Twister Plugins Guide

Twister Libraries, Plug-ins, Services

* **Note**: Before reading this document, please consult `**Twister Guide.pdf**`.

Twister framework can be enhanced in many ways:
- with **Libraries** that will be sent to the client and used by the tests ;
- with Central Engine **Plug-ins** that expose even more functions via XML-RPC ;
- with **Services**, programs that run in the background and are managed by the Twister Service Manager.

Libraries

The libraries are Python or TCL files or folders that are downloaded by the Execution Process, before executing the tests.

The test files can import the libraries like this:

```
# In Python
import SomeLibrary

# In TCL
package require SomeLibrary
```

By default, Twister has a few libraries (only Python):

- **TscCommonLib.py**: This module contains common functions to communicate with the Central Engine. You can use : getGlobal, setGlobal, getResource, setResource, logMessage.
- **TscThreadsLib**.py : used by the tests to queue long running functions, then start all the queued functions at the same time ;
- **TscFtpLib**.py : wrapper over python default FTP library, allows creating connections, login, make folders, delete files and folders, etc ;
- **TscTelnetLib**.py: smart TELNET library that wraps python telnet connection from 'telnetlib' and offers a manager for custom telnet connections that ensures they are up and kept alive. The library learns the prompt for login on devices you use and can login automatically. The manager allows the user to open, list and close telnet connections and the other basic operations as read, write.
- **TscSShLib.py**: The SSH library is a file that is downloaded by the Execution Process, before executing the tests. It is a library that wraps python paramiko library and offers the user a manager for custom ssh connections. The manager allows the user to open, list and close ssh connections and the other basic operations as read, write.
- **TscStepLib**.py: used to create steps inside tests. The idea is to organize one test into smaller steps, which will be executed in order. Before each step, you can call other functions, to setup something before execution. After the step runs, all CLI log is searched using regular expressions and if any of the expressions is found, the matching functions are called.
- **TscUnitTestLib**.py: contains Setup and Teardown. All functions that begin with "test" will be executed automatically, in alphabetic order. Twister Test implements the same methods as Python Unit Test.

The *Python libraries* can be accessed only from *Python tests*, and the *TCL libraries* can be accessed only from *TCL tests*, but you can easily pass objects between languages, using **getGlobal** and **setGlobal** functions (available in both languages).

There is also the `**py_exec**` function, available only in the TCL tests.

For example, in a suite of TCL tests, you want to access the smart Python TELNET library.

In order to do that, create a prerequisite Python file, for example `**setup.py**`, that creates a Telnet object and stores it in a global parameter:

```
# setup.py file
import TelnetManager

tm = TelnetManager()

# Create a connection called `connection1`
tm.open_connection('connection1', '127.0.0.1', 23, 'user', 'password', 'login:', 'Password')

setGlobal('tm', tm) # Save the python object in a variable, shared between tests for this EP
```

Then, you can call the Python object from all TCL tests like this:

```
# TCL test
py_exec tm.list_connections()
py_exec tm.write('ls')
py_exec tm.read()
```

The `**py_exec**` function can all any Python function or object from within TCL tests.

Ixia Library is a wrapper in Python, using TCL functions from IxTclHal.
The list of functions supported by the Ixia library can be found in lib/Ixia/funxtions.txt
Examples of using the Ixia library can be found in lib/Ixia/TscIxPythonLib.py at the end of the file.

A short example of how to use the library:

```
# Import Ixia library
import TscIxPythonLib.py

# Ixia commands examples

# Connect to Ixia chassis
print 'Connect to TCL Server:', ixConnectToTclServer("10.144.31.91")
print 'Connect to Chassis', ixConnectToChassis("10.144.31.91")

# Printing chassis config
print 'Config chassis...'
portList = ''
chassis('get 10.144.31.91')
py_chassis = chassis('cget -id')
print py_chassis

# Configuring an Ixia card
print 'Config card...'
py_card = 1
card('setDefault')
card('config -txFrequencyDeviation 0')
print py_card
```

The **SSH library** is a file that is downloaded by the Execution Process, before executing the tests.
It is a library that wraps python paramiko library and offers the user a manager for custom ssh connections.
The manager allows the user to open, list and close ssh connections and the other basic operations as read, write.

The test files can import the libraries like this:

```
import SshManager
```

Following are the parameters available for the Ssh library.
You can find an example of how to use it in ~/twister/demo/testsuite-python/ test_py_ssh_twist.py

- parameters:
  name = connection name;
  host = connection host;
  user = username for connection login;
  password = username password for connection login;
  port = connection port is default 22;
  command = string command to pass on telnet connection;
  timeout = connection timeout;

- open_connection:
  - mandatory parameters: name, host;
  - optional parameters: user, password, port;

- write:
  - mandatory parameters: command;
  - optional parameters: name;
If no name is supplied it will try to use the active connection otherwise returns False. If success returns the connection output.

- read:
  - mandatory parameters: None;
  - optional parameters: name;
If no name is supplied it will try to use the active connection otherwise returns False. If success returns the connection output.

- set_timeout:
  - mandatory parameters: timeout;
  - optional parameters: name;
If no name is supplied it will try to use the active connection otherwise returns False. If success returns True.

- get_connection:
  - mandatory parameters: None;
  - optional parameters: name;
If no name is supplied it will try to use the active connection otherwise returns False. If success returns a SshConnection instance (custom ssh connection).

- set_active_connection:
  - mandatory parameters: name;
  - optional parameters: None;
If success returns True otherwise False.

- list_connections:
  - mandatory parameters: None;
  - optional parameters: None;
Returns a list of connection names.

- close_connection:
  - mandatory parameters: None;
  - optional parameters: name;
If no name is supplied it will try to use the active connection otherwise returns False. If success returns True.

- close_all_connections:
  - mandatory parameters: None;
  - optional parameters: None;
If success returns True otherwise False.

Plug-ins

Twister plug-ins are of 2 types: **python** and **java**. Each user must configure the plug-ins on its own, the configuration is not shared for more users.

The Python plug-ins are ran by the Central Engine, each plug-in can be called with:

```
runPlugin( 'user', 'plugin name', parameters as String or Dictionary )
```

The Java plug-ins are ran by the applet and each plug-in is a new tab.

