ioService Framework

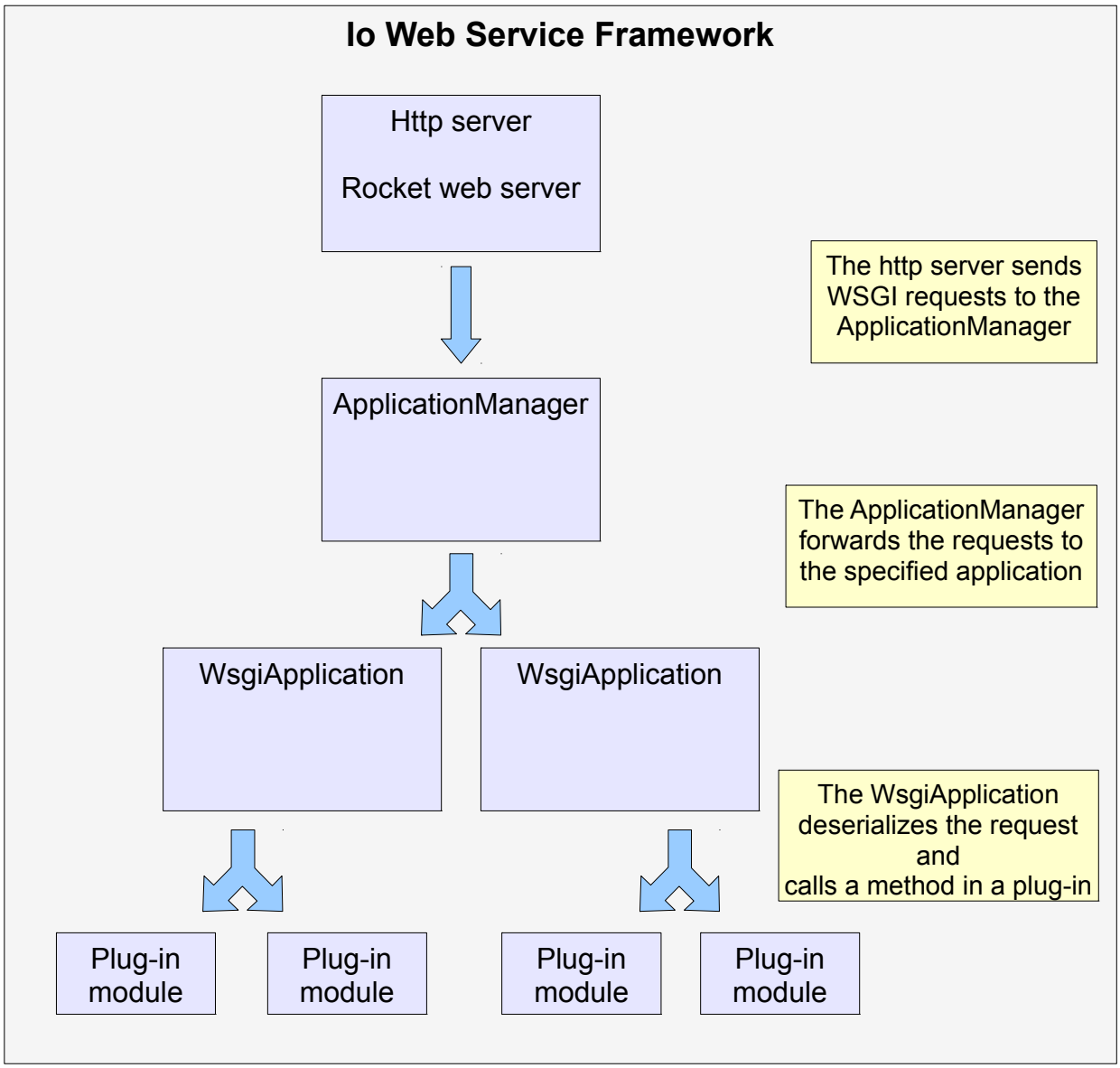
Purpose

This document gives an overview of the Web Service Framework.

Table of contents

1 Introduction

The Ioservice Web Service Framework is a small lightweight framework to develop web service applications. The framework is entirely developed in python. It uses JSON-RPC 2.0 to receive requests and send responses.

