Ockle Documentation

Release 0.5.0

Guy Sheffer

November 02, 2012

CONTENTS

Ockle is a tool which lets you control a group of power distribution units (PDUs) and the servers which connected to them. Servers can be dependent on each other, and Ockle can then determine which servers should be turned on according to those dependencies. After server is turned on Ockle can run automated tests to make sure they indeed provide the services that are required by the servers.

CHAPTER

ONE

DESIGN PRINCIPLES IN OCKLE

- *Extensibility* – I tried to implement the method “everything is a plugin”, by this I mean that every new form of logic or functionally could be added and removed from the configuration without changing the code itself. Every new feature would go in to its own module and process thread.
- *Lightweight* – Ockle is split to a control daemon and a webserver, so the device controlling the servers could be put on a embedded device on a separate power supply.
- *Easy to use* – The webserver aims to give an intuitive user experience, with helpful information about the server’s health and power usage status.

CHAPTER

TWO

WHERE TO GET OCKLE

Ockle is available at GitHub.

You can download it by cloning it:

```
git clone https://github.com/guysoft/Ockle.git
```

2.1 Ockle is Free Software

This software is distributed under the GNU General Public License, version 2

CHAPTER

THREE

USER MANUAL

3.1 Installing Ockle

Note: It is recommended to run Ockle in a virtualenv. This is so upgrades of your system won't break any control over your servers. So first make sure you have it.

- Installing virtualenv:

```
apt-get install python-virtualenv
```

3.1.1 Set up the python environment

- In order to compile some of the python modules you will need to install the following packages (or your distro's equivalent)

```
apt-get install libxslt1-dev libxml2-dev libgraphviz-dev
```

- Run the following commands to get a python environment with the correct modules and version.

Note: you can change *~/pythonenv* to any path that suits you

```
python2.7 /usr/bin/virtualenv ~/pythonenv
~/pythonenv/bin/easy_install pyramid==1.2.7
mkdir ~/pythonenv/downloads/
cd ~/pythonenv/downloads/
svn checkout http://networkx.lanl.gov/svn/pygraphviz/trunk pygraphviz
~/pythonenv/bin/easy_install waitress
~/pythonenv/bin/easy_install WebError
~/pythonenv/bin/easy_install pyramid-handlers
~/pythonenv/bin/easy_install pyramid-beaker
~/pythonenv/bin/easy_install pyramid_debugtoolbar
~/pythonenv/bin/easy_install psycopg2
~/pythonenv/bin/easy_install pycrypto
~/pythonenv/bin/easy_install SQLAlchemy
~/pythonenv/bin/easy_install lxml
~/pythonenv/bin/easy_install paramiko
```

- Edit the setup.py file ~/pythonenv/downloads/pygraphviz/setup.py

and add/replace the following lines:

```
library_path='/usr/lib/graphviz/'
include_path='/usr/include/graphviz/'
```

Then run: .. code-block:: bash

> ~/pythonenv/bin/python setup.py install

3.1.2 Installing Ockle's GUI

Ockle's web-based GUI uses Pyramid, a python-based web development framework. You can either deploy the pyramid app on a apache/nginx webserver, or you can run it on a standalone webserver. To run it on a standalone webserer you can run the supplied script:

```
~/pythonenv/bin/python src/webserer/application.py
```

Note: Currently if the GUI can't communicate with Ockle an error message is displayed. If this happens to you follow your server's error log to see why the communication has failed.

Note: The standalone webserver loads by default on port 8000 .

3.1.3 How to set up

- Copy config.ini.example to config.ini

Once the file is copied Ockle should be able to run. You can tweak the config.ini file manually or use the webserver GUI which should.

3.1.4 How to run

To run the Ockle simply exacute:

```
~/pythonenv/bin/python src/MainDaemon.py
```