By default, Twister has a few plug-ins:

- **Git** Plugin (python and java)
- **SVN** Plugin (python and java)
- **Jenkins** Plugin (python and java)
- **JIRA** Plugin (python and java)
- **PacketSniffer** (packets Twister) (python and java): captures network packets and list them to users, allowing them to filter the packets as they are captured, or after capture. All in all this plugin has functionality similar to Wireshark.
- **Scheduler** Plugin (just Java): allows starting the Central Engine automatically, either one time, daily, or weekly. This Java plug-in controls the Scheduler Service (more about this in the Services section).
- **ServiceConsole** (just Java): ServiceConsole plugin it helps you see the logs of the services.

The plug-ins configuration for each user is in `*/$USER_HOME/twister/config/plugins.xml*`.

To start using a plugin, after you install it (described for every plugin below), you must access Configuration -> plugins tab.

Here you must download and then activate a plugin to start using it.

As you can see from the screenshot below in the right window you can see the list of downloaded Plugins (with a short description) which you can activate.
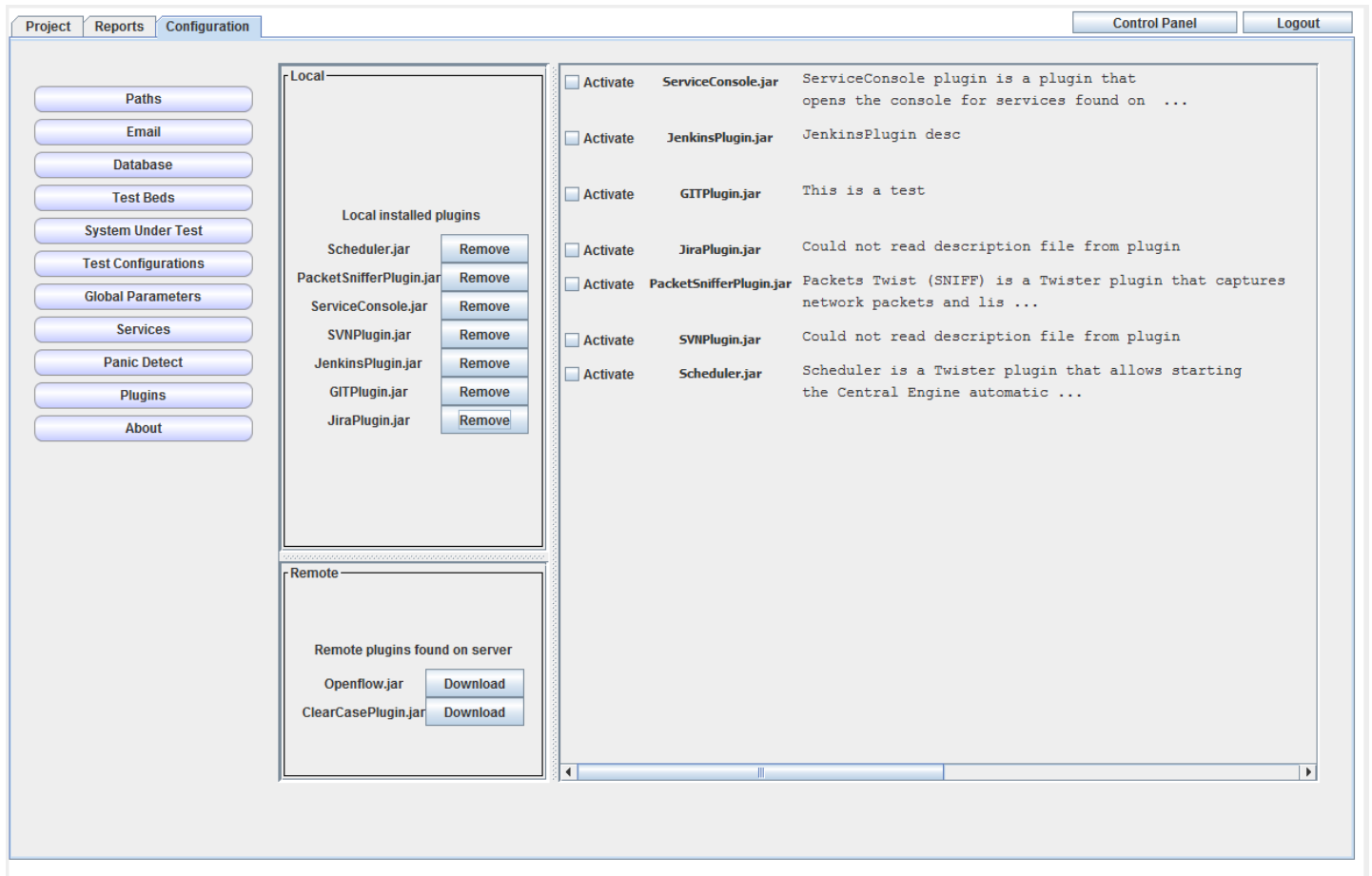
Git Plugin

Git Plugin is an interface for connecting to a Git repository where the tests are located and it will replicate the content locally.

After you download and activate the plugin a new tab will appear called GITPlugin. You can configure the how to connect to the repository.

If you click the Snapshot button it will create a clone locally and if you click Update after you first create a clone it will pull the last version of your tests.

Be aware that in order to use the Git folder as your main directory for your tests then you have to configure the same directory in Configuration -> Paths -> TestCase Source Path

How to install:

```
Default:
 - twister server side path: /opt/twister
 - twister client side path: /home/user/twister


      ---- SERVER side ----

sudo cp /path/to/GitPlugin/Git/* /opt/twister/plugins
```

GIT Plugin has a few parameters:
   - server complete path
   - branch used for clone
   - user and password to connect to server
   - snapshot folder, where all data is cloned

If command is Snapshot, execute a GIT clone;
     if the Snapshot folder is already present, delete it, then GIT clone.

If command is Update and Overwrite is false, execute a GIT pull;
    if Overwrite is false, delete the folder, then GIT clone.

