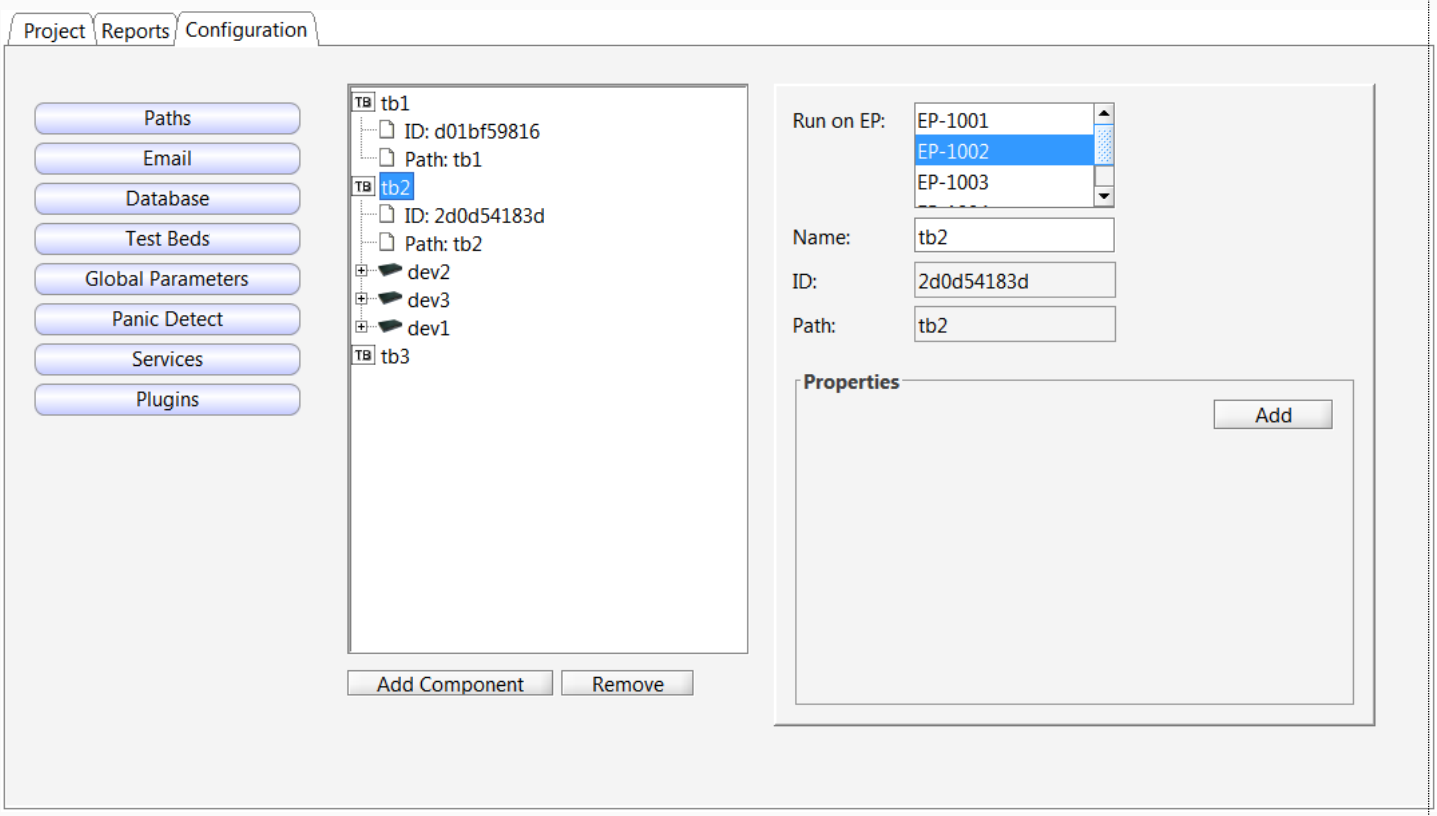
- ID: represents the name of the redirect and MUST be unique. Ideally, the ID should start with the word `*goto*`;
- Path: is the full path to a HTML page. It can be a link to a static page, to PhpMyAdmin for the current database, or a user web page served by any web server.