3.2 Using Ockle

3.2.1 Running Ockle for the first time

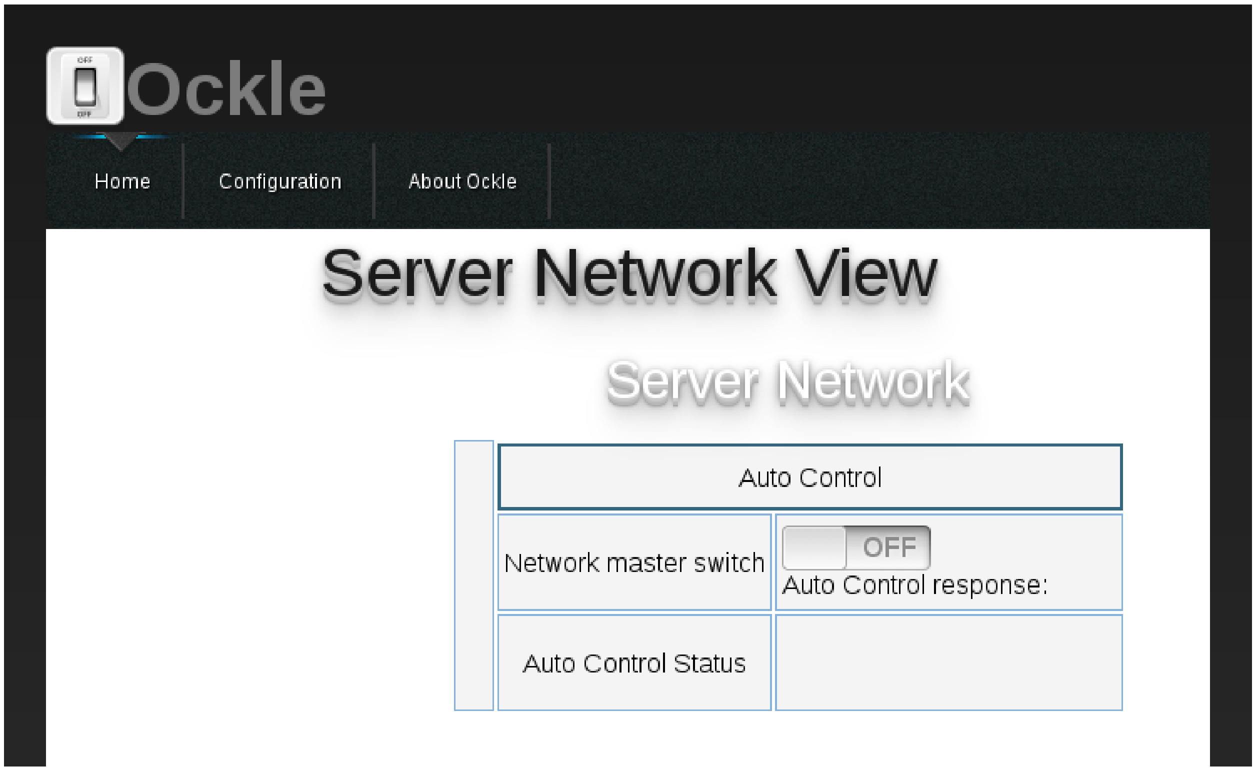
To run the Ockle simply exacute:

```
~/pythonenv/bin/python src/MainDaemon.py
```

Then you can run the GUI:

```
~/pythonenv/bin/python src/webserer/application.py
```

Once you have done that you can enter the webserver via port 8000 . You should see the following page:



This it not much, since there are no servers configured yet. You will need to enter the 'configuration' section at the top of the page and add servers to the server network.

3.3 Plugins List

One of Ockle's main features is that its completely plugin-driven. So functionality can be switch on or off by enabling and disabling plugins.

Disabling plugins can be done in `etc/config.ini` in the `[plugins]` section under `plugins`, or via the GUI in the general section of the configuration tab.

3.3.1 Plugins

- **AutoControl** - When enabled gives automatic commands requiring switching the whole network.
- **CoreCommunicationCommands** - This plugin gives basic communication commands such as listing the existing servers, their states etc.
- **EditingCommunicationCommands** - When enabled modifying INI files is possible via remote clients.
- **Logger** - This plugin logs periodically data from the outlets and controllers in all servers.
- **SocketListener** - This plugin enables sending commands to Ockle via server sockets.

CHAPTER

FOUR

DEVELOPER MANUAL

4.1 Ockle's Core functions

These are objects that are in the core of Ockle

4.1.1 MainDaemon.py

class `MainDaemon.`**`MainDaemon`**

The Main Daemon runs the Ockle Core, and controls the Server network, It loads the plugins which decide what the network behavior should be

`getAvailablePluginsListIndex``(`*dict={}*`)`

Get an Index of available plugins @return: a dict with available plugins with their name as the index, and the description as their value

`getPluginList``()`

Get a list of all class plugins @return: a list of all class plugins

`reload``(`*dataDict*`)`

A general function to reload everything

`shutdown``()`

Shutdown Ockle

4.1.2 Ockle's Network Tree Data Structure

The Whole Network Tree

Ockle's main data structure is a acyclic graph implemented by pygraph, that lives in an instance of network-Tree/ServerNetwork.py . This graph holds ServerNodes instances, each one represents a server.

You can build a server network from Ockle's ini files using the *Sever Network Generator*

class `networkTree.ServerNetwork.`**`ServerNetwork`**

The class that handles the graph server network

`addDependency``(`*server*, *dependency*`)`

Add a dependency to a server

Parameters

- **server** – the name of the server

- **dependency** – the name of the server the former is dependent on

Raises DependencyException Will raise an exception if there was a cycle in the server network

`addServer`(*node*, *dependencies*=[ ])
Add a server to the network

Parameters

- **node** – a server in the network
- **dependencies** – list of the server names this sever is dependent on

`allOff`()
Turn all servers off ungracefully

`getDependencies`(*server*)
Get a list of servers a given server is dependent on (only one level)

Parameters **server** – the server name

`getDependent`(*server*)
Get a list of servers that are dependent on this server

Parameters **server** – the server name

`getRoot`()
Gets the root server of the tree

Returns the root server

`getServer`(*serverNameSearch*)
Get a server by name

Parameters **serverNameSearch** – The server to search for

Returns The server class, None if not found

`getServernode`(*serverName*)
Get a server node by name

Parameters **serverName** – The name of the server node

Returns The server node

`getSortedNodeList`()
returns a list of the nodes topologically sorted

Returns a list of the nodes topologically sorted

`getSortedNodeListIndex`()
returns a list of the node names topologically sorted

Returns a list of the node names topologically sorted

`isAllOpState`(*opState*)
Check if all servers are ok

Returns True if all servers are on

`isReadyToTurnOn`(*server*)
Is a server ready to be turned on?