Screenshot of the configuration tab:
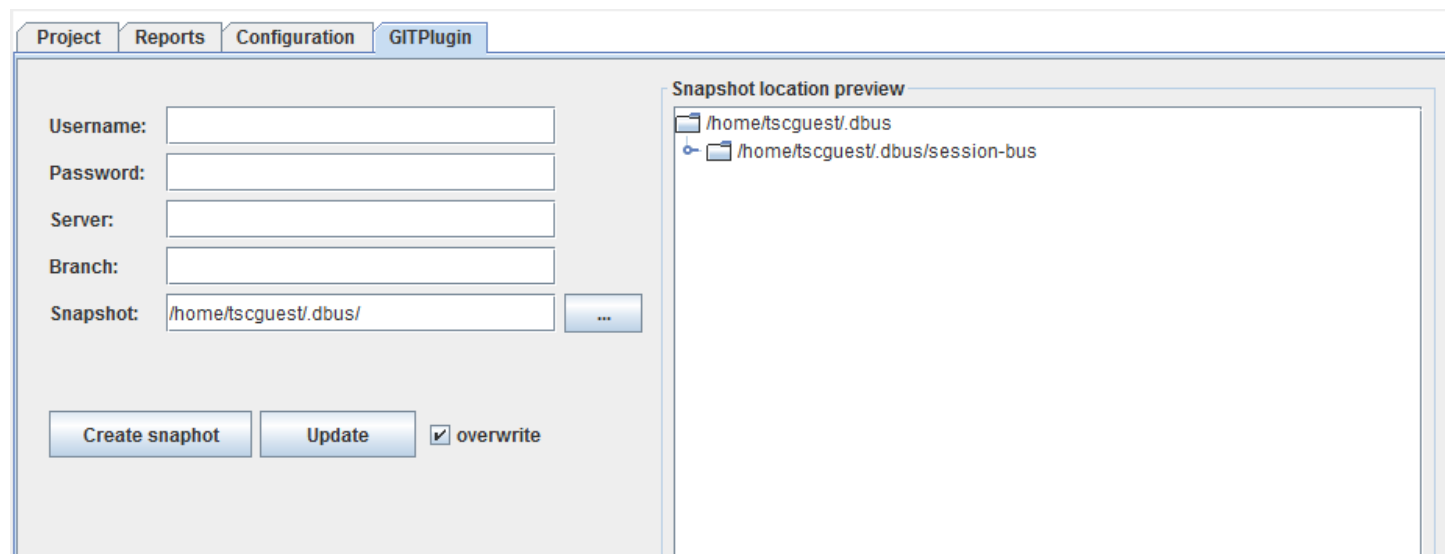
SVN Plugin

SVN Plugin is an interface for connecting to a SVN repository where the tests are located and it will replicate the content locally.

After you download and activate the plugin a new tab will appear called SVNPlugin. You can configure the how to connect to the repository.

If you click the Snapshot button it will create a clone locally and if you click Update after you first create a clone it will pull the last version of your tests.

Be aware that in order to use the SVN folder as your main directory for your tests then you have to configure the same directory in Configuration -> Paths -> TestCase Source Path

How to install:

```
Default:
 - twister server side path: /opt/twister
 - twister client side path: /home/user/twister


       ---- SERVER side ----

sudo cp /path/to/SvnPlugin/Svn/* /opt/twister/plugins
```


SVN Plugin has a few parameters:
- server complete path
- user and password to connect to server
- snapshot folder, where all data is cloned

If command is Snapshot, execute a SVN checkout;
    if the Snapshot folder is already present, delete it, then SVN checkout.
If command is Update and Overwrite is false, execute a SVN update;
    if Overwrite is false, delete the folder, then SVN checkout.

Screenshot of the configuration tab:

Jenkins Plugin

Jenkins Plugin is an interface for connecting to a Jenkins continuous integration server.
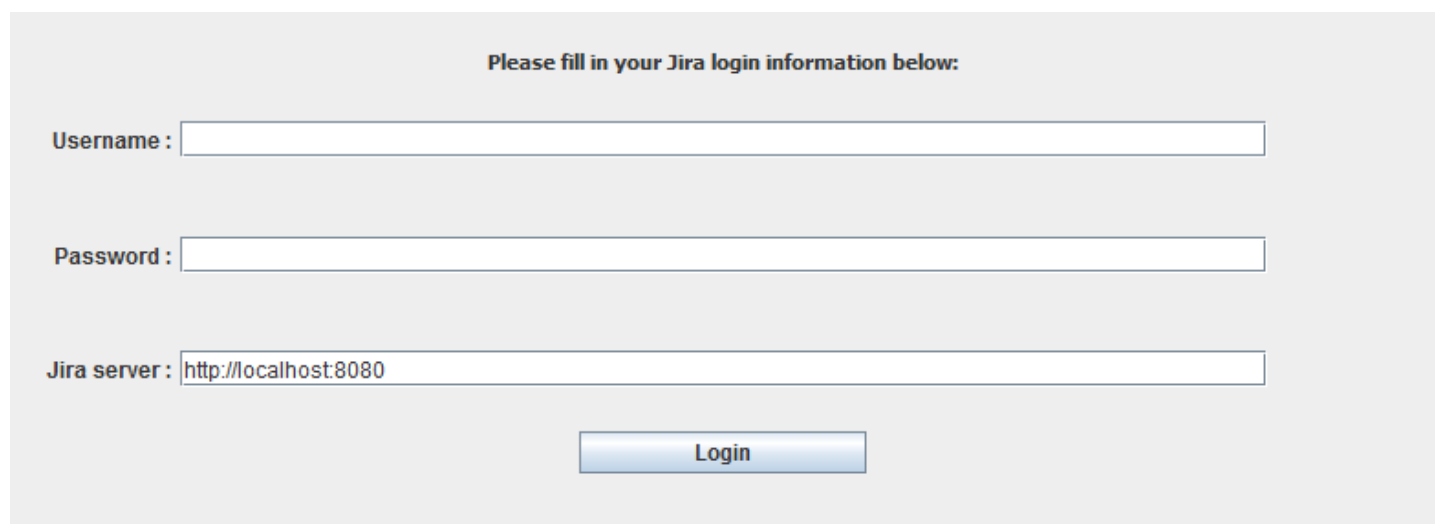
How to install:

```
Default:
 - twister server side path: /opt/twister
 - twister client side path: /home/user/twister


      ---- SERVER side ----

sudo cp /path/to/JenkinsPlugin/Jenkins/* /opt/twister/plugins
```

How it's working:
After a Jenkins job it's done it will execute the Jenkins Post Script.
This script will verify if the build finished successfully. If yes, it tells the Twister Plugin that the build is ready.
The plugin will run first the build script in order to upload the build on the DUT (Device Under Test).
After will start to run the tests previously saved in a Project File.
After execution it will save the result in the database if needed.