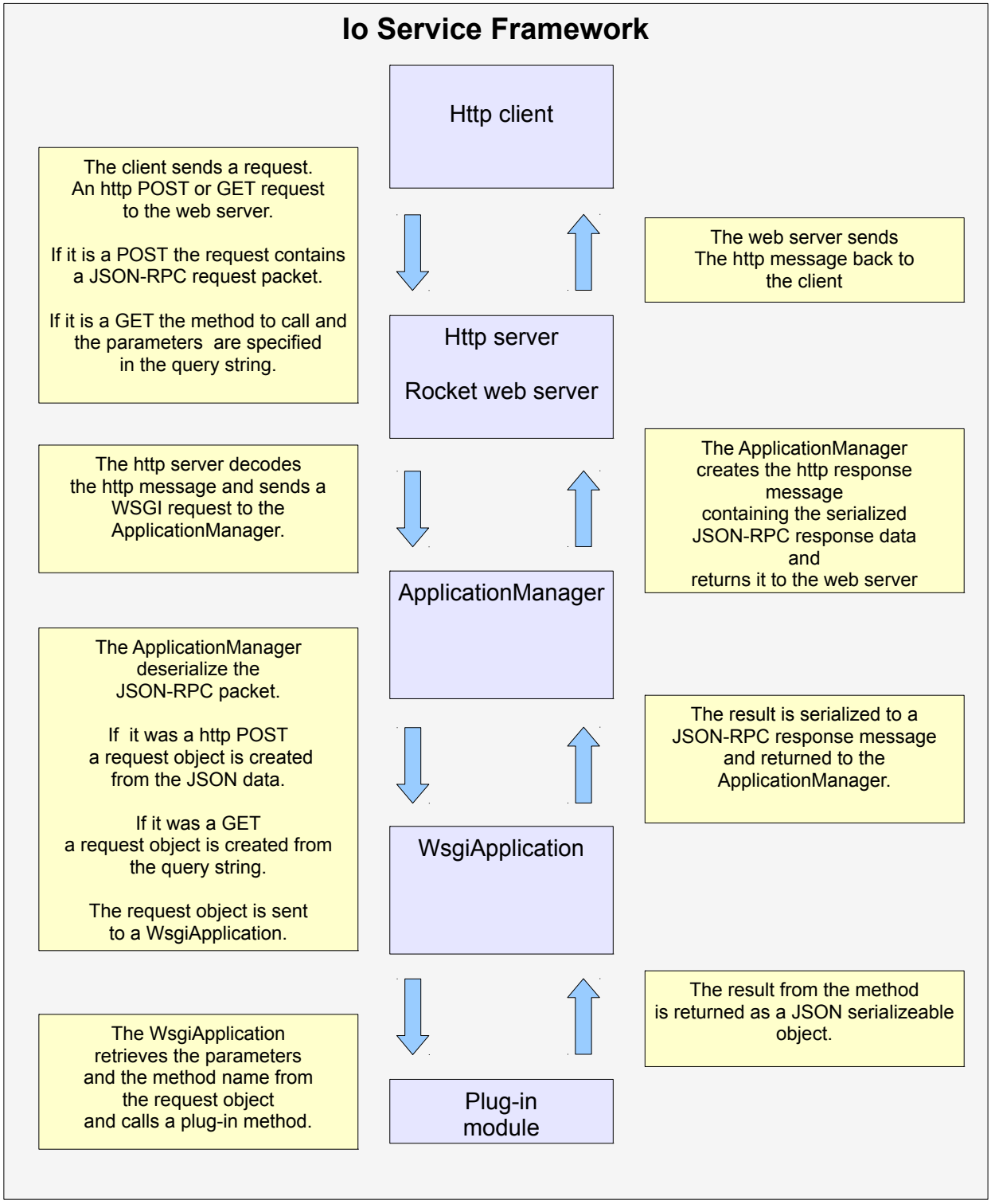


1.1 Design goals

- A small lightweight framework
- No large third party components to install
- Easy to add new functionality
- Dynamic loading and unload of plug-in modules
- Reuse as much as possible of the python library
- It should be possible to run it on Linux and Windows

2 Command flow

The picture below shows how a http request is handled by the framework.



3 Web service methods

The web service methods can be called using a http POST containing a JSON-RPC 2.0 request message. In this case the Content-type of the http request must be set to *application/json-rpc*.

The methods can also be called with a http GET request in this case the parameters should be included in the query string of the URL.

Example:

http://deserver:8080/DeviceMonitor.Devices.Release?ticket=3244342343333&id=43434

The framework will internally convert the GET request to a JSON message and call the method. The parameter id is the JSON-RPC id, if not included it is set to null.

The response of a method call is always returned as a JSON-RPC response message or a JSON-RPC error message if something went wrong decoding the request.

3.1 Communication Protocol

All requests and responses are encoded as JSON-RPC 2.0.

3.2 Message Structure Request

A request message contains the following fields: jsonrpc, method, id and params.

Field details:

jsonrpc	The version of json protocol.
method	The method to call.
params	Parameter to method, method specific see the api documentation
id	The message id.

JSON example:

```
{
    "jsonrpc" : "2.0",
    "method" : "System.Core.Ping",
    "id" : "1320855768109",
    "params": {
        "authentication": "1234",
        "model": "Z",
        "software" : "vodafone-v1.2"
    }
}
```

3.3 Message Structure Response

A response message contains the following fields: jsonrpc, result, id.

Field details:

jsonrpc	The version of json protocol.
result	The return value from the method,
id	The message id.

JSON Example:

```
{
    "jsonrpc" : "2.0",
    "result": {
        "status": "NO",
        "message": "The resource cannot be locked",
        "code" : "1029"
    },
    "id" : "1320855768109"
}
```

3.4 Message structure JSON error response

This message is an error message that is not returned by a method. The message contains the following fields: jsonrpc, error and id.

Field details:

jsonrpc	The version of json protocol.
error	The error field that contains code, message and data. Error codes for field code see 3.6.
id	The message id.

JSON example:

```
{
    "jsonrpc" : "2.0",
    "error": {
        "code": "-32601",
        "message": "Method not found",
        "data" : "Plugin error: -1025 - Plugin method not found: Session.noMethod"
    },
    "id" : "1320855768109"
}
```

3.5 Authorization

To access the service methods and html pages the http digest access authentication must be used. Digest authentication is supported by all major Web browsers.