Returns True if the server is ready to be turned on

`removeDependency`(*server*, *dependency*)
Remove a dependency from a server

Parameters

- **server** – the name of the server
- **dependency** – the name of the server the former is dependent on

`turnOffServer`(*serverName*)
Turn a server off by name

Parameters **serverName** – The server name

`turnOnServer`(*serverName*)
Turn a server on by name

Parameters **serverName** – The server name

`turningOn`()

Returns true if we have any servers that are in intermediate states

`updateNetwork`()
Updates the opstate of all the nodes and their outlets/tests and controllers

A Server Node Within the Network

The Server Node object holds the global operation state of the server, and methods to control the server as a whole. Server objects are also stored in this instance. Currently server objects are: Outlets, Controls and Tests.

class `networkTree.ServerNode.`**`ServerNode`**(*name*, *outlets=[]*, *tests=[]*, *controls=[]*)
This class represents a PC in the network

`action`(*actionString*, *ignoreDeps=False*)
Execute an on/off action on the server

Parameters

- **actionString** – Either “on” or “off”
- **ignoreDeps** – True if you want to ignore other server dependencies

`controlsStillStarting`()
Return true if any control is still on SwitcingOn OpState

`getControlsDataDict`()
Get the data dict of all the controls

Returns the controls data dict

`getFailedTests`()
return a list of failed tests

`getNotControlsOpState`(*opState*)
Returns controls that don’t have a given opState

Parameters **opState** –

Returns controls that don’t have a given state

`getNotOutletsOpState`(*opState*)
Returns outlets that don’t have a given opState

Parameters **opState** –

Returns outlets that don’t have a given state

getOutlet(*number*)
Get an outlet from the outlet list

:param number outlet number in the list :return: an outlet type that is in the given place

getOutlets()
Get a list of outlet numbers @return: a list of outlets

getOutletsDataDict()
Returns a dict that holds all the outlets and their data dict. This gets sent to the logger

Returns A dict with each outlet name, and a dict of its data

getShutdownAttempts()
Get number of shutdown attempts

Returns Number of shutdown attempts

getStartAttempts()
Get number of startup attempts

Returns Number of startup attempts

incrementShutdownAttempt()
Increment the stop attempt counter

Returns Number of shutdown attempts

incrementStartAttempt()
Increment the startup attempt counter

Returns Number of startup attempts

outletsStillStarting()
Return true if any outlet is still on SwitcingOn OpState

setControlOpState(*opState*)
Set all the controls to a given opState

Parameters **opState** – The opState to set the control to

setName(*name*)
Set the name of the Server :param name: The name ot be set

setOpState(*state*)
Set the operating state of the server

setOutletsOpState(*opState*)
Set all the outlets to a given opState

Parameters **opState** – The opState to set the outlets to

setOutletsState(*state*)
Sets the outlets all to a given state by force

Parameters **state** – set the outlets to state (boolean)

Returns A list of outlets the failed (note: you can check with "if not" to see if there was no failure

setState(*state*)
Set server state

Parameters **state** – server state type

turnOn(*ignoreDeps=False*)
Turn on the server outlets, and check if all services are in order

Parameters **ignoreDeps** – True if you want to ignore other server dependencies

updateOpState(*runTests=True*)
Update all the OpStates and run all tests of the server

Related Topics

Sever Network Generator

Using this class you can build a server network object from a collection of INI files in the etc folder.

class networkTree.ServerNetworkFactory.**ServerNetworkFactory**(*MainDaemon*, *reportDependencyexceptions=True*)
A class to take the config file folder and turn it in to a server network

Parameters **MainDaemon** – the *MainDaemon.py* singletron, only used for debug output

getControllersDictIndex()
Get the index of available controller types

Returns A list of strings of controller type names

getOutletsDictIndex()
Get the index of available PDU types

Returns A list of strings of PDU type names

getTestersDictIndex()
Get the index of available testers types

Returns A list of strings of tester type names

Operation States (OpStates)

All objects in *Ockle's Network Tree Data Structure* keep Operation States of their objects they represent. By tracking the states its easy to find out what component is faulty in the server network.

Server/Outlet/control OpStates

class common.common.**OpState**
Operation state enum, that all other operation states enums extend

common.**OpState** = **<class common.common.OpState at 0x38d2c80>**

OFF = **'OFF'**
Outlet/Control/server are off

OK = **'OK'**
Outlet/Control/server are on and running

SwitchingOff = **'Switching off'**
Outlet/Control/server is switching off

SwitcingOn = **'Switching on'**
Outlet/Control/server is switching on

`failedToStart` = **'Failed to start'**
    Outlet/Control/server failed to start

`failedToStop` = **'Failed to stop'**
    Outlet/Control/server failed to stop

`forcedOff` = **'Forced off'**
    Outlet/Control/server if forced off

`forcedOn` = **'Forced on'**
    Outlet/Control/server if forced on

`permanentlyFailedToStart` = **'Permanently failed to start'**
    Outlet/Control/server has permanently failed to start

`permanentlyFailedToStop` = **'Permanently failed to Stop'**
    Outlet/Control/server has permanently failed to stop

Test OpStates

class `testers.TemplateTester.`**`TesterOpState`**

`TemplateTester.`**`TesterOpState`** = **<class testers.TemplateTester.TesterOpState at 0x3c67328>**

`FAILED` = **'FAILED'**
    Test has failed

`SUCCEEDED` = **'SUCCEEDED'**
    Test has succeeded

4.1.3 Communication Handler

The communication handler is a class that stores all the commands Ockle can handle from an external client. There is one instance of this class on the whole program and it is used to add new commands all over Ockle (both core and plugins).