Jenkins Post Script (Jenkins_Post_Script.py) : after the Jenkins job is done this script verifies if the job/build it finished successfully.
If yes then it tells to the Twister Plugin that the build is ready. The Jenkins_Post_Script.py file it's located in /plugins/Jenkins folder.
You will need to install the Hudson Post build task plugin (http://wiki.hudson-ci.org/display/HUDSON/Post+build+task) in Jenkins.

Twister Plugin : it runs the Build script that uploads the build to the DUT(Device Under Test) and it runs the tests previously saved
in the Project File(XML file). After the execution it saves the results to the Database if needed. When you enable the plugin from
the interface you have to setup the Build Script and the Project File.

Build Script : It uploads the build to the DUT (Device Under Test).

You will have to edit the following files to match your configuration:
- Jenkins Post Script: enter the name of your Job in line 6
- Build Script: this script can be bash, python, tcl
- Project File (XML file): if you want to save the results in the database after test running you have to set dbautosave value as true.

Screenshot of the configuration tab:

Jira Plugin

Jira Plugin is an interface for connecting to a Jira server where you can create, edit , see comments and search for issues.

How to install:

```
Default:
 - twister server side path: /opt/twister
 - twister client side path: /home/user/twister


        ---- SERVER side ----

sudo cp /path/to/JiraPlugin/Jira/* /opt/twister/plugins
```

In order to use it, after you download and activate the plugin you must enter the details to connect to the server.

Screenshot of the configuration tab:

Screenshot of creating an issue:

Screenshot of the main window of the Jira Plugin:

PacketSnifferPlugin

The PacketSniffer Plugin is a network packet analyzer. It will try to capture the networks packets and to display that packet data.

How to install:

```
Default:
 - twister server side path: /opt/twister
 - twister client side path: /home/user/twister


       ---- SERVER side ----

1. sudo cp /path/to/PacketSnifferPlugin/PacketSnifferPlugin/* /opt/twister/plugins


       ---- CLIENT side ----

1. cp -R /path/to/PacketSnifferPlugin/PacketSniffer /path/to/twister/services

2. cp /path/to/PacketSnifferPlugin/start_packet_sniffer.py /path/to/twister/bin
```

After you download the plugin (Configuration -> Plugins tab) and activate it a new tab will show up called **PacketSnifferPlugin**. You have to click Start on this page before running your test in order to capture the packets.

Following are details of the PacketSniffer Plugin and how to run it.

Packet Sniffer contains four files: - '~/twister/bin/start_packet_sniffer.py'
                           - '~/twister/services/PacketSniffer.py'
                           - '~/twister/services/PacketSnifferClasses.py'
                           - '/opt/twister/plugins/PacketSnifferPlugin.py'

""" ~/twister/bin/start_packet_sniffer.py """

   This is the start script for Packet Sniffer service. You need sudo permissions to execute it and is mandatory to specify '--twister_path' (client's Twister path) and '--user' (client's username) parameters. At launching the script imports PacketSniffer class from '~/twister_path/services/PacketSniffer/PacketSniffer.py' and parses '~/twister_path/config/epname.ini' and initiates an PacketSniffer class instance and after calls the class 'run()' method.
   The '~/twister/bin/start_ep.py' script kills all the Sniffer processes before launching the EP instance that handels the start of the Sniffer process which can be triggered from the plugin interface.

""" ~/twister/services/PacketSniffer.py """

   This file contains the PacketSniffer class implementation. It uses two other classes: Sniffer and ParseData.
   The Sniffer class inherits Automaton class from scapy library and deals with the storing of intrenet packets in a list.
   ParseData class treats the CE status, sniffing filters and packets sending. It checks the CE status, updates the Sniffer filters and it pops out the packets stored in the list by Sniffer, parse them and sends them to CE PacketSnifferPlugin.

""" ~/twister/services/PacketSnifferClasses.py """

   In this file is implemented the class for the OpenFlow protocol detection. This class inherits Packet class from scapy and treats all the fields that can be present in a OpenFlow packet. It maps the captured packet data on the defined model.

""" /opt/twister/plugins/PacketSnifferPlugin.py """

   This is the Sniffer plugin class which is locataed on CE. It makes the connection between the sniffer service on client machine and Twister interface. The plugin holds the packets, deals with saving them in '.pcap' format and connects the commands from the interface with the Sniffer service on client machine.

Following is a screenshot of PacketSniffer plugin in action.

Scheduler Plugin

Scheduler Plugin is a plugin used to start automatically a set of tests at a certain date and time (one time, daily or weekly).

In order to use the Scheduler Plugin you must first of all start the scheduler service in Configuration -> Services.

After that download and activate the plugin. After that a new tab will appear called Scheduler. In that tab you can configure a date and hour when your tests will start automatically (one time, daily and weekly).

How to install:

```
Default:
 - twister server side path: /opt/twister
 - twister client side path: /home/user/twister
```

```
        ---- SERVER side ----
sudo cp /path/to/SchedulerPlugin/Scheduler/* /opt/twister/plugins
```

Screenshot of the configuration tab:

ServiceConsole Plugin

ServiceConsole Plugin it helps you see the logs of the services.