Examples of redirects:

```
<redirect ID="goto PhpMyAdmin" Path="http://my-server/phpmyadmin" />

<redirect ID="goto PHP Report" Path="http://my-server/some-report.php" />
```

10.4 - Config the devices (testbed)



The Resource Allocator server is used to view and edit the testbed properties.

The testbed is global for all users.

Each device == node == resource must have a name and some properties in the form of: {*key*: *value*}.

The name of a resource must be unique in its parent. For example you cannot have more nodes called `Device1` in parent `Testbed1`, but you can have nodes called `Device1` for both `Testbed1` and `Testbed2`.
This is important, because each resource can be accessed using its ID, or its full path (just like a Unix file system).

The Resource Allocator server exposes a simple API for accessing the resources:

- **getResource**(ID or full path) - returns a dictionary containing all the node properties
- **setResource**(name, parent ID or full path, properties in dictionary or JSON string)

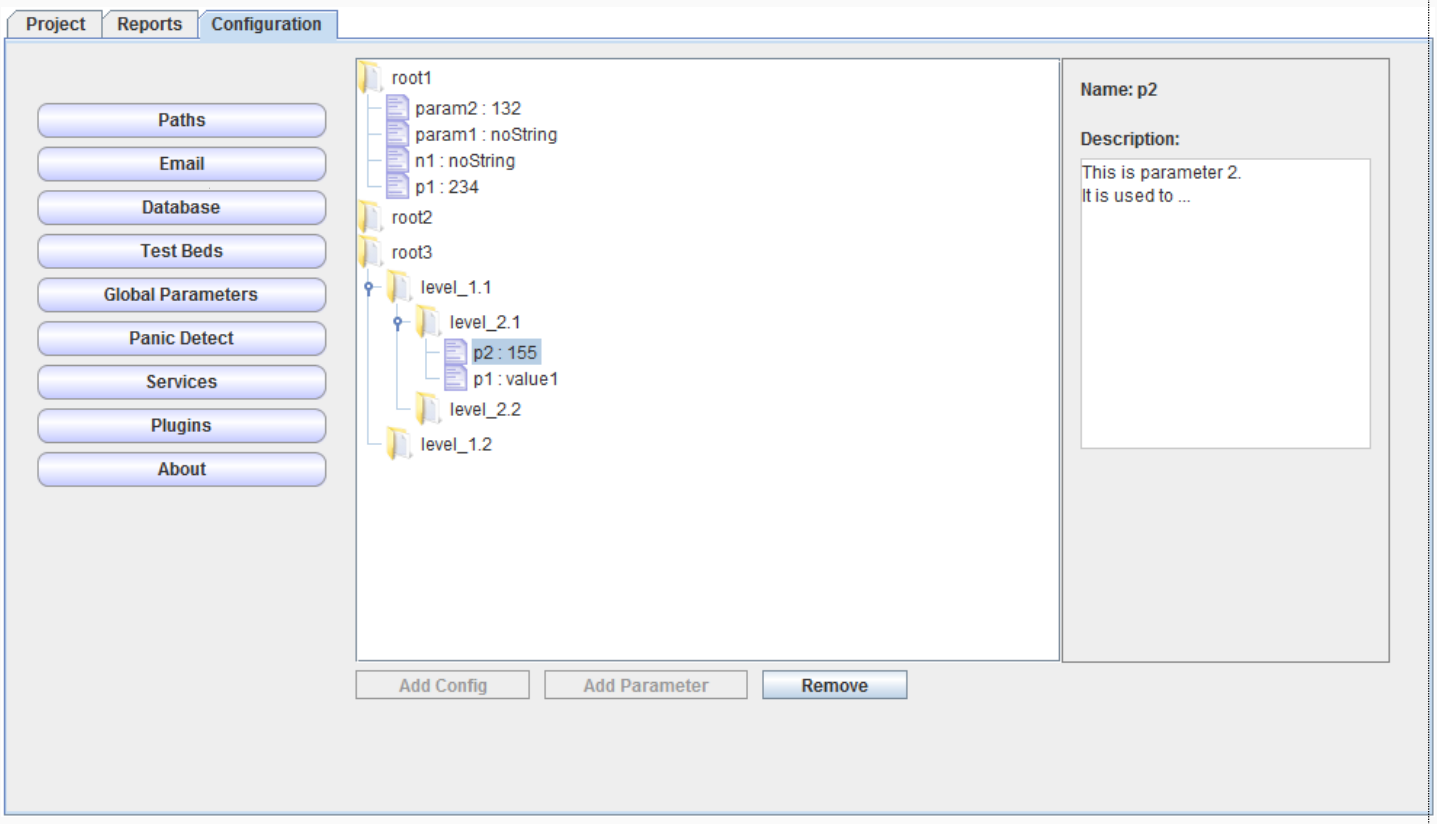
- This function is used to CREATE and MODIFY nodes. If the resource is created, the ID of the new resource is returned. If the resource is updated, the function returns True.
Example: `setResource('module1', '/tb1/device2', '{"ip":"10.0.0.1", "port":"80"}');`