A communication plugin (such as the SocketListner plugin) is then used to handle an incoming command.

A command consists of a command name and a data dict. A reply is either the same command with a dataDict holding the reply, or a command with the name "Unknown Command" if the communication Handler does not recognize the request.

The class that builds a message to be sent over a communication plugin is the Message class, located in CommunicationMessage module. It should not really be used directly since only the communication handler and a single function in *The Communication Client*.

Communication Handler Class

class `common.CommunicationHandler.`**`CommunicationHandler`**(*mainDaemon*)
    Handle communication massages from a listener plugin

`AddCommandToList`(*command*, *function*)
    Used by plugins to add an ability to handle a message in the CommunicationHandler

Parameters

- **command** – The command to be called
- **function** – a callback to a function that receives a dict of the data to process

handleMessage (*message*)
Receives a message class type, and returns the appropriate response

Parameters **message** – The message class we received

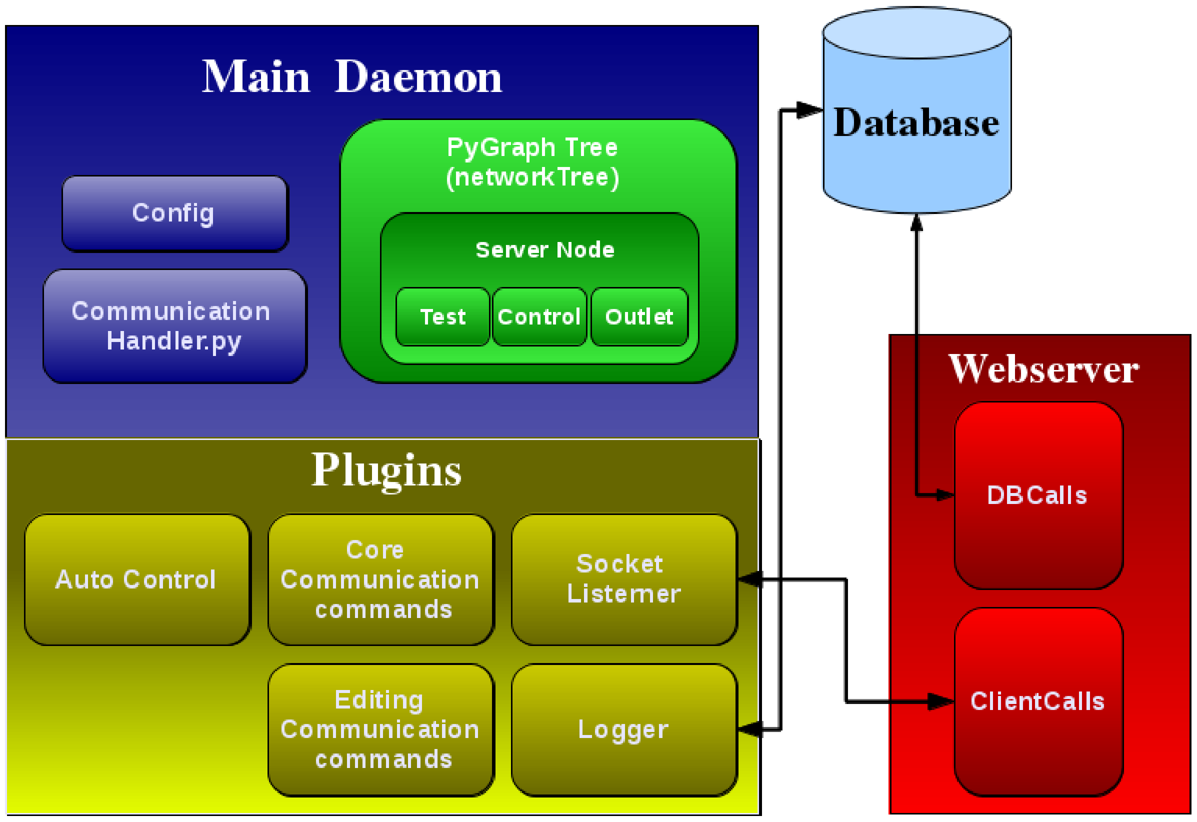
Returns A message class response

listCommands (*dataDict*)
A command to list all available commands on the communication server

Parameters **dataDict** – a dict of strings with the information passed to the handling method