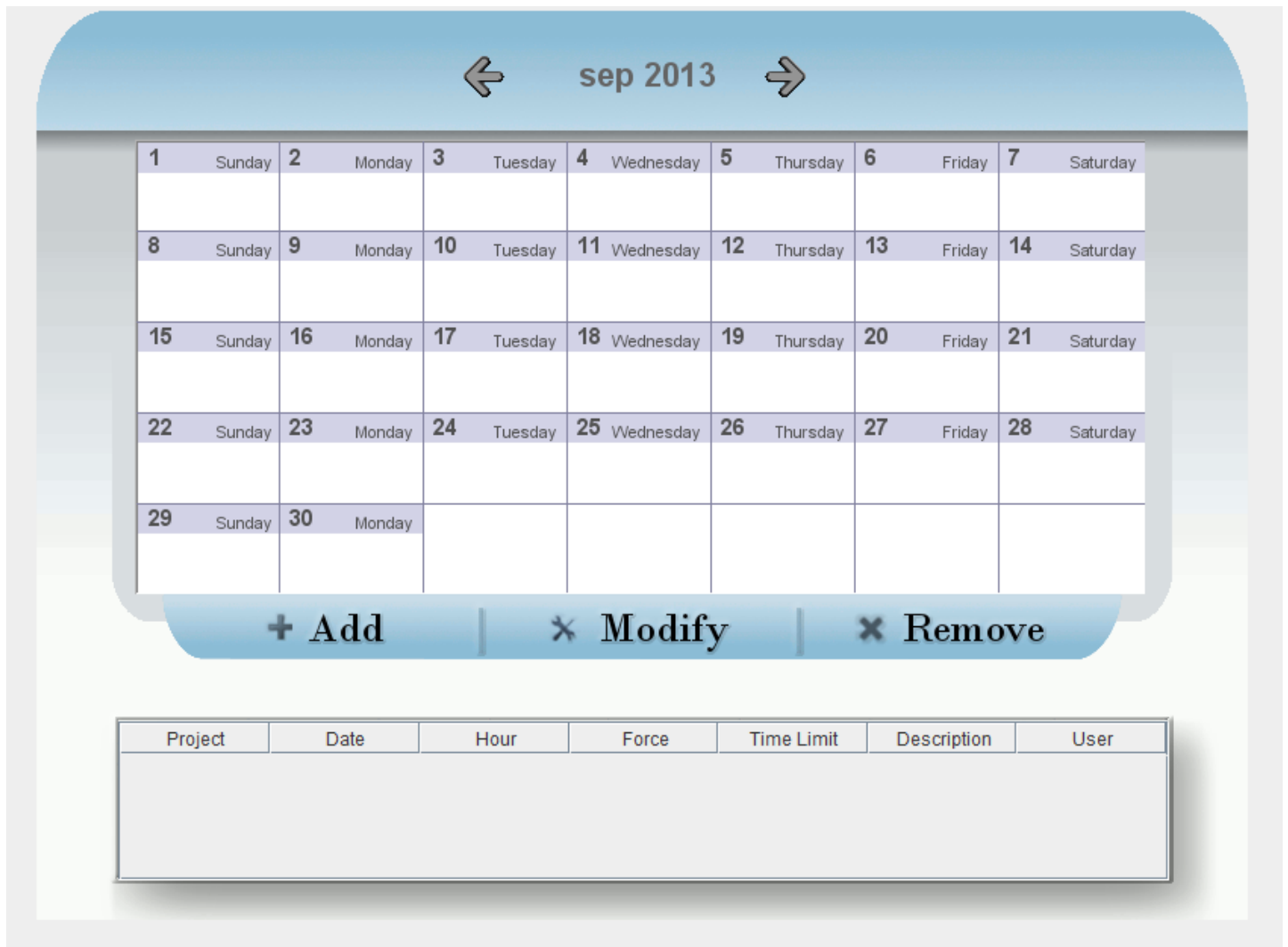In order to use you must download and activate the plugin from Configuration -> Plugins.
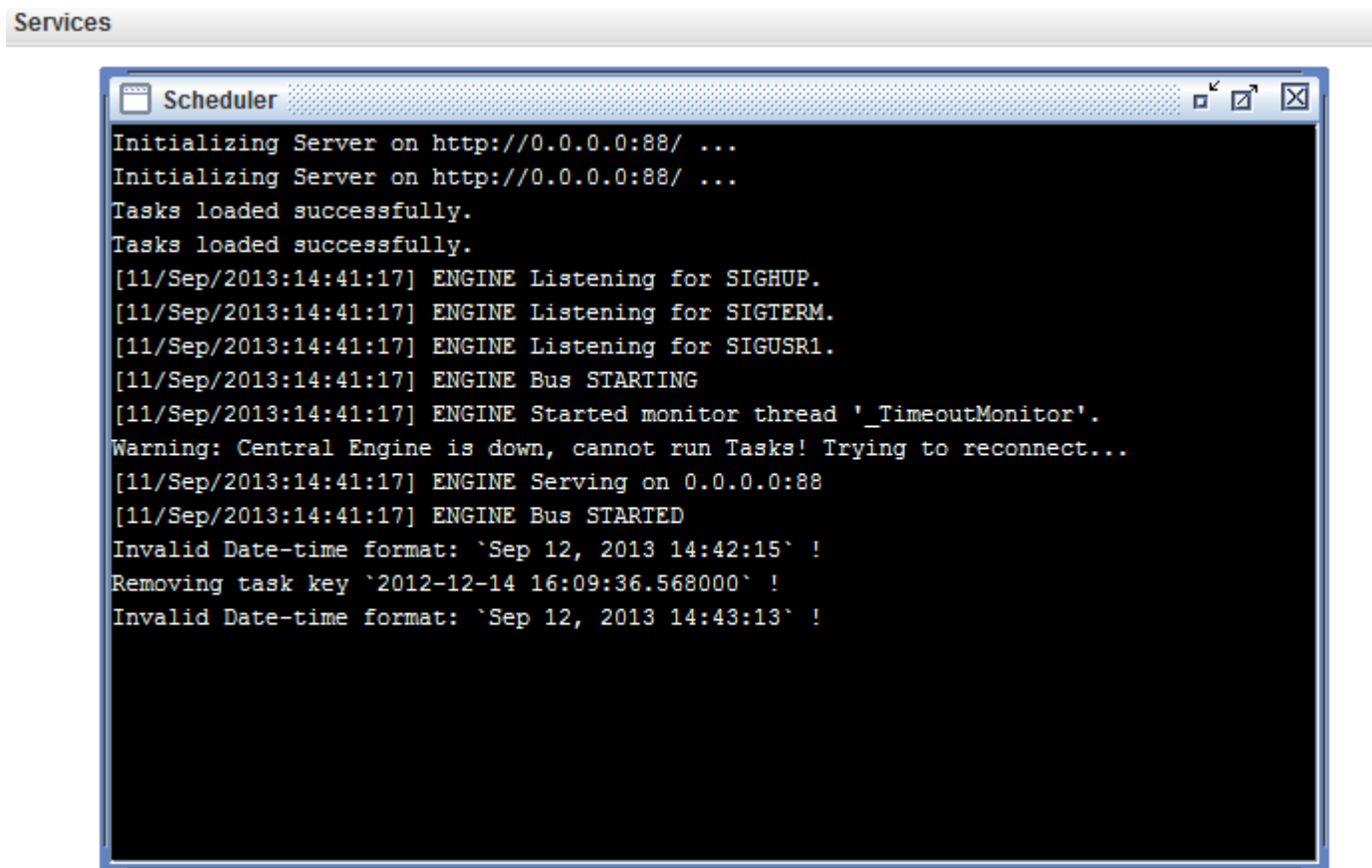
How to install:

```
Default:
 - twister server side path: /opt/twister
 - twister client side path: /home/user/twister


        ---- SERVER side ----
sudo cp /path/to/ServiceConsolePlugin/ServiceConsole/* /opt/twister/plugins
```

