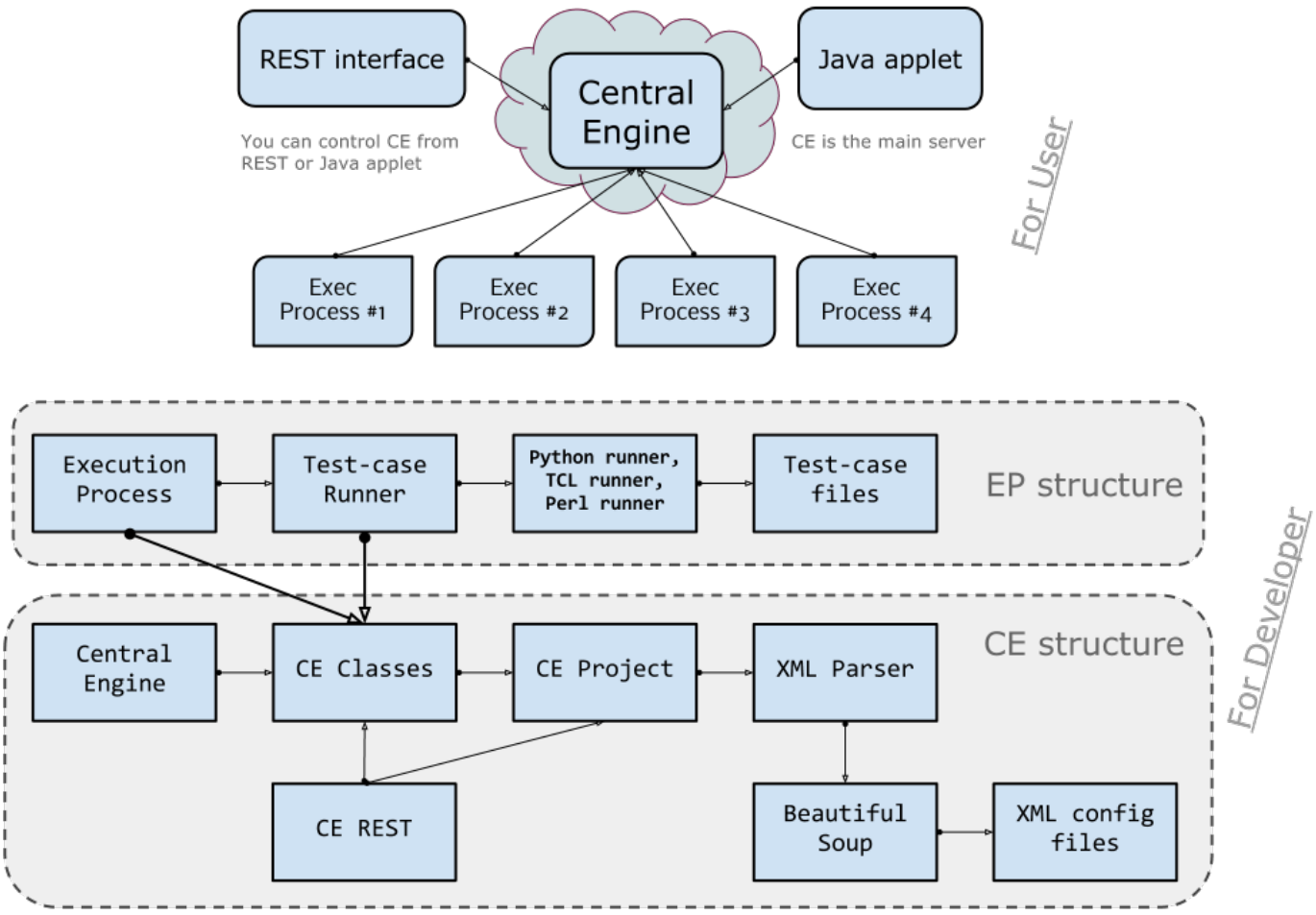
Twister framework guide

Contents

1 - How to install the framework
2 - Dependencies list
3 - Twister services
4 - How to compile the Java GUI
5 - Overview of the Java GUI
6 – How to define plugins for Java GUI
7 - How to define the suites and add tests
8 - How to run the test files
9 - How to configure the framework
10 – Performance and troubleshooting

Twister concept



1 - 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. Only the Execution Process can also run on Windows;
- Python 2.7. Python is installed by default, on most Linux systems; the framework is written and tested in Python 2.7;
- Python Tkinter. This is included by default in Python, but sometimes it doesn't come with the library `*python-tk*`, so it has to be installed;
- TCL Expect libraries. They are 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
exit()
```

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

The installer is located in the folder `*installer*` and is also written in Python. It must run as ROOT, because it will try to automatically install all the necessary Python libraries.

The recommended commands for starting the installer:

```
su --login
# Type the root password
python2.7 installer.py
```

If you are installing the *Twister Server*, you should be connected to internet, or else, before running the installer, you must also install `*Python-DEV*`, and then `*python-mysql*`.

The *Twister Client* doesn't have any required dependencies; `*pExpect*` is optional.

The installer will guide you through all the steps, in order. These are the steps :

1. Select what you wish to install (*client*, *server*, or *both*);
2. Select the user you want to install for (if there's only 1 user, this step is skipped)
3. 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 will be installed in the home of the selected user, in the folder `*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.

2 - Twister Framework dependencies list

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

- BeautifulSoup : (*www.crummy.com/software/BeautifulSoup/*)
  - XML and HTML documents parser;
  - BeautifulSoup is included in `*trd_party*` folder and *should not* be installed;

- 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 reports and the Java Applet;

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

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

3 - Twister services

Twister framework has 4 services:

1. the Central Engine = central server for script and library files;
2. the HTTP Server = server for reporting framework and java applet GUI;
3. the Resource Allocator = server for managing devices (allocate, release, add properties);
4. the Execution Process = service that runs the script files (python, TCL, perl).

All the executables are located in the `*bin*` folder.

The first 3 services should run on the same machine, because they depend on the same config files.

Before executing any service, make sure you have exported the `*TWISTER_PATH*` environment variable. It should point to *$HOME* of the user + `*twister*` folder.

The Linux EP service must be configured before run. You have to edit the file `*config_ep.json*`, 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:

```
# For Central Engine
./start_ce
# For HTTP Server
./start_http
# For Resource Allocator
./start_ra