Digest access authentication is specified by RFC 2617.

The ioservice currently implements the most basic authentication scheme.

HA1 = md5(username:realm:password)

HA2 = md5(method:uri)

response = md5(HA1:nonce:HA2)

Systems that needs to store the username and password could store the username and the HA1 hash instead of storing the password in plain text.

3.6 Error codes

There are three types of error codes:

- HTML
- JSON
- Method specific

The HTML error codes are the standard codes e.g. 400, 401 etc. When a JSON error is returned this is due to an unhanded error in the method. Method specific errors are handled by the methods, see the api documentation.

JSON error codes:

If the JSON request contains an invalid request an error response is returned.

Code	Name	Message
-32700	PARSE_ERROR	Parse error. Invalid JSON was received
-32600	INVALID_REQUEST	Invalid Request. The JSON sent is not a valid Request object
-32601	METHOD_NOT_FOUND	Method not found. The method does not exist.
-32602	INVALID_METHOD_PARAMS	Invalid parameters.
-32603	INTERNAL_ERROR	Internal JSON-RPC error
-32000	PROCEDURE_EXCEPTION	An exception occurred in the calling procedure

4 Applications and plug-in modules

The Applications and plugin modules are organized in a namespace structure that is used to access a web service method.

To access a plugin method, the application, plugin and method name must be specified.

<application>.<plugin>.<method>

Example:

To call the method *Ping* in the plugin *System* of the application Core use the name: *Core.System.Ping*

4.1 Directory structure

The applications and plugins module are located in the *applications* directory.

The applications directory contains one directory for each application.
An application directory contains a configuration file for the application a*pplication.cfg* the *__init__.py* file and a subdirectory named *plugins* where the application plugins must be located.

The namspace used by the web-service access is defined in the application and plugin configuration files and not related to the directory names.

When ioservice start it will search the applications directory and all subdirectories for application and plugin modules and load them if they are enabled.

4.2 Application configuration

The application configuration file *application.cfg* contains information how to load the application module.

```
[APPLICATION]
Enabled = True
Namespace = Ftp
Description = FTP server
ModuleName = genericApplicationHandler
ClassName = GenericApplicationHandler
```

To prevent the application from loading, set the *Enabled* property to False.

The value of the *Namespace* property is the application name to use when a method in the application is called.

The *Description* should be a short sentence describing the purpose of the application.

The *ModuleName* and *ClassName* are the name of the python module and class to load. For a normal application with no special handling at the application level they should be set to the module name *genericApplicationHandler* and the class *GenericApplicationHandler*.

If a special application handler needs to be developed, the module should be placed in the same directory as the main program *ioservice.py* and the handler class must inherit from the *WsgiApplication* class in the *wsgiApplication* module.

4.3 Plugin modules

Application plugins should be placed in the plugins subdirectory of the application each plugin in its own directory.

A plug-in consist of two files:

- <module name>.py

- plugin.def

The *.py* file contains the python implementation of the plug-in and the *plugin.def* file is used by the framework to find a plugin module, it also contains the default configuration used by the plugin.

The plugin python file and the default configuration file must be located in the same directory, except for that the directory structure can be freely organized.

The values in the table below must at least be defined in the *plugin.def* file:

```
[PLUGIN]
Enabled = True
Namespace = System
Description = System plugin
ModuleName = System
ClassName = System
```

Normally the *Namespace* name use by the framework is the same as the name of the python module, but it can be changed to another name.

The option *Enabled* is used to prevent the plug-in from loading. The default value of the *Enabled* option is True, to load the plug-in.

The *ModuleName* and the *ClassName* is the name of the python file and class in the file that should be loaded.

At runtime a copy of the *plugin.def* file is used, *plugin.cfg*. The *plugin.def* contains the default configuration. When the framework starts it will check if the *plugin.cfg* file exists, if it does not exist it will be created and the content of the *plugin.def* file is copied into it. If the *plugin.cfg* file exist the content of the two configuration files will be compared and values that do not exist in the *plugin.cfg* will be copied from the *plugin.def* file.

The configuration file can also be used to store other option values needed by the plug-in. The configuration data in the *.plugin* files uses the format specified by the ConfigParser in the python library.

The ConfigParser class implements a basic configuration file parser language which provides a structure similar to what you would find on Microsoft Windows INI files.

4.4 Plug-in methods

For a method in a plug-in to be callable from the web service it must have one of the two decorators:

@PluginMethod.

@InternalPluginMethod

Example:

```
@PluginMethod
def Ping(self, parameters = None):
    """
    Send back date and time
```

```
    :returns: Date and time in ISO 8601 format
    """
    return {'isodate':datetime.now().isoformat()}
```

The decorator @PluginMethod marks the method as a public method define in the API documentation and must follow the protocol defined.

The *@InternalPluginMethod* decorator should be used to mark methods as private for the framework and should not be called by external clients, the method marked as internal can be changed between releases and does not need to be specified in the public API.

The plug-in methods has one parameter containing the deserialized content of the *params* variable in a JSON-RPC request.

The *parameters* parameter can be of the types: tuple, list or dict.

The return value from a plug-in method must be JSON serializable, the types supported are: tuple, list or dict.