- **renResource**(ID or full path, new name) - renames resources or properties
- **deleteResource**(ID or full path) - deletes resources or properties
- **getResourceStatus**(ID or full path) - obsolete

Examples:
Let's say there are 3 testbeds: *tb1*, *tb2*, *tb3*. Each testbed has 2 devices: *dev1* and *dev2*. Each device has 3 modules: *mod1*, *mod2*, *mod3*.

To access module 2 from device 1 from testbed 3, you can use: `getResource('/tb3/dev1/mod2')`.
To access testbed 1, you can use: `getResource('/tb1')`.
To rename device 2 from testbed 2, you can use: `renResource('/tb2/dev2', 'dev_x2')`. Note that the new name is just a string, not the full path.

10.5 - Config the global parameters



Global parameters are variables available in all test files. There is no need to import any library.

The parameters are stored per user!

The API is very simple: in any test, use **`getGlobal`**`( name )` and **`setGlobal`**`( name, value )`. You don't need to import anything.

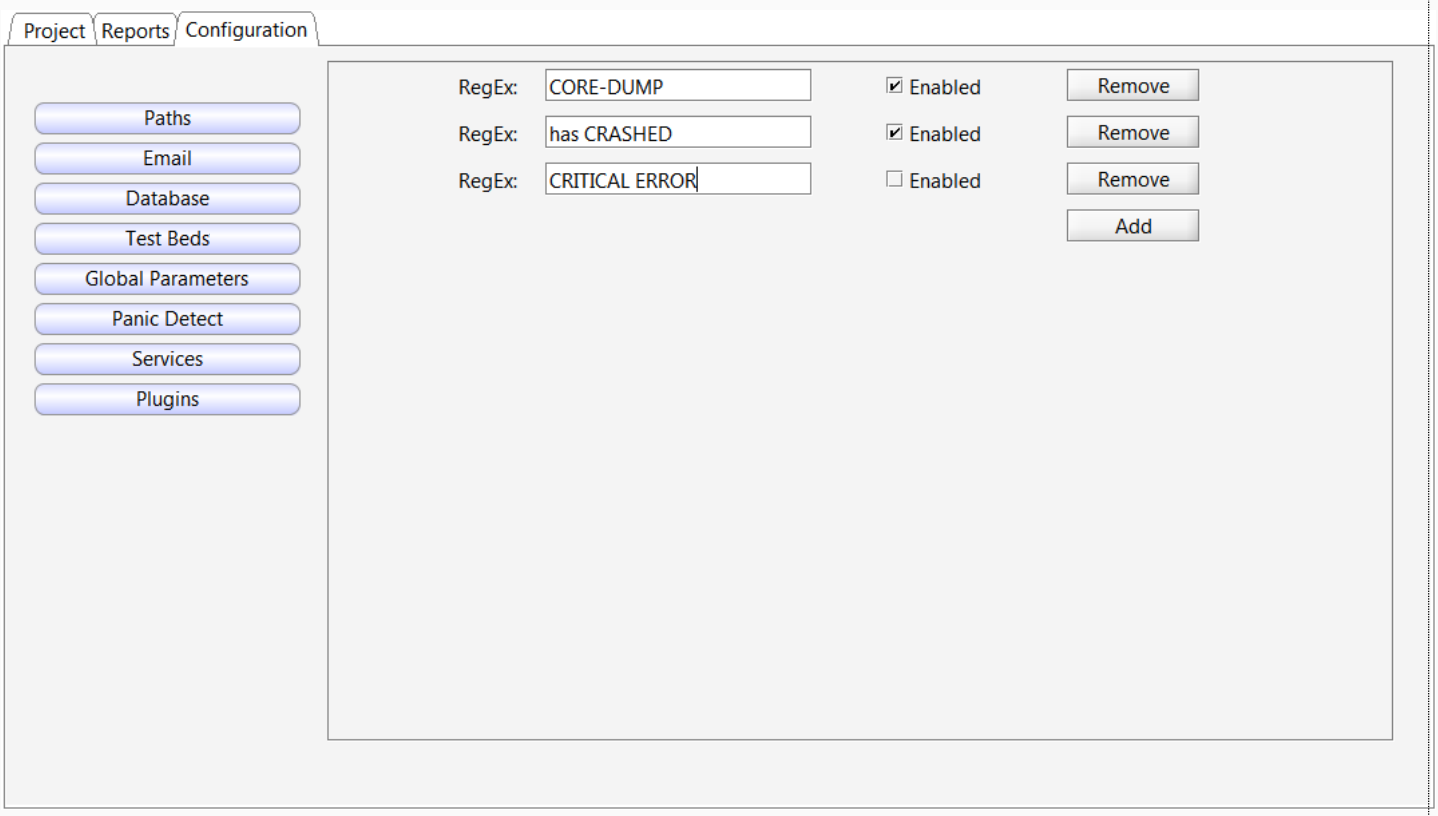
If the ***name*** is a folder (not a variable with a value), instead of a value, getGlobal returns a dictionary. If the parameter *name* doesn't exist, getGlobal returns *False*.