# For Linux Execution Process
./start_ep.py
```

4 - How to compile the Java GUI

The Java graphical user interface is located at `*client/userinterface/java*`.

The binary JAR files are included in folders `*target*` and `*extlibs*`, respectively.
The sources are located in folder `*src*`, in *client* or *server*.

If you need to modify the sources, the binary files must be recompiled, this is done in 4 steps:

1. compile the source files and generate JAR file;
2. generate a keystore, or import a certificate (*this is done only* ***the first time****!*);
3. sign the JAR file with the generated keystore;
4. export the keystore as X509 certificate.

All the steps require JDK (*Java Development Kit*).

In Linux, these are the commands:

```
# You can change the path to JDK here
JDK_PATH=/opt/java1.7/bin

EXTLIBS=extlibs/jsch-0.1.44.jar:\
extlibs/ws-commons-util-1.0.2.jar:\
extlibs/commons-vfs-1.0.jar:\
extlibs/VFSJFileChooser-0.0.3.jar:\
extlibs/jcommon-1.0.16.jar:\
extlibs/jxl.jar:\
extlibs/ws-commons-util-1.0.2.jar:\
extlibs/xmlrpc-client-3.1.3.jar:\
extlibs/xmlrpc-common-3.1.3.jar

# Compile sources
$JDK_PATH/javac  -deprecation -d classes -source 1.6 -target 1.6  -cp $EXTLIBS  src/*.java

cd classes
# Generate JAR file
$JDK_PATH/jar  cf ../target/applet.jar  Icons *.class

# Generate a keystore using a step by step wizard
$JDK_PATH/keytool -genkey -keyalg rsa -validity 360000 -alias Twister -keypass password
-storepass password

# OR, import a certificate, CER format
$JDK_PATH/keytool -import -alias Twister -file certificate_file.cer

cd ../target

# Sign the JAR file using the keystore
$JDK_PATH/jarsigner  applet.jar Twister -storepass password
# Export the keystore as X509
$JDK_PATH/keytool  -export -alias Twister -rfc -file sig.x509 -storepass password
```

In Windows, these are the commands:

```
SET EXTLIBS="extlibs/jsch-0.1.44.jar;extlibs/ws-commons-util-1.0.2.jar;extlibs/commons-vfs-
1.0.jar;extlibs/VFSJFileChooser-0.0.3.jar;extlibs/jcommon-1.0.16.jar;extlibs/jxl.jar;extlibs/ws-
commons-util-1.0.2.jar;extlibs/xmlrpc-client-3.1.3.jar;extlibs/xmlrpc-common-3.1.3.jar"

# Compile sources
javac.exe  -deprecation -d classes -source 1.6 -target 1.6  -cp %EXTLIBS%  src/*.java

cd classes
# Generate JAR file
jar.exe  cf ../target/applet.jar  Icons *.class

# Generate a keystore using a step by step wizard
keytool.exe -genkey -keyalg rsa -validity 360000 -alias Twister -keypass password -storepass
password