If None is returned from the method the ApplicationManager will consider the value as an error and send a JSON error message back to the client. A method that has no return value should return an empty list or dictionary.

If an exception occurs in a plug-in method that is not caught, the ApplicationManager will catch it and send back a JSON error message with error code -32000 to the client containing a stack dump.

4.5 Plugin module

The *ModuleName* and the *ClassName* is must be set to the name of the python module file and class. The class must inherit from the class *PluginBase* defined in the file *PluginBase.py*.

There are two methods that must be overridden from the base class *Initialize()* and *Dispose()*

If they are not overridden a not implemented an exception will be raised when the *pluginManager* initializes the plug-in.

The *Initialize* method should do initializations needed by the plug-in.
The method *_initializeMethodList()* in *PluginBase* must also be called to initialize a dictionary of methods that are callable from the web service.

The *Dispose* method is called just before the plug-in is unloaded and should contain necessary clean up to unload the module in a controlled way; such as closing files and stopping threads.

The method *ApplicationsInitialized()* can be overridden to do initializations after all applications are loaded and running.If not overridden it does nothing.

4.5.1 Example of a simple plug-in module and class

```
#!/usr/bin/env python

from pluginBase import PluginBase
from pluginBase import PluginMethod

class SomePlugin(PluginBase):
    def __init__(self, pmanager):
        self.message = None
        PluginBase.__init__(self,pmanager)

    def Initialize(self):
        self._InitializeMethodList()
    def ApplicationsInitialized(self):
        self.message = "Hello from plugin"

    def Dispose(self):
         pass

    @PluginMethod
    def Hello(self,parameters):
        return [self.message]
```

Plugin.def

```
[PLUGIN]
Enabled = True
Namespace = SomePlugin
Description = Example plugin
ModuleName = SomePlugin
ClassName = SomePlugin
```

4.6 Plugin base class

The *PluginBase* class contains code to manage plugins and helper methods to access different framework functions.

The content of the configuration file is loaded into a Configuration Manager object that is used to access configuration data for the plugin, *self.pluginSettings*.

Other configuration values are also accessible from the base class configurations for the web-service and the application, s*elf.serviceSettings* and *self.applicationSettings*. The serviceSettings will not be accessible in a remote plugin, a plugin running in another process or on another computer.

The base class also contains methods to use the messaging system and register web pages used by the plugin.

5 Message handling

To allow for plug-in modules to communicate with each other a kind of Publisher/Subscriber pattern is implemented.

The Message handling is implemented in the MessageManager module.

The *ApplicationManager* contains an instance of the *MessageManager* that implements methods to send messages. All plug-in modules have access to most of the methods in the *MessageManager*. In the PluginBase class there are methods to register publishers, subscribers and to send messages.

The message handling is asynchronous. The *MessageManager* runs the message handling in a separated thread. Messages are put into a queue by the *SendMessage* method. When the *MessageManager* discovers a message in the queue it is removed from the queue and all subscribers registered for the specified publisher will be called with the parameters supplied in the *SendMessage* call.

5.1 RegisterPublisher(name)

This method takes one parameter the name of the publisher. It returns True if the registration was successful.

5.2 AddSubscriber(name, callback)

This method takes two parameters, the name of a publisher and a method to call when a message arrives. The callback method has one parameter the *params* parameter used in the SendMessage call. The method returns True if the subscriber was added successfully.

5.3 SendMessage(name, params, callback)

The method sends a message to a publisher. The parameter *name* is the name of the publisher, *params* is the parameters to sends. The callback method is optional it can be used to get return values from the subscriber methods that are called. The *SendMessage* method returns immediately after the message has been put into *MessageManager* message queue.

If the subscriber callback methods returns values they are collected in a list and returned as a parameter in the callback method specified in the *SendMessage* method. The *SendMessage* parameter *callback* is optional and should only be used if there is need to return values from the subscriber methods.

The subscriber callbacks are run in the context of the *MessageManager* thread.

5.4 Sending emails

When the *ApplicationManager* is initialized two Publishers are registered that make it possible to send emails. The publisher with the name **email.text** can send plain text mails. The publisher **email.attachment** can send plain text message with attached files. See the *emailHandler.py* module for the parameters to use.

6 Web page application

A special application with the namespace Web exist in the framework. The Web application can be used to create web page that can be used by the plugins to display e.g. maintenance pages.

6.1 Web page handling

A simple web page template system has been added to the framework to allow for a plugin module to supply a user interface for e.g. configuration or status handling.

There are three methods in the plugin base class to handle html pages used by a plugin module.

- RegisterWebPage() Register a web page to be displayed by the web server
- RegisterWebPageCallback(): Register a web page using callback methods to supply the page content.
- UnregisterWebPage() Remove a web page from the web server.

6.1.1 Web page classes

In the module webPagebase.py there are three new classes to create web pages:

- *WebPageCallback* Uses callback methods to a plugin to create the web content.
- *WebPageFromFile* Creates a simple web page with the content loaded from file and a menu. Item.
- *WebPageFromTemplate* Create a web page from a web page template.

6.1.2 Jquery and Ajax support

The Jquery, Jquery-UI and an Ajax javascript library is loaded per default when a web page is displayed. All functionality in those libraries can be used when a web page is created.