Screenshot of the ServiceConsole plugin in action:

ClearCase Plugin

Twister framework offers integration with ClearCase through a built-in plugin.

IMPORTANT: The clearcase environment **MUST** be set on the Twister server. The plugin applies for dynamic views. The following record MUST be added into the client's configuration file plugins.xml:

```
<Plugin>

    <name>ClearCase</name>

    <jarfile></jarfile>

    <pyfile>ClearCasePlugin.py</pyfile>

    <status>enabled</status>

</Plugin>
```

The ClearCase plugin will allow user to have access to test cases hosted in a ClearCase repository. The plugin must be activated from Configuration->Plugins section. Since it is a built-in component, there is no need to download anything.

When the ClearCase plugin is activated, a new tab appears in Twister.
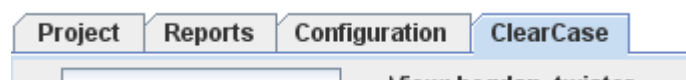
The ClearCase plugin can be used to list the views, to set a view, to show the config spec for a view or event to create a new view.

To show the available views, you press the "List Views" button and the list of available views is displayed.
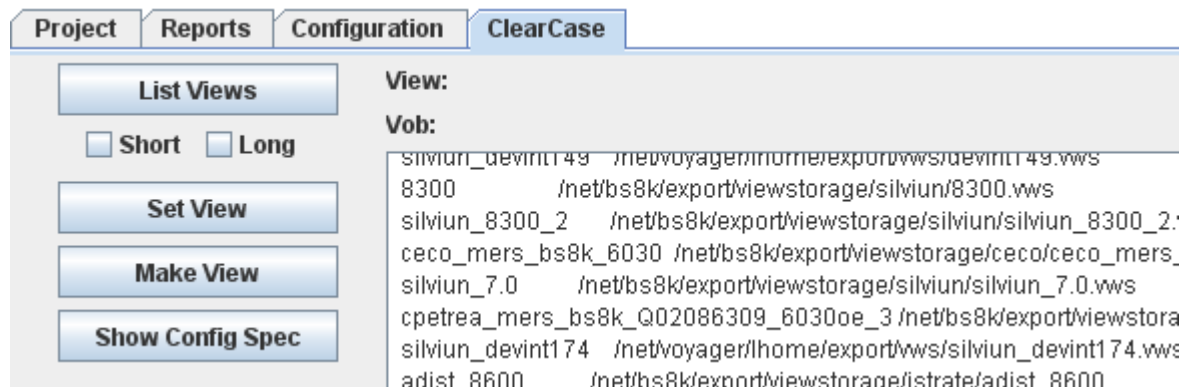
You can specify the short or the long flags to list the available views.

To select a view that you want to set, press the "Set View" button. A list with available views is displayed.
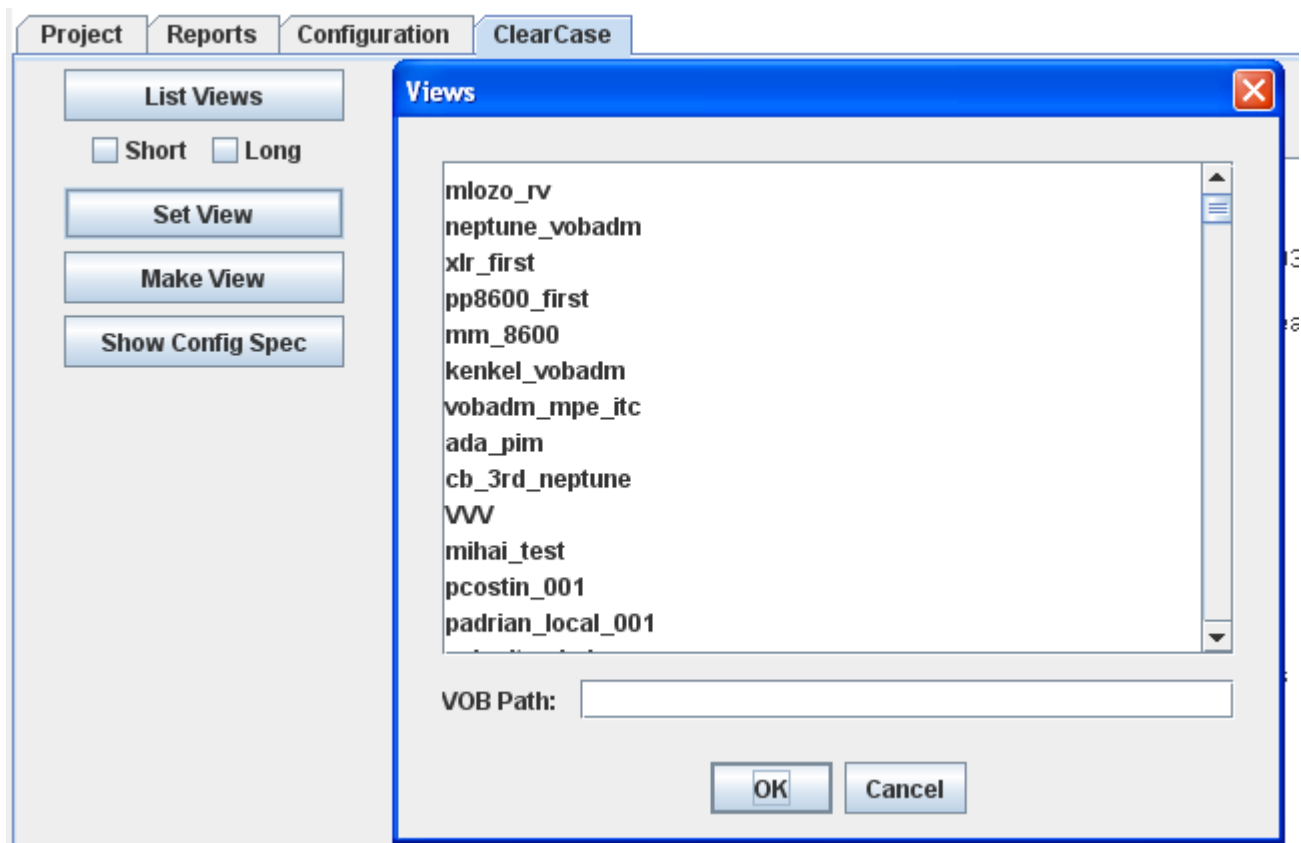
You must select a view from the list. Because a view can use multiple VOB's, you must introduce manually the VOB path. Once, the view and VOB paths are selected, the information will be displayed in the ClearCase plugin.