cd ../target

# Sign the JAR file using the keystore
jarsigner.exe  applet.jar Twister -storepass password
# Export the keystore as X509
keytool.exe  -export -alias Twister -rfc -file sig.x509 -storepass password
```

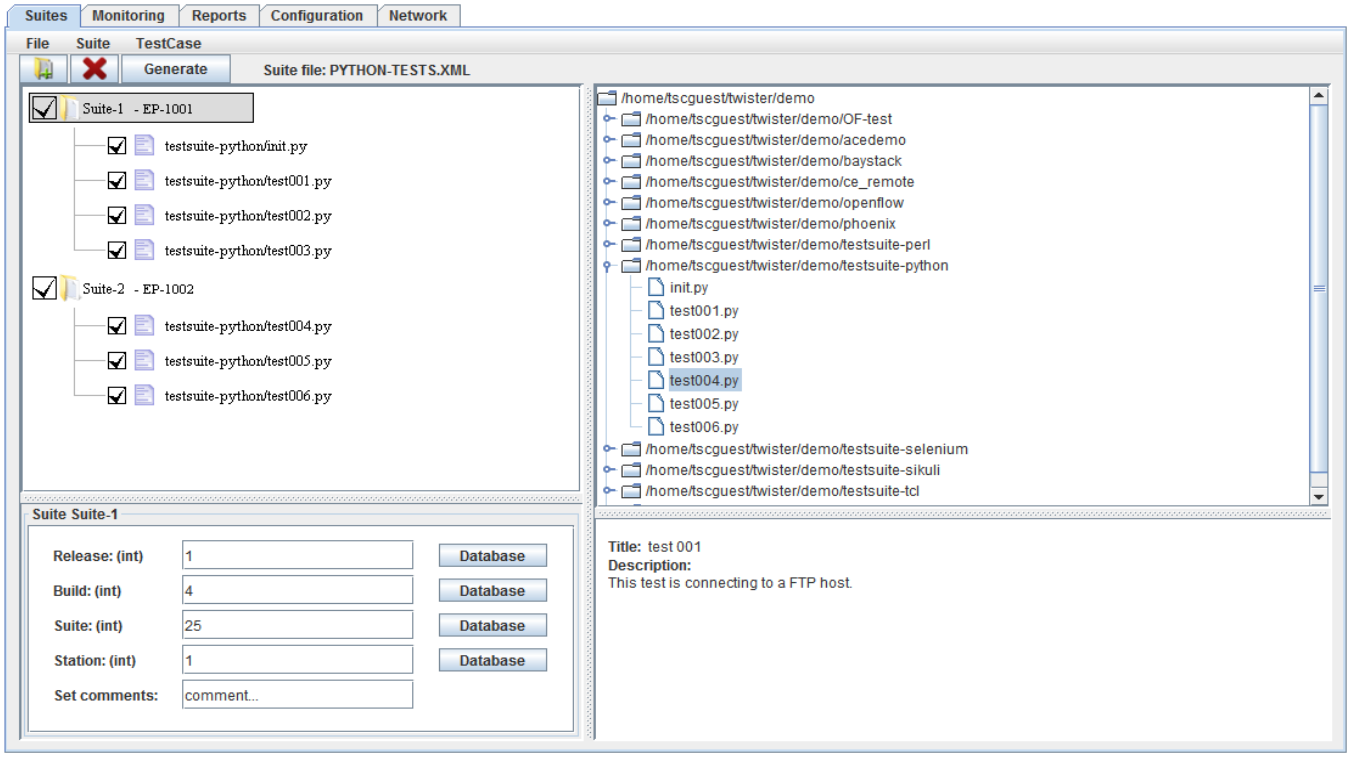
All the files from `*target*` and `*extlibs*` should be copied in `*server/httpserver/gui*`. When starting the HTTP Server, the GUI will be visible in the browser, in `*http://server-ip:port/gui*`.

5 - Overview of the Java GUI

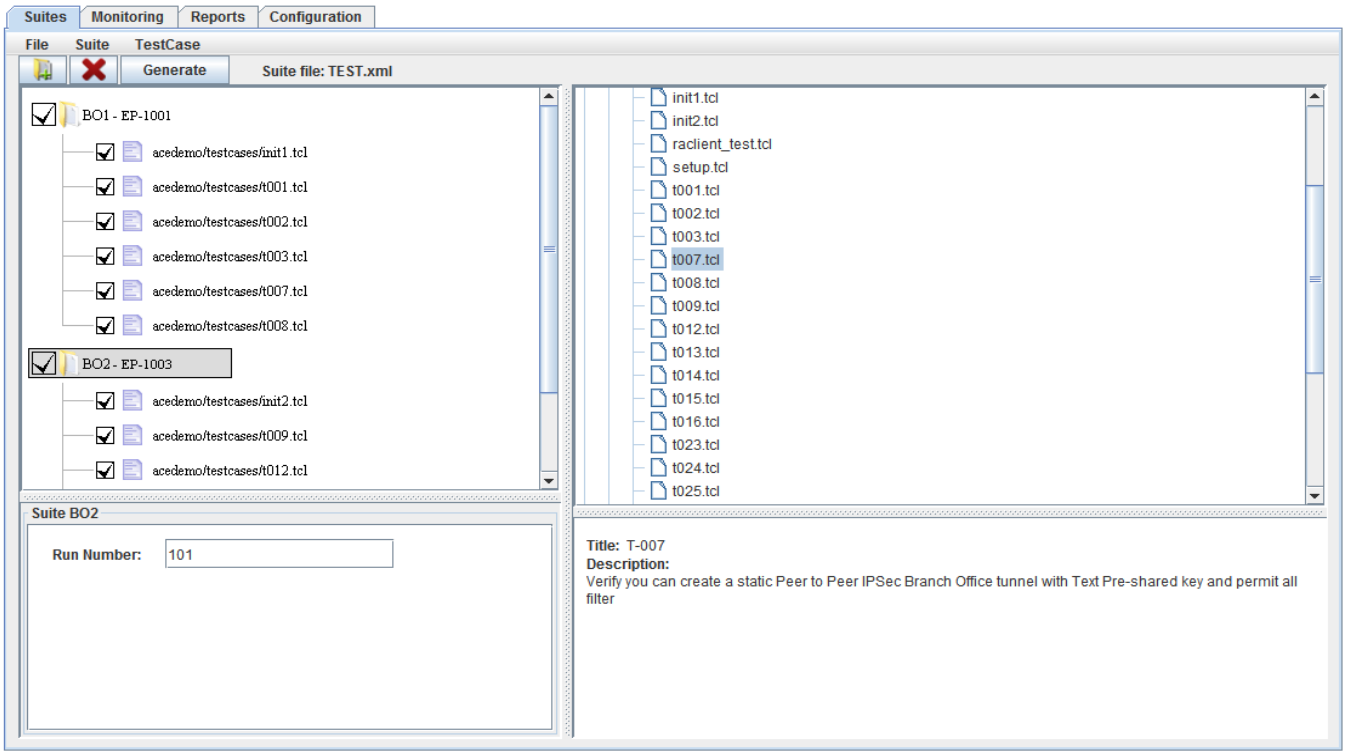
The **first tab (Suites)** is split in four panes:

- 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 suite information is added. This information is defined in the file `DB.xml`, section name `field_section` (*more about this in the configuration section*);
- bottom right, you can see information about the currently selected test file.

A configuration, with Python scripts :



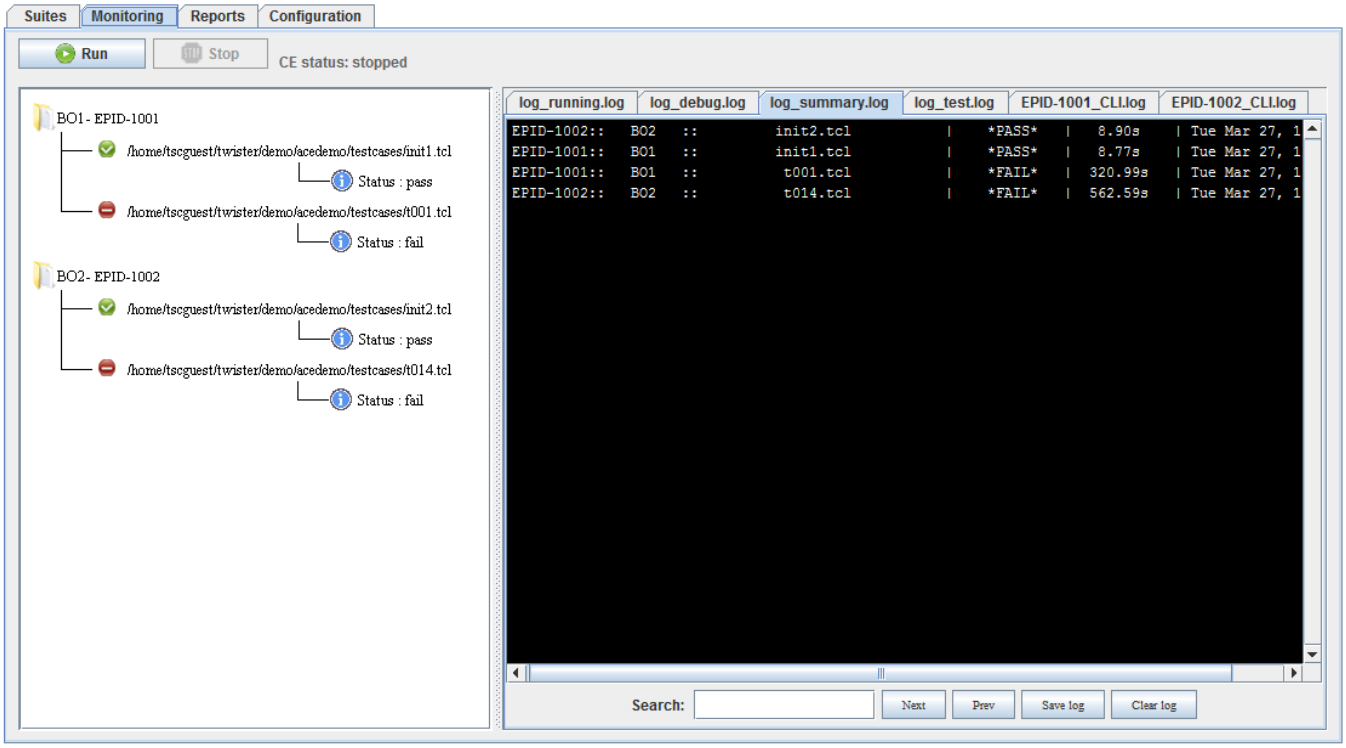
A different configuration, with TCL scripts :



The **second tab (Monitoring)** is divided into:

- 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 of the tests. The logs can be cleaned, exported, or searched for keywords.

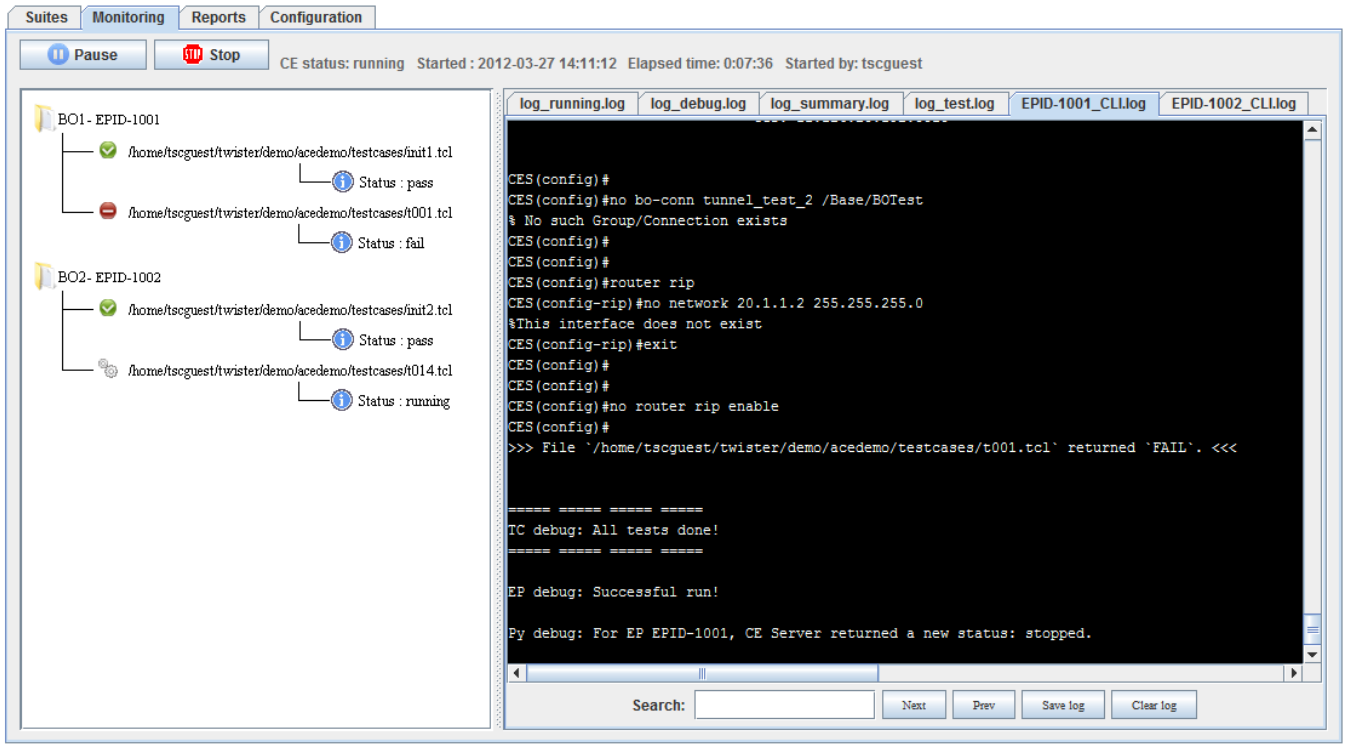
Here the Central engine is stopped ; in this case you can see the most recent statuses :