6.1.3 Other javscript libraries

DataTables is a plug-in for the jQuery Javascript library. Which is used to create tables and adds advanced interaction controls to any HTML table.

Usage:

```
<style type="text/css" title="currentStyle">
   @import "datatables/css/demo_page.css";
   @import "datatables/css/demo_table_jui.css";
</style>
<script type="text/javascript" language="javascript" src="datatables/js/dataTables.js"></script>
```

7 How to create a web page

The easiest way of creating a web page is to use the class WebPageFromTemplate. This class creates a web page using a page description stored in the plugin configuration file.

Example:

```
def ApplicationsInitialized(self):
  # ftp settings html page
  page =  WebPageFromTemplate('ftpsettings.html', self.path, self)
  page.SetSortOrder('Y')
  self.RegisterWebPage(page)
```

- In the *ApplicationsInitialized* Method of the plugin class create a new *WebPageFromTemplate* object.

  WebPageFromTemplate(*pagename*, *webroot*, *plugin*)
  pagename = The name of the webpage
  webroot = The root directory when referring to files used by the web page.
  plugin = A reference to the plugin that owns the web page.
- Optionally set the sort order prefix for the menu item associated with the page. The sort order prefix determines the position of the menu item in the menu list. The default value of the prefix is the character 'Z' which will place the menu item last in the menu list.
- Register the page with the web server page handler.

A web page should be unregistered in the dispose method of the plugin.

```
def Dispose(self):
  self.UnregisterWebPage('ftpsettings.html')
  .
  .
```

7.1 Page description in the configuration file

To set up a page, a number of sections must be added to the plugin configuration file to define the content of the page.

In the section [PAGES] the page name must be set to a template description.

```
[PAGES]
ftpsettings.html = TEMPLATE_FTPSETTINGS
  .
  .
```

7.2 Template description

```
[TEMPLATE_FTPSETTINGS]
Template = f|content/ftpsettings.htm
$MainMenuItem = System
$MainSubMenu = FTP accounts:ftpsettings.html;
```

The template description must contain an option called Template. The value of the template option contains two parts a type and a reference to the html data to use separated by a "|" character.

<type>|<html data>

- type = s The data is a string with the html code for the page.
- type = m The data is a name of a plugin method to call to get the html code for the page.
- type = f The data is a name of a file containing the html code for the page.

Examples:

```
Template = f|content/ftpsettings.htm # Get the html code from a file

Template = s|<div> Some text </div>  # Html from a string

Template = m|Core.System.GetPageData # Call a method to get the code
```

7.2.1 Template variables

The html code can contain variable names beginning with a $ character. When the page handler assembles the web page the variable names are replaced with the content of the variables. The variables are defined in the same way as the Template option value.

Example:

```
[TEMPLATE_SYSTEM]
Template = s|<div>$TEXT</div><div>$USERTABLE</div>
TEXT = s|System menu
USERTABLE = m|System.HtmlGetUserTable
```

The variable name in the configuration file must not begin with a “$” character.

Variable names starting with a “$” character are used for the menu handling.

7.2.2 Menu variables

A page can be associated with a menu item that is placed in the main menu list or in a sub menu. There are two variables used to define a menu associated with a web page, *$MainMenuItem* and *$MainSubMenu*.

The *$MainMenuItem* should be set to the text to display in the main menu. The main menu is the menu to the left that is always displayed.

If The main menu item should contain a sub menu the variable *$MainSubMenu* should be set to a list of sub menu items. Additional menu variables can also be defined to create nested sub menus.

7.2.3 Menu examples

Only a main menu item:

```
[PAGES]
system.html = TEMPLATE_SYSTEM

[TEMPLATE_SYSTEM]
Template = s|<div>$TEXT</div>
TEXT = s|System menu
$MainMenuItem = System
```

In the above example a main menu item with the text System is created containing a link to the page *system.html*. When the menu item is clicked the page *system.html* is displayed.

A main menu item with a sub menu:

```
[PAGES]
system.html = TEMPLATE_SYSTEM

[TEMPLATE_SYSTEM]
Template = s|<div>$TEXT</div>
TEXT = s|System menu
$MainMenuItem = System
$MainSubMenu = Settings:settings.html;Plugins:plugins.html
```

In the above example a main menu item with a sub menu is created. The sub menu contains the menu items *Settings* and *Plugins*.

A main menu with a sub menu containing one menu item and a new sub menu:

```
[PAGES]
system.html = TEMPLATE_SYSTEM

[TEMPLATE_SYSTEM]
Template = s|<div>$TEXT</div>
TEXT = s|System menu
$MainMenuItem = System
$MainSubMenu = Settings:settings.html;Plugins:$SubSubMenu
$SubSubMenu = Create plugin:create.html;delete plugin:delete.html
```

In the above example a main menu item with a sub menu is created. The sub menu contains a menu item *Settings* and a new sub menu *Plugins*.

A sub menu can contain single menu items or new sub menus.

7.2.4 Adding a sub menu item to an existing sub menu

It is possible to add a sub menu item in any sub menu. The example below first defines a menu called *System.* In another plugin a page is defined that we want to add to the system menu. This can be done by setting the *$MainMenuItem* to the name of the main menu where the menu item should be inserted.