Returns the response from the handling method

4.1.4 Ockle's Diagram



4.2 Plugins

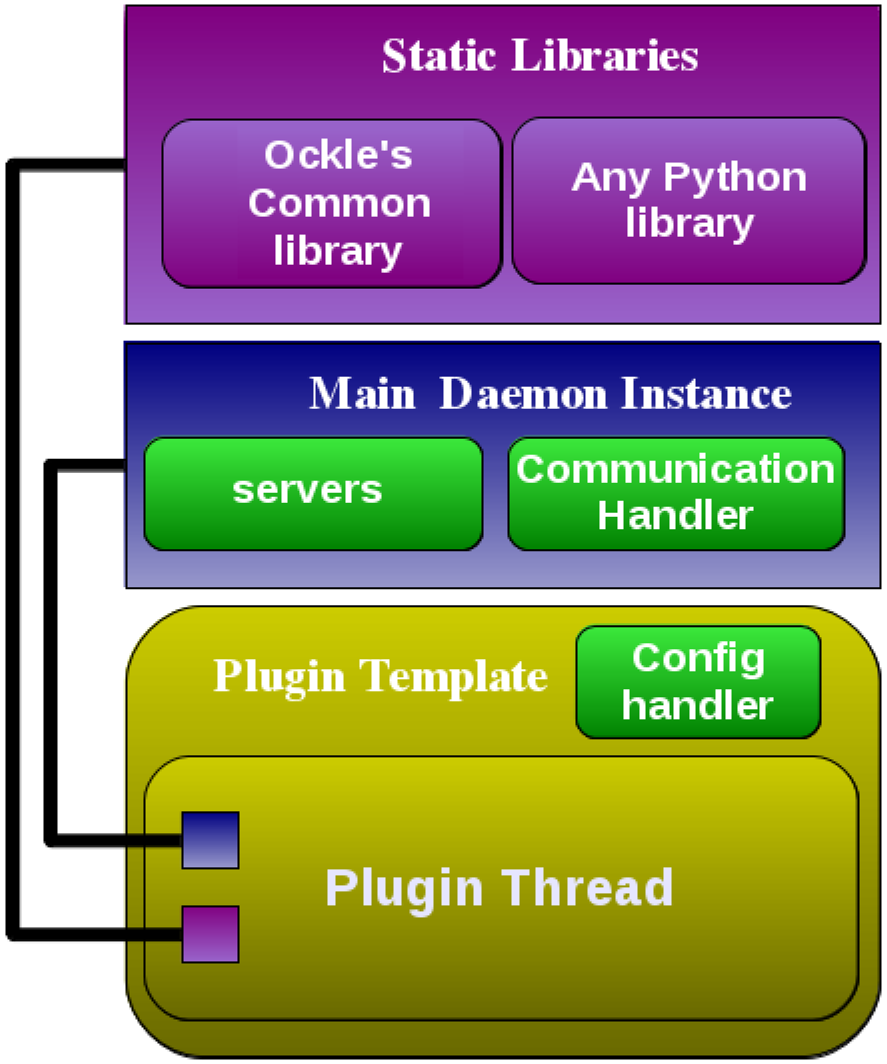
Note: One of the main concept of Ockle's design is that everything that could be a plugin, should be.

Ockle allows to add major features by the use of plugins. Each plugin is python class instance that gets executed in its own thread, allowing the developer to add new logic and behavior. You should be able to write a plugin without modifying Ockle's core. But should be able to access any method within it. Many core functions in Ockle are plugins themselves including the Automatic server control and the communication to the web-based GUI.

In order to write a plugin, you should know that there are many pre-built tools that would help you in building one. Including a way to place your configuration variables in the GUI via simple Plugin ini template files

A general description of the tools available for the plugin would look like this:

4.2.1 Plugin Framework Diagram



Every plugin is supplied with a pointer to the Main Daemon singletron, allowing access to services such as the server tree data-structure (to change the state of the servers) and the communication handler (which lets you add more commands to the communication with the webserver or any other external client). The plugin also gets access to all the functions defined in the plugin template class, such as special functions that arrange the configuration variable storage.

4.2.2 The Template plugin Class

To use this plugin framework simple extend the `plugins.ModuleTemplate.ModuleTemplate` . You may either extend the `__init__` function to do things with Ockle starts, or the `run` method that will run your code in a seprate thread with access to Ockle's functionality. You can also use the `__init__` function to register new commends to send to Ockle as done in `plugins.CoreCommunicationCommands.CoreCommunicationCommands`.

class `plugins.ModuleTemplate.`**`ModuleTemplate`**`(`*MainDaemon*`)`
The basic plugin that that all other plugins must extend

`debug``(`*message*`)`
Debug message for a module

Parameters **message** – debug message

`getConfigInt``(`*value*`)`
Get a value from the config ini for a plugin

Parameters **value** – The value you want to load

Returns the value from config.ini

`getConfigVar``(`*value*`)`
Get a value from the config ini for a plugin

:param value - the value you want :return: the value from config.ini

`run``()`
To be implamented by the plugin, The main thread of the damon, this function runs in its own thread

`stop``()`
Called to request the thread to terminate

Example

Here is a simple plugin example, this plugin simply sends to debug "I am a test plugin" message every X seconds, as defined in its config var.

4.2.3 Plugin ini template files

If you want the configuration variable to be changeable at the webserver GUI, you must provide a template ini file in the `src/config/plugins` folder. The files should have the name of the plugin class proceeded with the .ini ending.

The section should be named `plugins.<plugin name>` .

These template files follow Ockle's *INI Template file format* .

Example

Lets look at our `TimerPluginExample` example from before. We will need it to have a `src/config/plugins/TimerPluginExample.ini`. This file should contain the following text:

With those two files in place Ockle takes it from here and the plugin would be available to the user in the config sections.

4.3 Webserver - Ockle's GUI

Ockle's GUI is a pyramid 1.2 application that communicates to the Ockle Daemon.

