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
from ce_libs import SomeLibrary

# In TCL
package require SomeLibrary
```

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

- **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.
- **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
from ce_libs 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.

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)
- **Sniffer** (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).

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

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:

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

Notice 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'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.

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.

The end