A main menu item is defined with a sub menu:

```
[PAGES]
system.html = TEMPLATE_SYSTEM

[TEMPLATE_SYSTEM]
Template = s|<div>$TEXT</div>
TEXT = s|System menu
$MainMenuItem = System
$MainSubMenu = Settings:settings.html;Plugins:plugins.html
```

The *$MainMenuItem* is set to System. In this cases the menu item is inserted into the System menu.

```
[PAGES]
plugin1.html = TEMPLATE_PLUGIN1

[TEMPLATE_PLUGIN1]
Template = s|<div>$TEXT</div>
$MainMenuItem = System
$MainSubMenu = Plugin1 menu:plugin1.html
```

When the menus are created the menu handler checks if a main menu already is defined. If an existing main menu item is found the sub menu item is inserted there.

7.2.5 Template description syntax

Html variables must not start with a “$” character in the template description.

In the html code the variable must start with a “$” character.

Variables starting with a “$” character in the template description are use for the menu handling.

Html variable:

<name>=<type>“|”<html string>|<plugin method>|<html file>

The web page handler uses the python Template class to replace the variables in the html code with their actual values. Templates provide simpler string substitutions as described in **PEP 292**. Instead of the normal `%`-based substitutions, Templates support `$`-based substitutions, using the following rules:

- `$$` is an escape; it is replaced with a single `$`.
- `$identifier` names a substitution placeholder matching a mapping key of `"identifier"`. By default, `"identifier"` must spell a Python identifier. The first non-identifier character after the `$` character terminates this placeholder specification.
- `${identifier}` is equivalent to `$identifier`. It is required when valid identifier characters follow the placeholder but are not part of the placeholder, such as `"${noun}ification"`.

Sub menu list:

$MainMenuItem = <main menu name>

$MainSubMenu=<sub menu list>

“$”<sub menu name>=<sub menu list>

The sub menu list contains a list of menu items separated by a semicolon and menu item values separated by a colon.

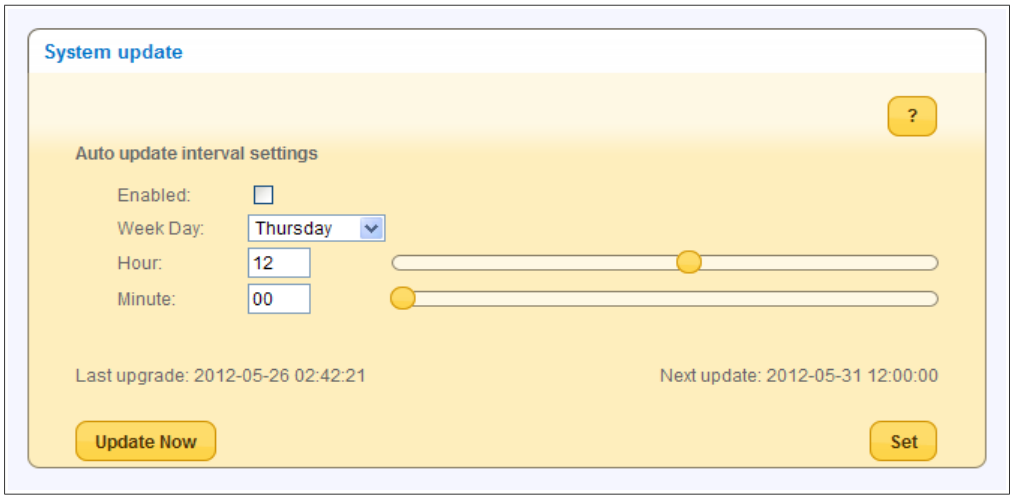
<caption>:<page|submenu>:[<sort order>];<caption>:<page|submenu>:[<sort order>]

- *caption* The text to display in the menu
- *page|submenu* The page to display when the menu item is clicked or a sub menu name
- *sort order* An optional value defining the sort order prefix for the sub menu items.

7.3 Automatic updates

This is a **preliminary** documentation of a new plugin to handle automatic updates. The plugin is supposed to downloads debian packages installs them and restarts the service if the service was upgraded.

The plugin has a user interface where the update handling can be configured.



The update handler can be enabled or disabled. A date and time can be set when the update should occur.

7.3.1 Plugin handling of updates

Some plugin modules needs to be shut down in a controlled way before the service can be restarted. To handle this situation three message handlers has been added to the framework.

- *system.update_listener_add* Publisher to register a listener for update_pending messages
- *system.update_pending* Message to inform plugins that an update is pending
- *system.update_proceed* Publisher to handle messages from update listeners

A plugin that needs to monitor when an update is in progress must send the message *system.update_listener_add* when it is initialized.

When an update is in progress the update handler will send the message *system.update_pending* to all plugin modules that has requested to be informed when an update is in progress.

When a plugin receives the *system.update_pending* message it should place it self in a state so the framework can be shut down. When the plugin is ready for shut down it must send the message *system.update_proceed.*

When the update handler has received the message *system.update_proceed* from all registered plugins it will continue with the update.

8 Client communication

The ioproxy module can be used by python clients that want to access the service. The ioproxy module is located in \devenv\commons directory.

The deproxy handles the authentication and communication with the web service.