There are a few helper functions for the view's page

4.3.1 Helper fuctions for generating multi-choice config pages

When creating config pages with multi choice fields, you *must* populate the `multiListChoices` variable and pass it to the template, this can be done using the `views.multiChoiceGenerators` module: Ockle PDU and servers manager Helper functions for creating multi-choice fields that can then be displayed by the GUI

Created on Oct 27, 2012

@author: Guy Sheffer <guy.sheffer at mail.huji.ac.il>

`views.multiChoiceGenerators.`**`_makeMultichoice`**`(`*section, option, multiListChoicesCallback, INIFileDict, multiListChoices=None*`)`

Generate a multilist format for a template. So it can be rendered on a template

Parameters

- **section** – The option section in the ini file
- **option** – The name of the option in the ini file
- **multiListChoicesCallback** – a callback function the returns a dict of the available options
- **INIFileDict** – An INI file dict that holds the list of selected choices
- **multiListChoices** – If there is a multiListChoices dict you want to append the existing configuration to

Returns a multiListChoices dict ready to be rendred in a template

`views.multiChoiceGenerators.`**`_makeSelectMulitChoice`**`(`*existingType, objectType, item, getObjectDict, multiListChoices=None*`)`

Make a multi select option for the select type

Parameters

- **existingType** – The selected option
- **objectType** – The section to build
- **item** – The item to build
- **getObjectCallback** – the Dict holding the select list
- **multiListChoices** – an existing multiListChoices dict (optional)

Returns The updated multiListChoices dict

4.4 The Communication Client

The communication is a python library that lets you send commands to Ockle from a python shell using an external program. Ockle PDU and servers manager Client calls to the Ockle server

Created on Apr 25, 2012

@author: Guy Sheffer <guy.sheffer at mail.huji.ac.il>

`ockle_client.ClientCalls.`**`deleteINIFile`**`(`*iniPath*`)`

Delete an INI file from Ockle's configuration

Parameters **iniPath** – the path of the ini file starting from the 'etc' folder

Returns A response from Ockle

`ockle_client.ClientCalls.`**`deleteINISection`**`(`*section*, *iniPath*`)`
Delete a section from an INI file in Ockle's configuration

Parameters

- **iniPath** – the path of the ini file starting from the 'etc' folder
- **section** – the section to be deleted

Returns A response from Ockle

`ockle_client.ClientCalls.`**`getAutoControlStatus`**`()`
Get the status of the Auto Control plugin

Returns A dict with a key 'status' holding the status of Auto Control

`ockle_client.ClientCalls.`**`getAvailableControllersList`**`()`
Get the currently available controller types list

Returns a sorted dict of controllers, the key is the name of the controller, and extra information is within the dict's value

`ockle_client.ClientCalls.`**`getAvailablePDUsList`**`()`
Get the currently available PDU types list

Returns a sorted dict of PDUs, the key is the name of the PDU, and extra information is within the dict's value

`ockle_client.ClientCalls.`**`getAvailablePluginsList`**`()`
Get the list of available plugins

Returns a dict with the plugin names as keys and the description as the value

`ockle_client.ClientCalls.`**`getAvailableServerControls`**`(`*server*`)`
Get the currently configured controls of a given server

Returns a sorted dict of controls, the key is the name of the test, and extra information is within the dict's value

`ockle_client.ClientCalls.`**`getAvailableServerOutlets`**`(`*server*`)`
Get the currently configured servers list

Returns a sorted dict of servers, the key is the name of the server, and extra information is within the dict's value

`ockle_client.ClientCalls.`**`getAvailableServerTesters`**`(`*server*`)`
Get the currently configured tests of a given server

Returns a sorted dict of tests, the key is the name of the test, and extra information is within the dict's value

`ockle_client.ClientCalls.`**`getAvailableTestersList`**`()`
Get the currently available testers type list

Returns a sorted dict of testers, the key is the name of the tester, and extra information is within the dict's value

`ockle_client.ClientCalls.`**`getControllerDict`**`()`
Get a dict of the current controllers that are configured

Returns A dict of controllers with the key as their name and the value as their description

`ockle_client.ClientCalls.`**`getControllerFolder`**`()`
Get the configuration folder of all controllers

Returns A string of the folder name

`ockle_client.ClientCalls.`**`getDataFromServer`**`(`*command*, *paramsDict={}*, *noReturn=False*`)`
Send a command to the Ockle server, and return the responce dict

Parameters

- **command** – The command to send
- **paramsDict** – the dictionary that is sent with command arguments
- **noReturn** – Should we not expect a reply. Used in cases were we want to restart the Ockle server

Returns A dict with the response data, None if we failed to connect

`ockle_client.ClientCalls.`**`getINIFile`**`(`*iniPath*`)`
Get an INI file from Ockle's configuration

Parameters **iniPath** – the path of the ini file starting from the 'etc' folder

Returns A string with the ini file contents

`ockle_client.ClientCalls.`**`getPDUDict`**`()`
Get a dict of the current PDUs that are configured

Returns A dict of PDUs with the key as their name and the value as their description

`ockle_client.ClientCalls.`**`getPDUFolder`**`()`
Get the configuration folder of all PDUs

Returns A string of the folder name

`ockle_client.ClientCalls.`**`getServerAvilableDependencies`**`(`*server*`)`
Get a dict of the available dependencies that can be created for a server