At the top, there are two buttons, that 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.

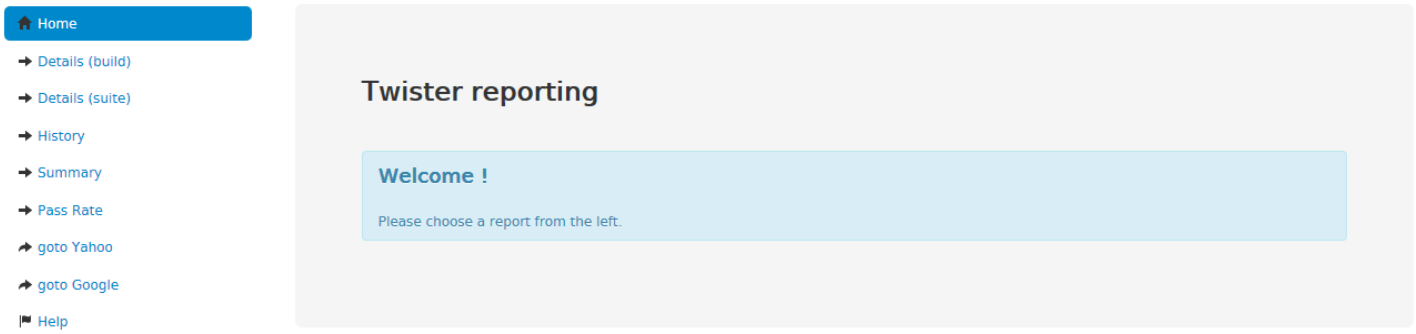
When the Central engine is running ; the first suite is complete, the second is still running :



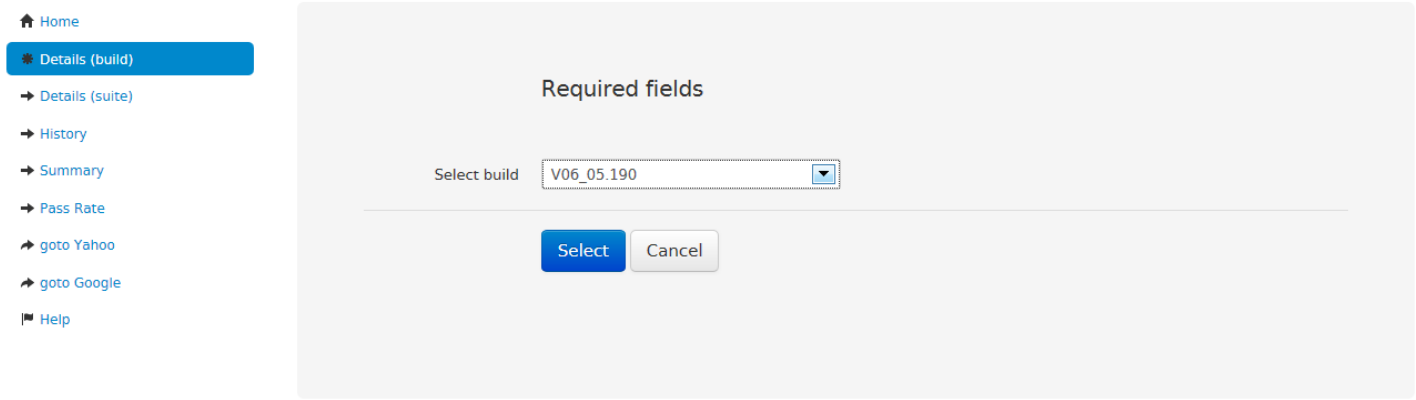
The **third tab (Reports)**

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

Reporting home



A report with user chosen fields



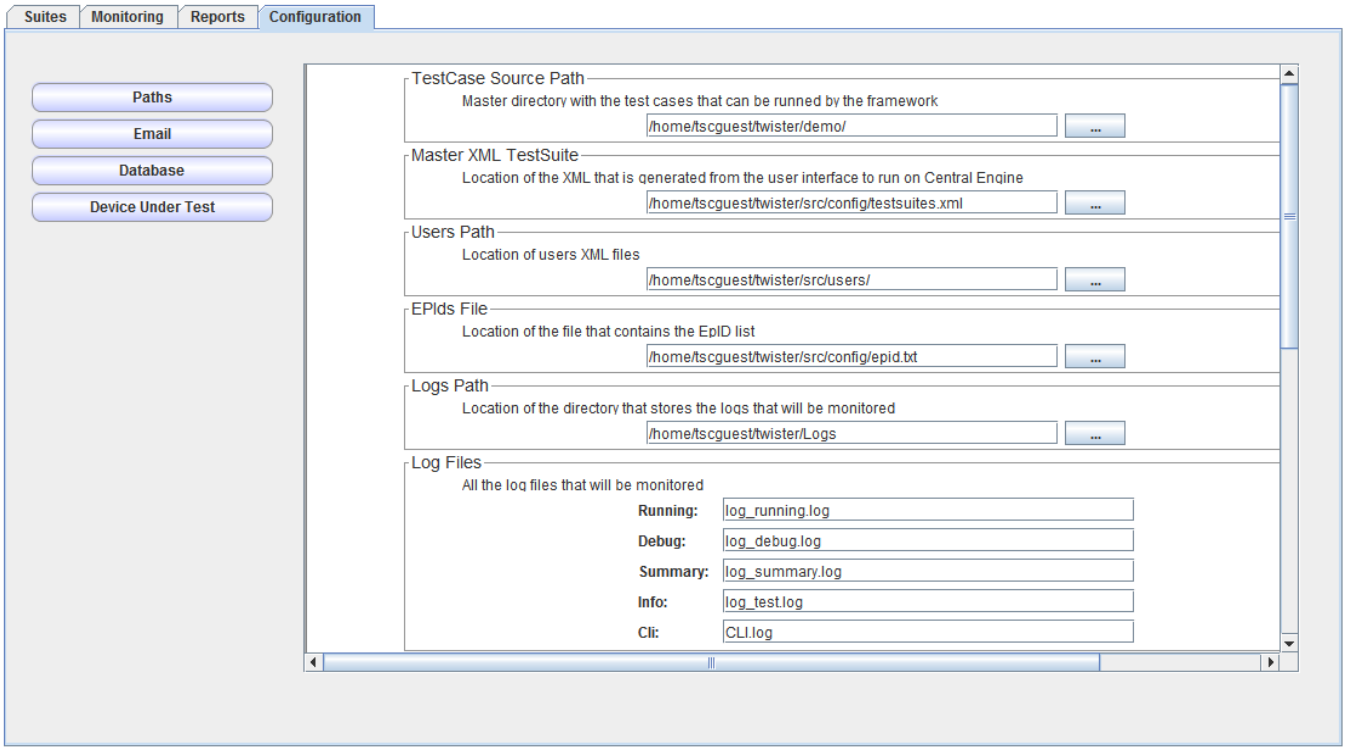
The same report, after the user chose the build

The **configuration tab**

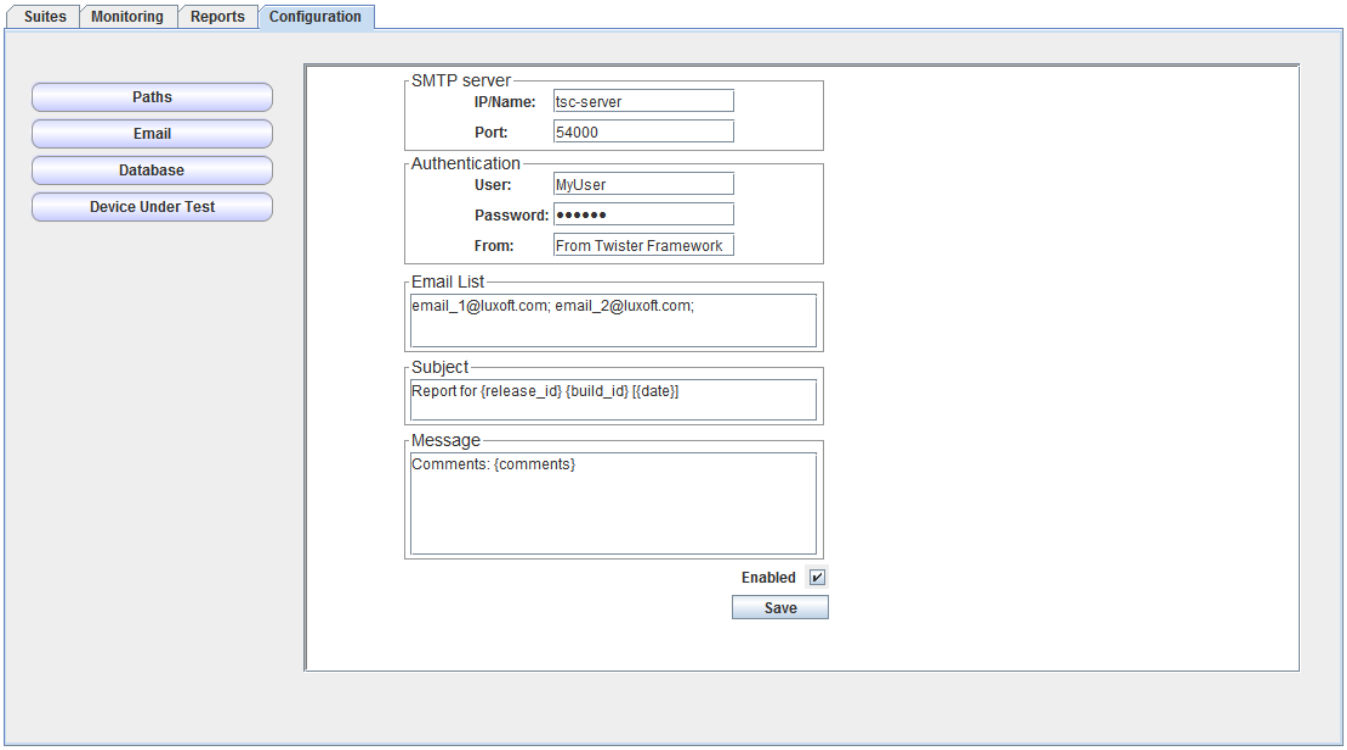
Here, you can configure:

- the port of Central Engine (*default 8000*), HTTP Server (*default 8080*) and Resource Allocator (*default 8001*);
- the path of the test files, logs files, user files;
- the path of the test suites xml, database xml, hardware config xml, EPs;
- the names of the log files;
- e-mail configuration;
- database configuration;
- devices configuration for resource allocator.

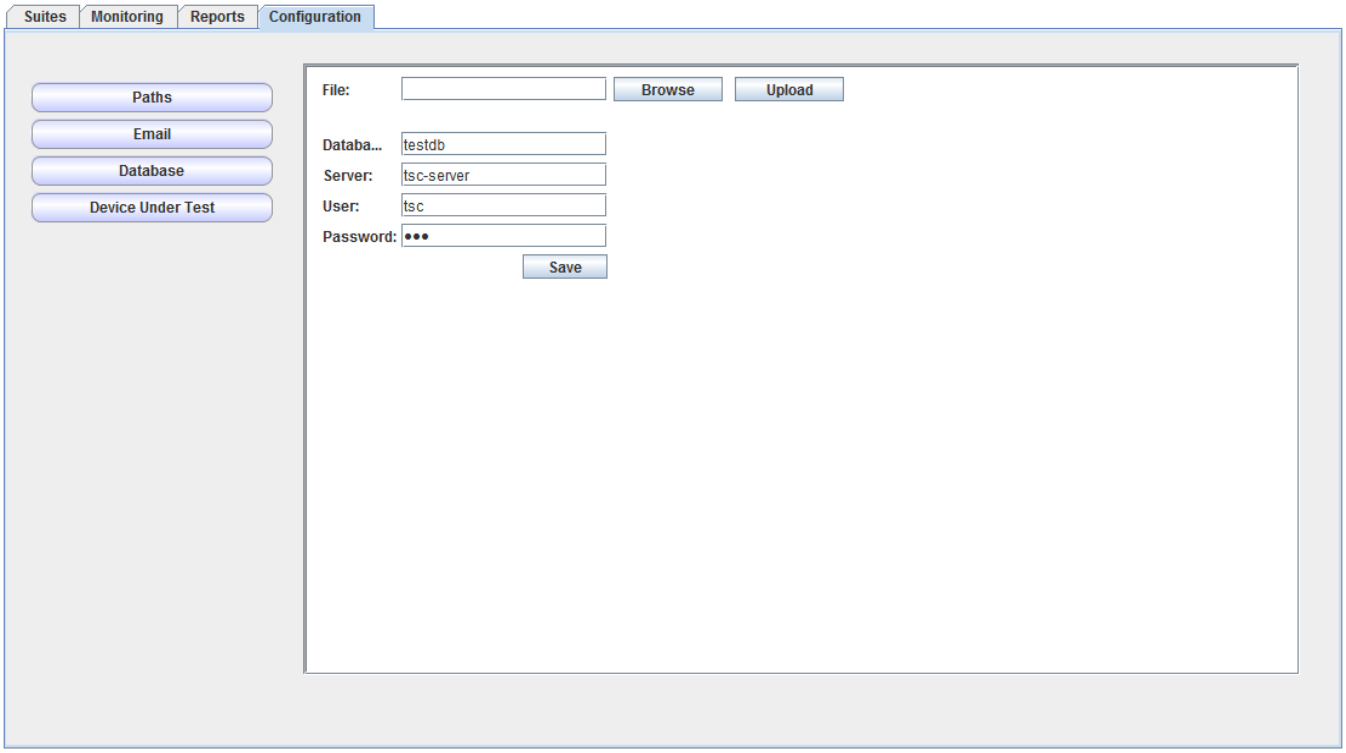
Print screen with the Paths configuration



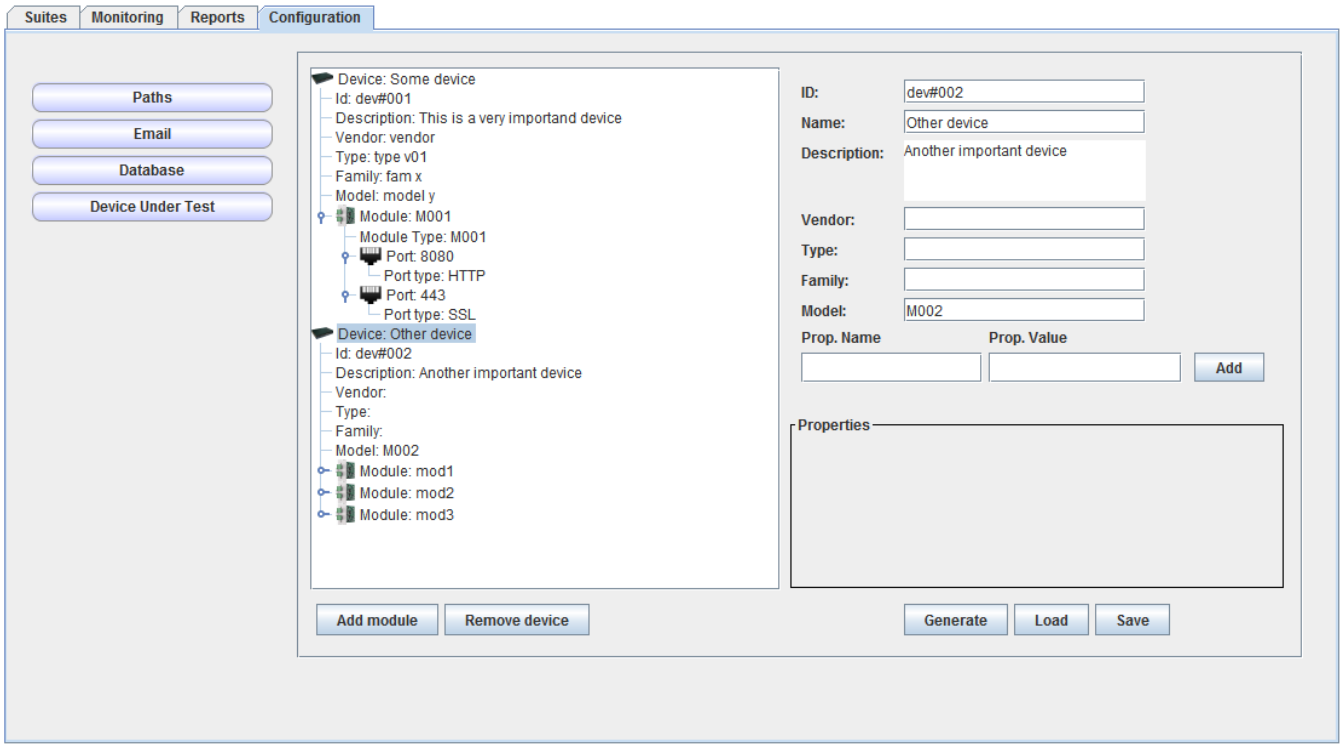
Print screen with the E-mail configuration



Print screen with the Database configuration



Print screen with the Devices configuration



6 – How to define plugins for Java GUI