Examle:

```
Import ioproxy

URI = 'htp://seldlx1480:8080/'
Server = ioproxy.IoPProxy(URI)

#Call Core.System.Ping
 try:
   result = Server.Core.System.Ping()
 Except Exception as e:
   #handle exception

 try:
   ping = ioproxy(URI, methodName="Core.System.Ping")
   result = ping()
 Except Exception as e:
   #handle exception

# The result returned from a method call is the return value
# from the method

# If some error occur during the method call an Exception is raised.
# The exceptions can be of different type depending on where the
# error occurred.
```

9 Directory structure

The Root directory contains documentation, scripts for running tests, creating Debian packages and other components and applications used by the service e.g. a patched adb and statistics server.

doc

Contains the auto-generated documentation and configuration files for epydoc that is used to generate the documentation. The doc directory has a subdirectory *html* where the auto-generated html version of the documentation is located.

test

Contains test code and scripts to test the modules in the src directory.

src

Contains the source code .

The **src** directory has a number of subdirectories:

- **log**: logfiles for the web server and the framework.
- **upload**: used by the web server when files are uploaded.
- **plugins**: location of the plug-in modules.
- **applications**: applications and plugin modules.
- **Pyro4**: Pyro remote procedure call library.
- **certificates**: Certificates for the Web-server, to use for testing HTTPS.

New plug-ins should be placed in an applications *plugins* directory for the application manager to be able load them when the service starts.

10 Documentation

Detailed documentation of the framework is automatically generated using epydoc.

In the doc directory there are script files to generate the documentation.

- *devicemonitor_rest.epy* is the configuration file for epydoc.
- *makedoc.sh* a shell script file to run epydoc on Linux.
- *makedoc.bat* a bat file to run epydoc on windows.

The doc directory has a subdirectory *html* where the auto-generated html version of the documentation is located.

The Jenkins verification job that is run every time something is uploaded to gerrit for code review runs epydoc to generate the documentation. The generated documentation is saved in the job artefacts and can be downloaded from there.

10.1 Running epydoc on the local computer

To be able to run epydoc on the local machine epydoc must be installed together with some extra components:

Docutils

The documentation tags used in the python files uses the *reStructuredText* markup language, it requires that Docutils 0.3 or higher is installed.

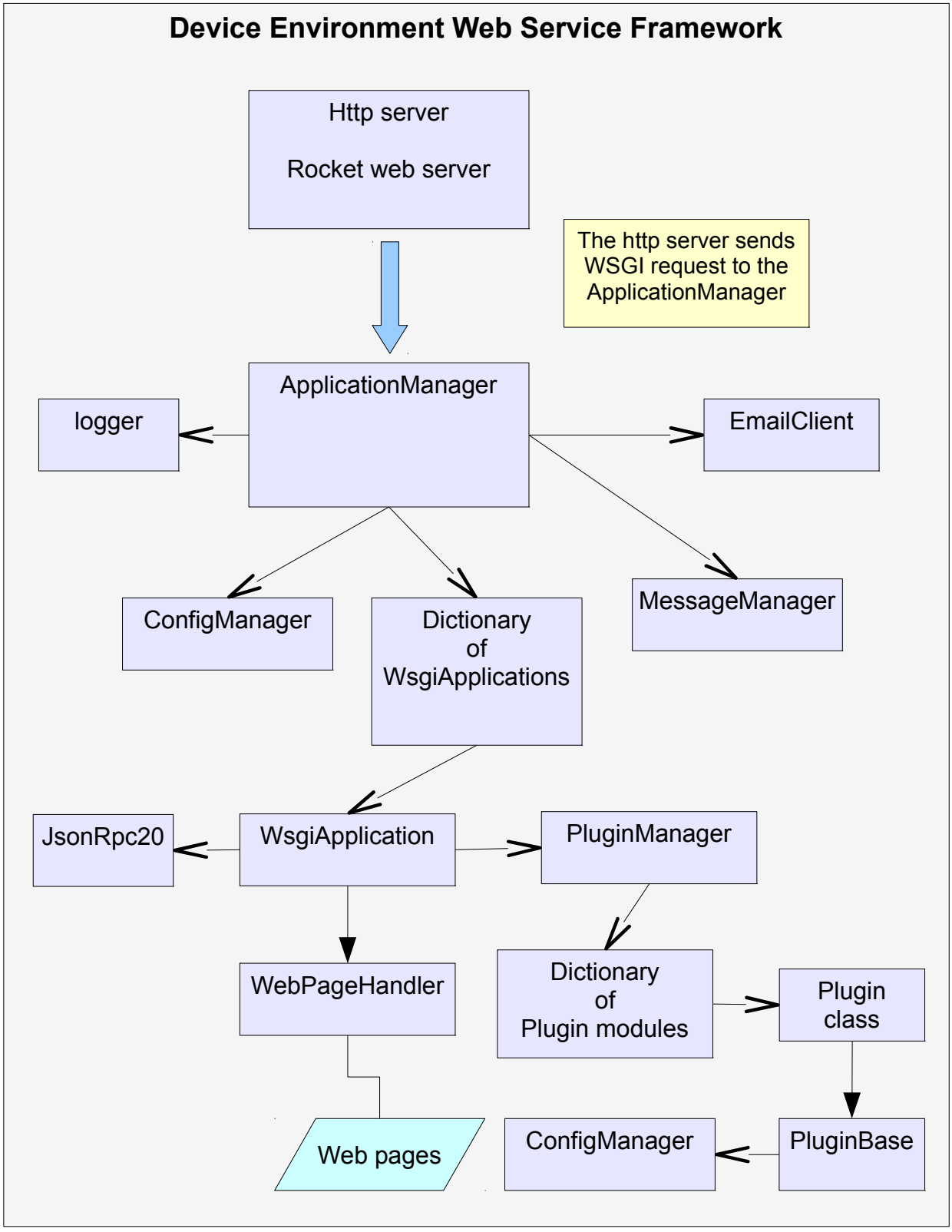
Graphviz

To be able to draw diagrams and user defined graphics Graphviz needs to be installed. Epydoc uses the Graphviz dot executable to generate graphs.