Parameters **server** – the server that is going to have the new dependency

Returns A dict of servers and their description

`ockle_client.ClientCalls.`**`getServerDict`**`()`
Get a dict of the current servers that are configured

Returns A dict of servers with the key as their name and the value as their description

`ockle_client.ClientCalls.`**`getServerFolder`**`()`
Get the configuration folder of all servers

Returns A string of the folder name

`ockle_client.ClientCalls.`**`getServerTree`**`()`
Get a server tree status from the Ockle server, and return a dict ready to be parsed by a pyramid view

Returns a string with the dot graph

`ockle_client.ClientCalls.`**`getServerView`**`(`*serverName*`)`
Get information of the server

Parameters **serverName** – the server's name

Returns a dict of string of the server's info

`ockle_client.ClientCalls.`**`getTesterDict`**`()`
Get a dict of the current testers that are configured

Returns A dict of testers with the key as their name and the value as their description

`ockle_client.ClientCalls.`**`getTesterFolder`**`()`
Get the configuration folder of all testers

Returns A string of the folder name

`ockle_client.ClientCalls.`**`listCommands`**`()`
A command to list all available commands on the communication server

Returns A dict with the command names as the key and a description if available as their value

`ockle_client.ClientCalls.`**`loadINIFileConfig`**`(`*configPath*`)`
Get the config on an ini file @param configPath: the path to the config relative to etc @return: a dict of the config

`ockle_client.ClientCalls.`**`loadINIFileTemplate`**`(`*templatePaths*`)`
Load an INI file and template data so it would display correctly. Is called with loadINIFileConfig(configPath)

Parameters **templatesPaths** – A path, or list of paths relative to 'src/config'

Returns A dicts of the template

`ockle_client.ClientCalls.`**`restartOckle`**`()`
Restart Ockle

`ockle_client.ClientCalls.`**`runTest`**`(`*dataDict*`)`
Switch a server outlet on or off

Parameters **dataDict** – a dict holding two keys: 'server' key for the server's name and an 'obj' key for the outlet's name

Returns the OpState of the test

`ockle_client.ClientCalls.`**`serversDependent`**`(`*server*`)`
Get a dict of servers that this server is dependent on

Parameters **server** – The server to check for

Returns a dict of servers that this server is dependent on

`ockle_client.ClientCalls.`**`setAutoControlStatus`**`(`*dataDict*`)`
Set the status of Auto Control

Param dataDict: A dictionary with the field status which is either 'on' or 'off'

Returns A dict similar to ::func: getAutoControlStatus

`ockle_client.ClientCalls.`**`setINIFile`**`(`*iniPath*, *iniDict*`)`
Set an INI file from Ockle's configuration

Parameters

- **iniPath** – the path of the ini file starting from the 'etc' folder
- **iniDict** – a dict holding the structure of the ini file

Returns A response from Ockle

`ockle_client.ClientCalls.`**`setServer`**`(`*dataDict*`)`
Set a server on or off

Parameters **dataDict** – A dict with two keys, one with the key 'server' which holds the server name in its value, and another with the key 'state' where its value is wither 'on' or 'off'

Returns A dict with the key 'status' containing a string reply from Ockle

`ockle_client.ClientCalls.`**`switchControl`**`(`*dataDict*`)`
Switch a server control on or off

Parameters **dataDict** – a dict holding three keys: 'server' key for the server's name, an 'obj' key for the control's name and the 'state' key with a string 'on' or 'off'

Returns the OpState of the control

`ockle_client.ClientCalls.`**`switchNetwork`**(*dataDict*)
A master command to turn all the servers on the network on or off

Parameters **dataDict** – a dict with the key 'state' that has a string 'true' or 'false'

Returns a dict with the key 'status' with a string reply from Ockle

`ockle_client.ClientCalls.`**`switchOutlet`**(*dataDict*)
Switch a server outlet on or off

Parameters **dataDict** – a dict holding three keys: 'server' key for the server's name, an 'obj' key for the outlet's name and the 'state' key with a string 'on' or 'off'

Returns the OpState of the outlet

4.4.1 Example usage

Here is a simple example on how to use the ockle_client module:

```
import webserver.ockle_client.ClientCalls as ockleClient
ockleClient.PORT = 8088
ockleClient.OCKLE_SERVER_HOSTNAME = 'localhost'

print ockleClient.listCommands()
```

4.5 Server Objects and Object Generators

In Ockle, a server holds a collection of *Server Objects* which the Ockle's *Plugins* interact with. A server object instance is created from an *Object Generator* class. Currently there are three Object Generator are: PDUs, Controllers and Testers. Those generate the Outlet, control and test objects respectively.

4.5.1 Power distribution units (PDUs) - Outlets

PDUs are object generators that create outlets for a server. Outlets represent a physical power socket that that can switch the server's power on or off. Outlet also have a `data` field that gets logged in the `PluginLogger`.

Coding a new PDU type

When creating a new one you should extend the class `outlets.OutletTemplate.OutletTemplate`.

The python file containing the class should be placed in the `src/outlets` package.