If a view was selected, you can see the config spec of the view by pressing the "Show Config Spec" button.
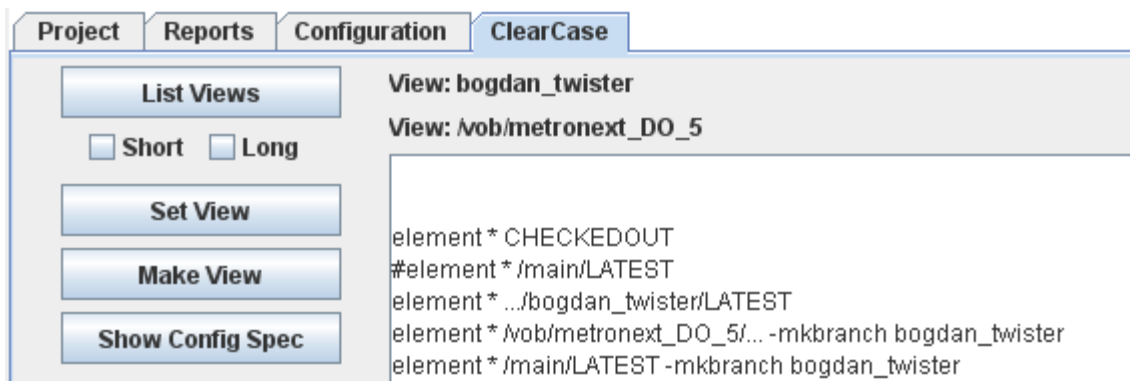
You can create a new view, by pressing the "Make View" button.
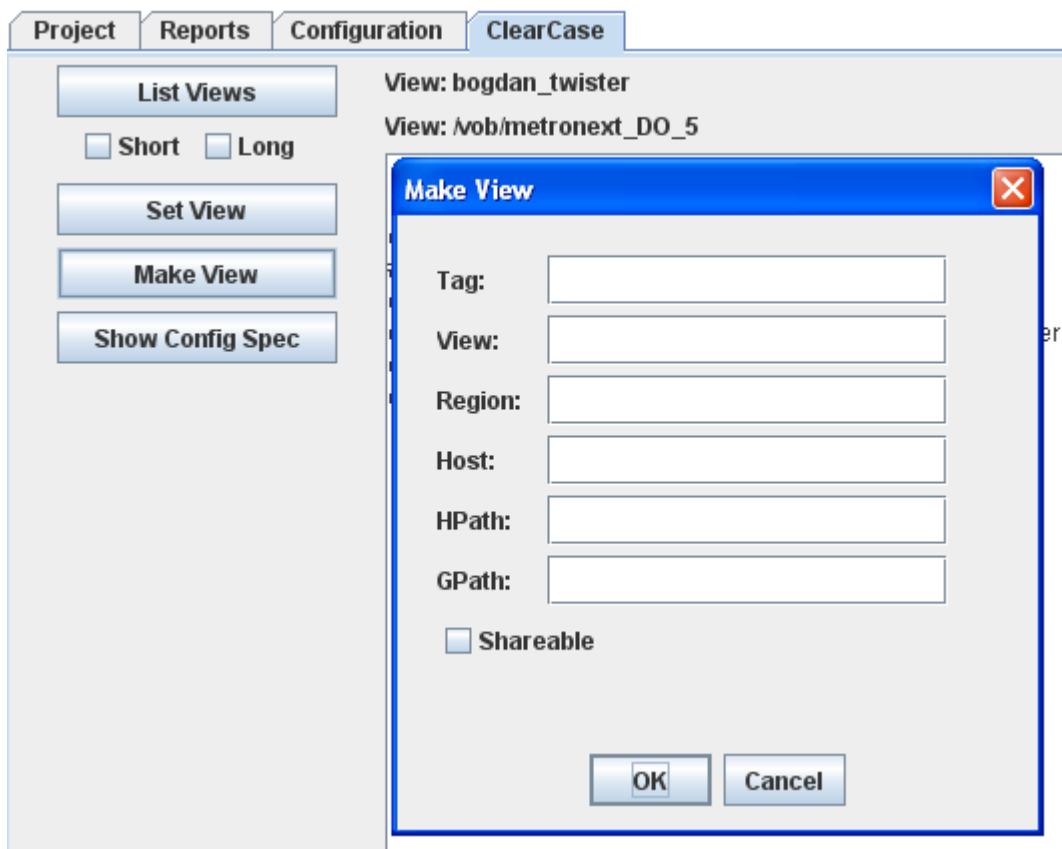
The Tag and View fields are mandatory to create a new view.