These components can be installed using the Synaptic packet manager on Ubuntu.

11 Class overview

The diagram show the main classes used by the framework.



11.1 Class summary

11.1.1 Rocket web server

The rocket web server is a small open source web server written in python and can be distributed as one python module without the need to install any extra components.

It uses the WSGI API to communicate with the web service applications.

11.1.2 ApplicationManager

The application manager handles the WSGI requests received from the server and forwards the request to an application handler. The ApplicationManager contains classes for handling application configuration, sending emails, a message handler that implements a system to send messages between plug-in modules and a dictionary of web service applications.

11.1.3 WsgiApplication

The WsgiApplication is the base class of all applications. It manages the de-serialisation of JSON-RPC request and keeps a list of plug-ins belonging to the application.

11.1.4 PluginManager

The pluginManager is a a part of the WsgiApplication and handles loading and unloading of plugin modules.

11.1.5 PluginBase

The pluginBase is the base class from which every plug-in class must inherit. It handles initializations and clean-up when a plug-in is loaded or unloaded. It also contains methods to register and send messages to other plug-in modules.

11.1.6 JsonRpc20

Is a JSON-RPC 2.0 encoder/decoder class.

11.1.7 ConfigManager

Handles configuration data stored in files, internally it uses the ConfigParser from the python library.

11.1.8 Python logger

The logger uses the logging handler from the python library, to handle logging to files or a remote TCP/IP port.

11.1.9 MessageManager

The MessageManager implements a kind of Publisher/Subscriber pattern that can be used to send asynchronous messages between modules.

11.1.10 EmailClient

The EmailClient class can connect to a SMTP server and send emails. The class can handle plain text messages as well as messages containing file attachments.

11.1.11 WebPageHandler

The WebPageHandler is an application that inherits from the WsgiApplication and implements a web interface that can be used to send web pages to a web browser. It can handle GET and POST request.

12 ioservice application

The framework is started by running the python program ioservice.py.

It can run as a Linux daemon or in the foreground for debugging.
Running on Windows, it must run in the foreground mode in a console window.

Notes:

There are currently some plugins that only runs on Linux, but most of the framework runs on both Linux and Windows.

Usage: ioservice.py [start | stop | restart | foreground] [options]

Commands:

start Run the application as daemon

stop Stop the daemon

restart Restart the daemon

foreground Run the application in a console

Options:

--version	show program's version number and exit
-h, --help	show this help message and exit
-p PURGE	remove log files and restore the default configuration.
-p1	remove log files
-p2	remove all user configuration
-p3	remove both log files and all user configuration.
-c, --config	display configuration settings

12.1 Service configuration file

Service configurations are stored in a configuration file:

- Linux: Linux_ioservice.cfg
- Windows: Windows_ioservice.cfg

The service configuration file contains server configurations, access rights settings and cluster setup.

12.2 Accessing the service maintenance pages

[http:|https:]//<server address>/index.html

Depending on if the authentication is enabled or not a user id and password has to be given before the web pages can be opened.

13 Open source Components

The framework uses some open source libraries in its framework and plugins which are listed below.

13.1 Ftp server library

Python FTP server library provides a high-level portable interface to easily write asynchronous FTP servers with Python. pyftpdlib is currently the most complete RFC-959 FTP server implementation available for the Python programming language.

It is used in projects like Google Chromium and Bazaar and included in Linux Fedora and FreeBSD package repositories.

13.2 Web server

The Rocket web server is a small web server written in python implementing the WSGI interface.

13.3 User interface components

The web user interface uses the jQuery and jQuery-UI JavaScript libraries published under the MIT-license.

13.4 MD5 and SHA256 JavaScript libraries

These libraries are used to generate MD5 and SHA256 hash sequences.

13.5 Datatables Jquery plugin

Datatables is a Jquery plugin to create formatted html tables of various types.

13.6 Python Remote Objects

Pyro is a python library used to communicate with remote python objects over a network.

14 Definitions, acronyms and abbreviations

Term	Description
API	Application Programming Interface
RPC	Remote Procedure Call
XML	Extensible Markup Language
JSON	JavaScript Object Notation
WSGI	Web Service Gateway Interface

14.1 References

[1]XML-RPC http://www.xmlrpc.com/
[2]JSON http://www.json.org/
[3]JSON_RPC http://json-rpc.org/
[4]PYTHON http://www.python.org/
[5] WSGI http://www.python.org/dev/peps/pep-0333/
[6] EPYDOC http://epydoc.sourceforge.net/
[7] UUID http://tools.ietf.org/html/rfc4122.html

15 Revision history

Date	Revision	Author	Comments