The setGlobal function will update, create or delete parameters. If the name exists, it is updated. If it doesn't exist, it is created. If you use **setGlobal** (name, False), the parameter is deleted.
The changes made by setGlobal are temporary and will RESET every time the tests start running.

There are 3 types of parameters:

1. the default parameters stored in the configuration, that are saved in globals.xml file ;
2. the serializable parameters saved by the test files are shared between all Eps from a user ;
3. the complex, not serializable parameters are stored on the EP that is running the tests.

10.6 - Config `panic detect`



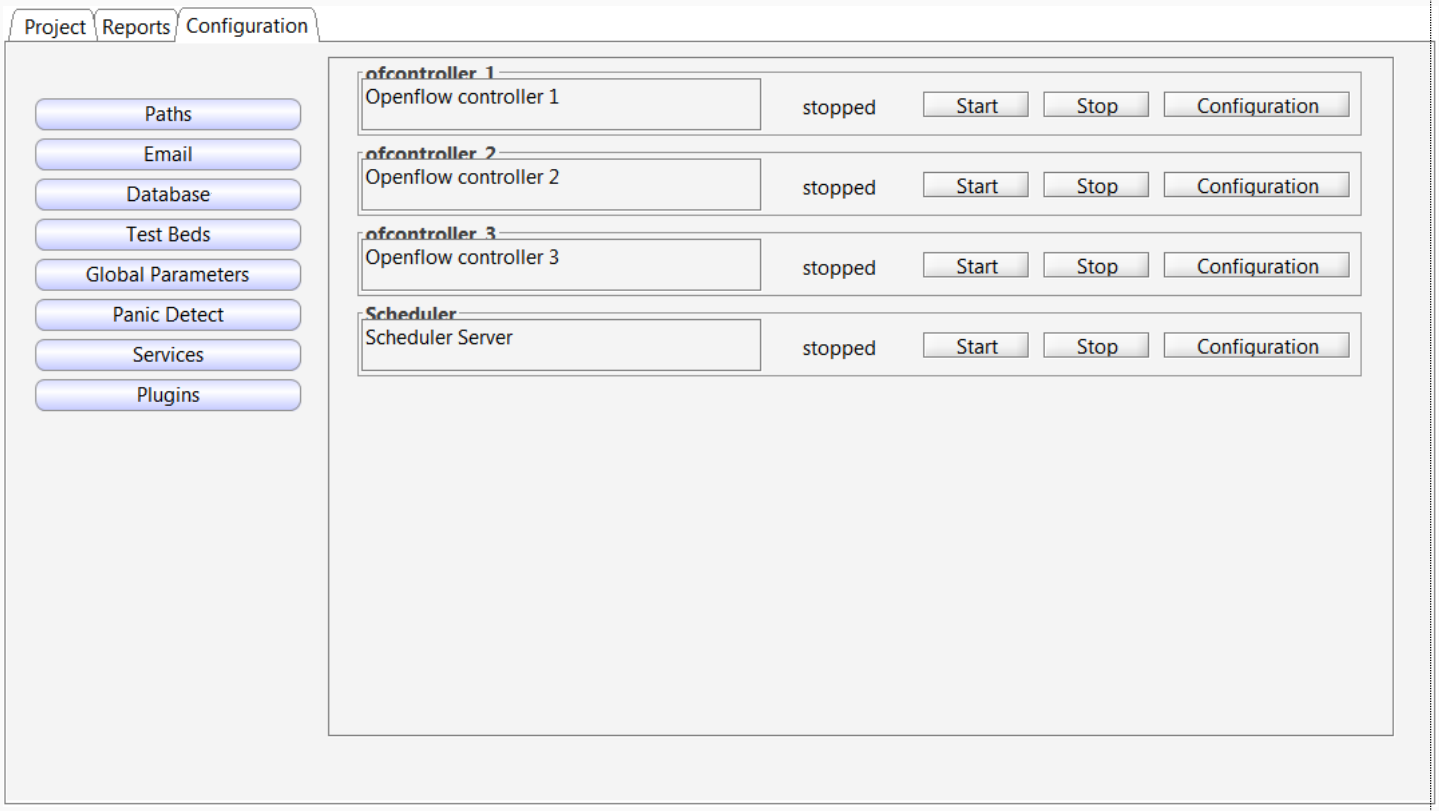
Panic detect is a mechanism that allows the users to add `expressions` that will be searched in the test logs. If any of the expressions is detected, the execution STOPS.

An `*expression*` can be a normal string, or a regex.

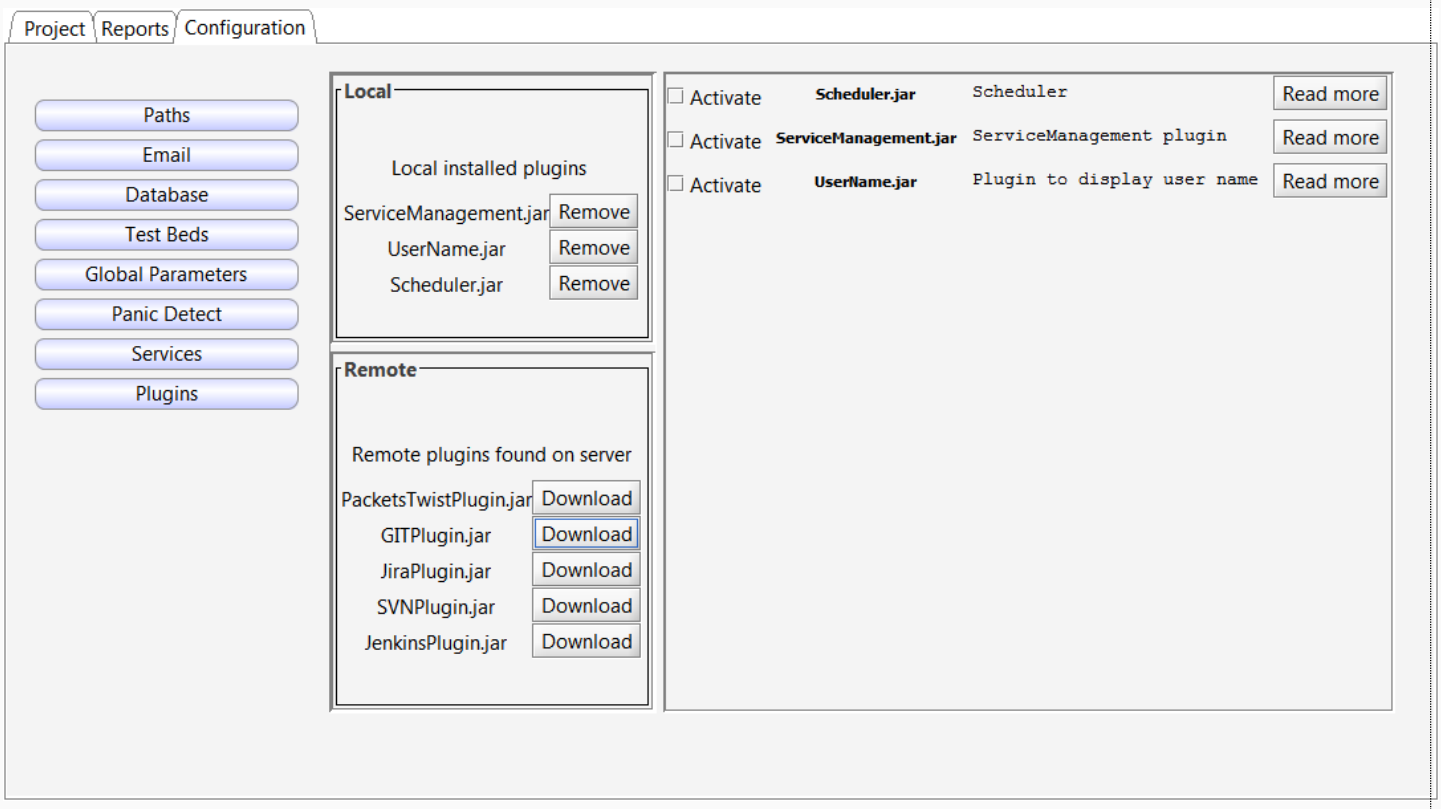
This is useful to check for core dumps and critical errors; in these extreme cases, it's useless to run any test.

10.7 - Services and Plug-ins

Print screen with Services



Print screen with Plug-ins



More about this in the `**Twister Libraries, Plugins, Services**` help file.

How to write Twister tests

Twister framework can run **Python**, **TCL** and **Perl** (limited) tests.

Writing a **Twister test** is just like writing a normal Python 2.7 test, or TCL 8.5 test, or Perl test, with a few exceptions.

Twister tests are most likely **incompatible** with the original language; for example a Twister Python test will not run with Python by default, because Twister inserts a few helper variables and functions, which are not available in the usual environment. So if the a test uses the helper variables and functions, the script will become incompatible with the original Python/ TCL/ Perl language and will be executed only from Twister.

Variables inserted in all Twister tests:

- **USER** : the username running the current test ;
- **EP** : the name of the Execution Process running the current test ;
- **SUITE_NAME** : the name of the suite that contains the current test ;
- **FILE_NAME** : the full path of the test file from the machine that runs the Central Engine ;
- **currentTB** : the current test bed ;
- **PROXY** : this is a pointer to the Central Engine XML-RPC server. It is used for development.

And a few functions:

- **logMsg**`( logType, message )` : this function sends a message in a special log and will not appear in the CLI. Valid log types are : *logRunning*, *logDebug* and *logTest*. It is used for sending debug messages from the tests.
- **getGlobal**`( path )` : get a global parameter;
- **setGlobal**`( path, new_value )` : set a global parameter;
- **getResource**`( ID or full path )` : get a resource;
- **setResource**`( ID or full path, new name )` : create, or update a resource;
- **renResource**`( ID or full path,  new name )` : rename one resource or property of a resource;
- **deleteResource**`( ID or full path )` : delete one resource or property of a resource;
- **getResourceStatus**`( ID or full path )` : get the status of a resource: free, busy, or reserved;
- **reserveResource**`( ID or full path )` : reserve one resource;
- **py_exec** *some_python_command* : this function works **only in TCL** tests and allows running Python commands, or calling functions and objects from global parameters, defined in the previous tests.

11 – Performance and troubleshooting

The Central Engine and the Reporting Server are instances of Python CherryPy and were tested with 750+ simultaneous connections, without crashing, or losing connection.

★An article concerning python web servers: *http://nichol.as/benchmark-of-python-web-servers*.

Even if the Central Engine is fast enough, for a smooth experience, it's not recommended to run more than 100 Execution Processes on one Central Engine instance. If you need more, you can simply open another instance of CE, on a different port and connect the rest of the clients on the new one.

The Execution Processes should be running on different workstations and their performance depends on the hardware of the respective machine.

All services have logs that describe every operation that is being executed. If something fails, it will be easy to know where exactly the error was produced.

On the server side, you can check the logs from */opt/twister/server_log.log*, or */opt/twister/logs/* folder.
On the client side, you can check the logs from */$USER_HOME/twister/.twister_cache/*. Every EP has its own log.

If you notice a crash, or wish to report a bug, you can use the `*create_bug_report.py*` script from the twister folder,

§