When the ClearCase plugin is activated and a view is selcted, the Project workspace is changed to show the list of test cases available in the selected view.
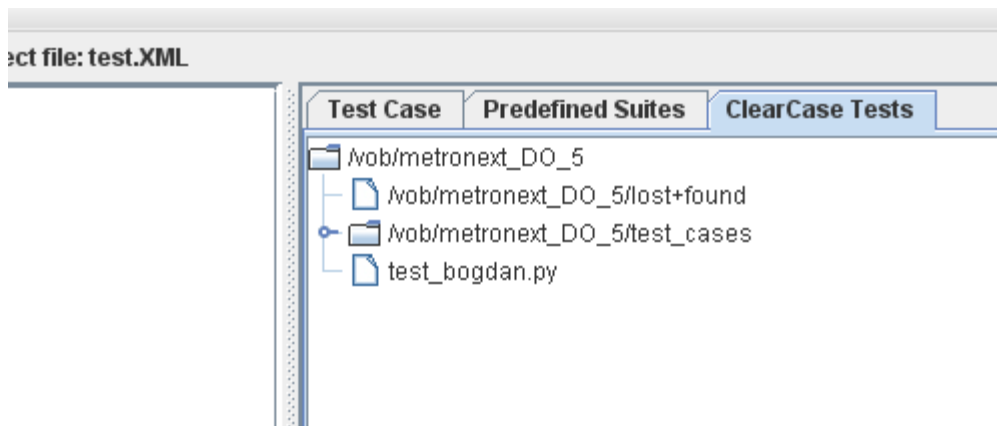
If a test case is not checked out and you right click on it, you can view or check out the file.
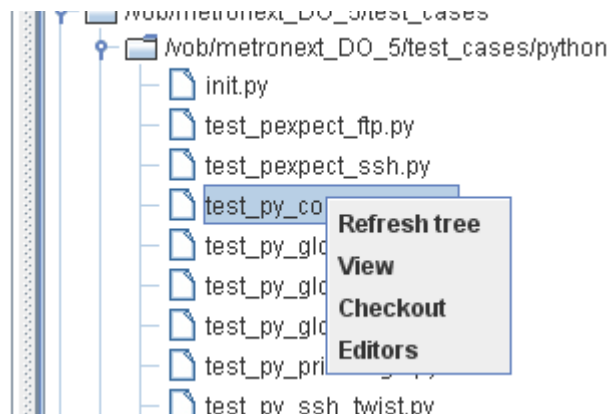
If you check out a file, you are prompted to add a comment for checkout.

If a test case is checked out and you right click on it, you can edit it, checkin or uncheckout the file.
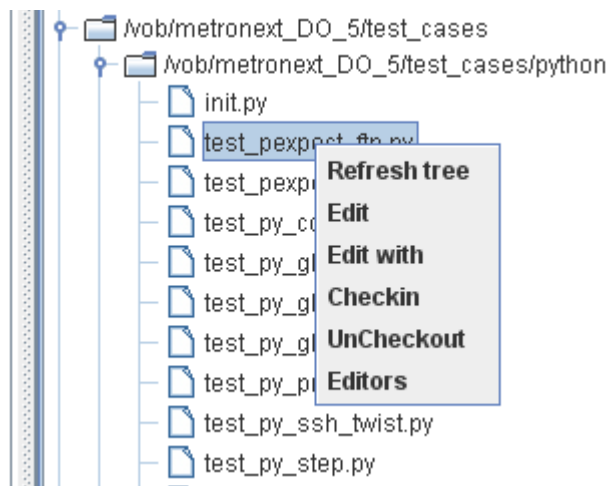
If you checkin a file, you are prompted to add a comment for checkin.

The test cases from ClearCase can be added in test suites by drag&drop, like regular tests.

How to define plug-ins for Java GUI

Twister framework is designed to load user created plug-ins and display them in the main interface, as a new tab.

In order for the framework to communicate with the plug-in, it must implement the TwisterPluginInterface interface found under com.twister.plugin.twisterinterface in the Twister.jar library.

When a new plug-in 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 plug-in 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 : ref to an ArrayList of Items defined by the user, also found under the Suites tab ;
- ArrayList<Item> suitetest : ref 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*)

Plug-in tutorial (Java)

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:

```java
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.

```java
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"));
      }
}
```

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

```java
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's provide a description, filename that contains this plugin, and the title of the plugin tab.
By default the plugin looks for a filename named: filename_description.txt where "filename" is the string returned by getFilename() without the ".jar" ending(EX.:UserName_description.txt). The framework downloads this file from the plugins directory on server if it finds it. We will override this method and provide one of our own.

```
@Override
public String getDescription(String localplugindir) {
        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 META-INF/services directory. 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 plug-in class UserName.

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

Plug-ins for Python

Python plug-ins should implement ***additional*** methods necessary to communicate with the Java interface, or with a service that you created. If the default Central Engine methods are sufficient, you don't need to implement a Python plug-in.

You can implement a Python plug-in without a paired Java plug-in. In this case, you will call the new methods using the `***runPlugin***` function from Central Engine, via XML-RPC API.

The file(s) should be placed on the server side, in the `/opt/twister/**plugins**`. Typically, you should name the main Python file the same as the Java plug-in file (ex: GitPlugin.java and GitPlugin.py), if you make a paired plug-in.

You can store each plug-in in a separate folder, you just need to specify the paths in `plugins.xml`, from `/$USER_HOME/twister/config` folder. Note that the plug-ins are shared for all users, but each user has a different configuration! For example, a user chooses to ignore all the plug-ins, another user might use a few plug-ins, but with different paths from a third user.

All Python plug-ins must import the `**BasePlugin**` class, from `/opt/twister/plugins/BasePlugin.py` file. All inherited functions can be re-written.

A plug-in is instantiated in the CE memory the first time it is used and it will be deleted only when the CE stops.

In order to be executed, the plug-in must implement only the `**run**` function; it's the only function called automatically. The `run` function receives only one argument, a dictionary, containing all commands received from the paired Java plug-in, or from another XML-RPC client that executed `***runPlugin***`.

Example: For Git Plugin, `run` can have the argument like:

- {'command': 'snapshot', 'src': '/home/user/src', 'dst': '/home/user/dst'}, OR
- {'command': 'update', 'overwrite': 'false', 'src': '/home/user/src', 'dst': '/home/user/dst'}, OR
- {'command': 'delete', 'src': '/home/user/src', 'dst': '/home/user/dst'}.

In every case you should implement the methods to make it happen.

Services

Services are long running programs, managed by the Twister Service Manager.

They can be controlled from within the applet, in Settings > Services tab.

By default, Twister has 1 service:

- **Scheduler**: is a Python service that starts the Central Engine automatically, either one time, daily, or weekly. It is controlled using the Java Scheduler plug-in.