Twister framework is designed to load user created plugins and display them in the main interface. In order for the framework to communicate with the plugin, the plugin must implement the TwisterPluginInterface interface found under *com.twister.plugin.twisterinterface* in the Twister.jar library.

When a new plugin is downloaded from the server it is automatically initialized by JVM, so, the initialization can't be controlled by the framework, this is the reason why the plugin should have an empty constructor. Instead, the initialization should be made in the init method.

- init(ArrayList<Item> suite, ArrayList<Item> suitetest, Hashtable<String, String> variables)

Init method accepts 3 parameters, references to variables found in Twister framework.

- ArrayList<Item> suite : reference to an ArrayList of Items defined by the user, also found under the Suites tab

- ArrayList<Item> suitetest : reference to an ArrayList of Items generated by the user, also found under the Monitoring tab

- Hashtable<String, String> variables: a Hashtable of String that points to diferent paths defined by the user.

The keys of the Hashtable are:

- user: framework user
- password: user password
- temp: temporary folder created by the framework
- Inifile: configuration file
- remoteuserhome: user home folder found on server

- remotconfigdir: config directory found on server
- localplugindir: local directory to store plugins
- httpserverport: server port used by EP to connect to a centralengine
- centralengineport: centralengine port
- resourceallocatorport: resource allocator port
- remotedatabaseparth: directory that contains database config file
- remotedatabasefile: database config file
- remoteemailpath: path to email configuration directory
- remoteemailfile: email configuration file
- configdir: local config directory
- usersdir: local directory to store suites configuration
- masterxmldir: local directory to store generated suite
- testsuitepath: remote directory that contains tc's for suite definition
- logspath: directory to store logs
- masterxmlremotedir: remote directory to store generated suite
- remotehwconfdir: remote directory to store hardware config file
- remoteepdir: remote directory to store EP file
- remoteusersdir: remote directory to store suites configuration

The framework calls terminate() method when the user wants to discard the plugin. The plugin should override the method terminate() to handle the release of all the resources. If some resources are not managed corectly, ex. threads will continue to execute after the call terminate(), these resources will continue to run in background.

The plugin should offer the framework a Component with the content that will be displayed on getContent() method. The framework will take that Component and put it under a new tab with the name provided by the plugin on the method getName(). The plugin should initialize the Component in the init method and should hold a reference to it so that it will serve the framework the same component every time getComponent() method is called.
The getFileName() method should return the name of the file that contains the plugin.
The plugin should be packed in a jar archive and uploaded in the Plugins folder found on server. The jar archive must contain a configuration file found in META-INF/services named com.twister.plugin.twisterinterface.TwisterPluginInterface.
This file contains a single line listing the concrete class name of the implementation, the plugin class name.

(More on http://java.sun.com/developer/technicalArticles/javase/extensible/index.html)

6.1– Plugin tutorial

For better understanding a brief tutorial for creating a small plugin will be presented. We will create a plugin to display the username found in the Hashmap and put it in a Jlabel. First include the Twister.jar library to your favourite ide. In this library we will find, besides the interface to implement, a base plugin class that eases the initialization process.

Create the class UserName that looks like this:

```
import com.twister.Item;
import com.twister.plugin.baseplugin.BasePlugin;
import com.twister.plugin.twisterinterface.TwisterPluginInterface;

public class UserName extends BasePlugin implements TwisterPluginInterface {

}
```

The BasePlugin holds the Hashtap in a variable named variables, the suite array in a variable named suite and the generated suite in a variable named suitetest. So, in order to get the username provided by the framework we will use variables Hastable, and put it in a Jlabel initialized in init.

```
import com.twister.Item;
import com.twister.plugin.baseplugin.BasePlugin;
import com.twister.plugin.twisterinterface.TwisterPluginInterface;

public class UserName extends BasePlugin implements TwisterPluginInterface {
private JPanel p;
private JLabel label;

	public void init(ArrayList<Item> suite, ArrayList<Item> suitetest,
	Hashtable<String, String> variables) {
		super.init(suite, suitetest, variables);
		p = new Jpanel();
		label = new JLabel(variables.get("user"));
	}
}
```

Observe how we are holding a reference to the Jpanel p because this is the component that we will serve to the framework.

```
import java.awt.Component;
import java.util.ArrayList;
import java.util.Hashtable;
import javax.swing.JLabel;
import javax.swing.JPanel;
import com.twister.Item;
import com.twister.plugin.baseplugin.BasePlugin;
import com.twister.plugin.twisterinterface.TwisterPluginInterface;


public class UserName extends BasePlugin implements TwisterPluginInterface {
	private JPanel p;
	private JLabel label;
```

```
	@Override
	public void init(ArrayList<Item> suite, ArrayList<Item> suitetest,
			Hashtable<String, String> variables) {
		super.init(suite, suitetest, variables);
		 p = new JPanel();
		label = new JLabel(variables.get("user"));
	}

	@Override
	public Component getContent() {
		return p;
	}
}
```

Let us provide a description, filename that contains this plugin, and the title of the plugin tab.

```
@Override
public String getDescription() {
	String description = "Plugin to display user name";
	return description;
}

@Override
public String getFileName() {
	String filename = "UserName.jar";
	return filename;
}

@Override
public String getName() {
	String name = "UserName";
	return name;
}
```

Also for consistency we should release the references on the terminate() method. In case we would have Threads running, we should terminate them here.

```
@Override
public void terminate() {
	super.terminate();
	p = null;
	label = null;
}
```

The final class should look like this:

```
import java.awt.Component;
import java.util.ArrayList;
import java.util.Hashtable;
import javax.swing.JLabel;
import javax.swing.JPanel;
import com.twister.Item;
import com.twister.plugin.baseplugin.BasePlugin;
import com.twister.plugin.twisterinterface.TwisterPluginInterface;

public class UserName extends BasePlugin implements TwisterPluginInterface {
```

```
	private static final long serialVersionUID = 1L;
	private JPanel p;
	private JLabel label;

	@Override
	public void init(ArrayList<Item> suite, ArrayList<Item> suitetest,
			Hashtable<String, String> variables) {
		super.init(suite, suitetest, variables);
		p = new JPanel();
		label = new JLabel(variables.get("user"));
		p.add(label);
	}

	@Override
	public Component getContent() {
		return p;
	}

	@Override
	public String getDescription() {
		String description = "Plugin to display user name";
		return description;
	}

	 @Override
	public String getFileName() {
		String filename = "UserName.jar";
		return filename;
	}

	@Override
	public void terminate() {
		super.terminate();
		p = null;
		label = null;
	}
}
```

We will pack this in an archive named UserName.jar. This archive must contain the folowing directory *META-INF/services*. In the services directory we must put a file named com.twister.plugin.twisterinterface.TwisterPluginInterface and, in this file we put the name of our plugin class UserName.

After we upload the UserName.jar file to the server Plugins directory, we should be able to download the plugin from Twister framework and activate it in the Plugins section.

7 - How to define the suites and add tests

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

After choosing the suites file, click on Add Suite. The required fields for a suite are: **the name** of the suite and **the EP** (the workstation where the tests from this suite will run).

Each suite can also have other fields, like `release`, `build`, `comments`, etc. These fields are defined in `DB.xml` file (*more about this in the database configuration section*).

Each suite must contain one or more unique script files, that will be executed on the EP.

The script files and the suites can be removed, or re-arranged, using drag & drop.

When the *Execution Process* is run, the scripts from suites are run in order. The execution doesn't stop if one of the scripts fails.

In order to save the **suites file**, click on File > Save suite XML.

After all the suites and scripts are defined in the correct order, click on Generate, to prepare for running the tests. Only the scripts that are checked will be exported for running.

8 - How to run the test files

In the second tab (*Monitoring*), you will see the files that were generated in the first tab.

There are two buttons: Run and Stop.

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 enable, 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 is successful;
- (failure) if the execution fails;
- (skip) if the file is marked as skip (*runnable=false*);
- (timeout) if the suite has timeout and the file was killed because of timeout;
- (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 result can be seen in the `*log_summary.log*` and the icon of the file will also change.

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

9 - How to configure the framework

In the configuration tab, there are 4 things you can configure: the paths, the e-mail, the database and the devices.

9.1 - Config 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).

Master XML is the file generated in the first tab (suites), and used in the second tab (monitoring). Usually, this doesn't need to be changed.

Users 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 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, 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 *Hardware config XML* 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*, *Resource Allocator port* and *HTTP Server port* are, of course, the ports to the respective servers. By default, the values are: *8000*, *8080* and *8001*, respectively.

9.2 - Config 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 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`, section name `field_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}]
```

So 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]
```

9.3 - Config 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.

In the root of the XML file, there are 2 sections: `*db_config*`, that is written by the interface and `*twister_user_defined*`, that has to be written manually.

The section `*twister_user_defined*` has 3 sub-sections: `*field_section*`, `*insert_section*` and `*reports_section*`.

The *field section* contains all the information that was defined in the **Suites tab** for each and every suite, things like: release, build, station, 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, that 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_ep_ip = the IP of the Execution Process;
- twister_ep_name = EP name, defined in **Suites tab**;
- 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 a 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;
- 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 defined report made in PHP.

Examples of redirects:

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

9.4 - Config the devices

There are 3 levels of depth for the devices:

- level 1 is *device*;
- level 2 is *module*;
- level 3 is *port*.

Each *device*, can have any number of *modules*; each *module*, can have any number of *ports*.

Each element can be define custom *properties* and save *values* for them.

After creating the tree of device elements, you must save it. You will be asked for a file name.

10 – Performance and troubleshooting

The Central Engine and the Resource Allocator 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 50 Execution Processes on a CE 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 are 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.