Here are the methods you should implement when writing a new PDU class:

class `outlets.OutletTemplate.`**`OutletTemplate`**(*name*, *outletConfigDict={}*, *outletParams={}*)
Template for an outlet object that all other outlets extend

Variables **data** – Holds a dict of the data from the outlet

`_setOutletState`(*state*)
To be implemented by the child, sets the outlet's state

Parameters **state** (*bool*) – The state to set

_getOutletState()
To be implemented by the child, sets the outlet's state

Returns The current outlet state

updateData()
To be Implemented in the child, updates the `self.data` variable

Example Dummy Outlet

Here is an example dummy outlet implementation

Example for an outlet INI Template File

Here is an INI template file from the Raritan PDU, located at `src/config/conf_outlets/Raritan.ini`:

4.5.2 Controllers - Controls

Controllers are object generators that create controls for a server. Controls are a set of commands that can tell a server to switch itself off on the software level (before the outlets switch off its power). Controllers also have a `data` field that gets logged in the `PluginLogger`, enabling logging information from the servers.

Coding a New Controller Type

When creating a new controller type you should extend the class `controller.ControllerTemplate.ControllerTemplate`

The python file containing the class should be placed in the `src/controllers` package.

Here are the methods you should implement when writing a new Controller class:

class `controllers.ControllerTemplate.`**ControllerTemplate**(*name*, *controllerConfigDict={}*, *controllerParams={}*)
Template for a control object that all other controls extend

Variables **data** – Holds a dict of the data from the control

_setControlState(*state*)
To be implemented by the child, sets the control's state

Parameters **state** (*bool*) – The state to set

_getControlState()
To be implemented by the child, sets the control's state

Returns The current control state

updateData()
To be Implemented in the child, updates the `self.data` variable

Example Dummy Control

Here is an example dummy outlet implementation

Example for a control INI Template File

Here is an INI template file from a control to send ssh commands to a server, located at `src/config/conf_controllers/SSHController.ini`:

4.5.3 Testers - Tests

Testers are object generators that create tests for a server. Tests are a set of commands that runs after a server has been switched on, to make sure its serving the network correctly.

Coding a New Tester Type

When creating a new tester type you should extend the class `testers.TemplateTester.TemplateTester`.

The python file containing the class should be placed in the `src/testers` package.

Here is the methods you should implement when writing a new Tester class:

class `testers.TemplateTester.`**`TemplateTester`**(*name*, *testerConfigDict*, *testerParams*)

> **`_test`**()
>
> To be implemented by the child, runs the test
>
> **Returns** Return True if succeeded

Example Dummy Tester

Here is an example dummy outlet implementation

Example for a control INI Template File

Here is an INI template file from the dummy test above, which is placed in `src/config/conf_testers/SSHController.ini`:

4.5.4 Object Generators common tools

INI Template files

You can specify global parameters for the PDU, controllers and testers and specific parameters for each server outlet, control and test.

Object Generator parameters go in a section named after that object generator. For example, PDUs have a `[pdu]` section.

Server Object parameters on in a section named after the Server Object, followed by the word Params. For example an outlet will will have an `[outletParams]` section.

These template files follow Ockle's *INI Template file format* .

4.6 INI Template file format

Note: INI Template file format is only accessible by developers, it should not be changed by users.

Ockle has various configuration directives that are set in a common INI Template file format. By using these templates Ockle module developers simply specify what configuration variables their module has, and Ockle's core would let the user edit them comfortably in the gui.

These files define how the INI configuration files should be written.

4.6.1 Available settings data types

INI Template files include the variable name as items, and a json formatted list with the type followed by a default variables.

Current types supported:

Type	Field	Example
string	default	["string","yay"]
int	default	["int",1]
bool	default	["bool",true]
intrange	default, range	["intrange",1,"1-8"]
select *	select disabled?	["select",false]
multilist *	ordered? , sorted?, Url Pattern **	["multilist",true,"~~name~~"]

* These require the mulichoice variable to be defined

** ~~name~~ string would be replaced by the multichoice's value

4.7 Libraries used (learned?)

- pyGraph – python graph data structure
- PyDot library / xDot format
- SQLAlchemy – cross-platform databas
- Pyramid – Webserver framework
- Chameleon template engine
- Graphviz / Canviz – Graph visualization libraries
- JqPlot - a plotting and charting plugin for the jQuery Javascript framework
- PySNMP – Communication with the Raritan Dominion PX Remote Power Control
- straight.plugin – A plugin loading facility
- Socket (python standard library class) - Low-level networking interface
- prototype.js - The main page requires prototype for Canvoiz to work
- sphinx - Documentation

CHAPTER

FIVE

PROJECT

5.1 Future Work

What could be added:

1. Add option to rename server on the webserver
2. Make group webserver functions in to an object-oriented structure.
3. Support to turn on and off a specific server and all its dependents
4. Change Message class to work with json and not xml (so the javascript calls won't hold a mixture of json and xml)
5. Add more generic controllers and testers
6. Add a virtual machine outlets (So an outlet could turn a virtual machine on, not a physical one)
7. More AJAX live updates of the network in the GUI
8. Catastrophe handling - make Ockle start up when major config variables are not set.
9. More Socket communicators apart from the socket handler
10. Make the control/outlet/test scheme more universal
11. Support more types of databases in the logger
12. Get canviz to work with jquery and drop the need for prototype.js
13. Better installer, have a nice bootstrap with main setup options
14. Add more standard methods to pull config variables in the server objects (instead of doing things like `self.state = json.loads(testerParams["succeed"])`).

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX