How to Make a Thunder Nano Service

Most of the up-to-data examples with the new JSONRPC can be found here : https://github.com/rdkcentral/ThunderNanoServices/tree/master/examples

This directory contains two Example plugins, called : JSONRPCPlugin and OutOfProcessPlugin.

This directory also contains an example on how to communicate from outside of the Thunder framework, to the ThunderFramework (bidirectional). The JSONRPC Client is a stand-alone application that connect to the JSONRPCPlugin.

For using the plugins, the documentation (API documentation) is pretty well documented. This is due to the fact that the documentation is automatically generated from the interface specification. The API specifications for each plugin can be found here:
https://github.com/rdkcentral/ThunderNanoServices/blob/master/README.md

Plugins can be developed in a large variety, in process, out-of-process, out-of host, and each plugin can exploit a large scale over communication protocols, JSONRPC/COMRPC, MessagePackRPC.

In **In-process**, we can share the worker pool from the WPEFramework thread pool (it will be used a process which is light weight and does not require its own process space : lot of plugins are in-process (Device Info, RemoteControl, FirmwareControl, TraceControl etc)

Out-of-process is used to run a plugin which is required to handle in a separate process context (if by any action it can affect WPEFramework process, has to be run as out of process), for example BrowserPlugin (cobalt, webkit, spark, Netflix).
It will provide its own thread pool, then also it can communicate with the WPEFramework using COMRPC

OUT-of Host can be run outside the device and connect with the WPEFramework externally. COMRPC is used to communicate between process and it should be used if there is large data. Currently there is no existing plugin, only example plugin: https://github.com/WebPlatformForEmbedded/ThunderNanoServices/tree/master/examples/RemoteHostExample

COMRPC is used to communicate between the plugins (out of process) or to communicate for larger data
https://github.com/WebPlatformForEmbedded/ThunderNanoServices/tree/master/examples/COMRPCClient

JSONRPC is used to fetch/update info to or from plugins externally (most of the plugins provide this in interface, similar to ReST API)
also it can be used from applications : https://github.com/WebPlatformForEmbedded/ThunderNanoServices/tree/master/examples/JSONRPCClient
It will be used only for small data

Message pack also, similar to JSONRPC, we can have example at https://github.com/WebPlatformForEmbedded/ThunderNanoServices/tree/master/examples/JSONRPCClient plugin. But this feature is not exposed outside through the WPEFramework Process. It can implemented by using another application as like https://github.com/WebPlatformForEmbedded/ThunderNanoServices/blob/master/examples/JSONRPCPlugin/JSONRPCPlugin.h#L149

Steps involved in implementing new Thunder Plug-In

Interface Specification in wpeframework

1. **<PluginName>.json**

Add <PluginName>.json in wpeframework module so as to define the interfaces.

https://github.com/WebPlatformForEmbedded/Thunder/tree/master/Source/interfaces/json

eg: DeviceInfo.json

After adding the json file and compilation of wpeframework, it will autogenerate header file JsonData_PluginName.h

eg: JsonData_DeviceInfo.h

This header will be used from wpeframework-plugins module.

Plugin Development in wpeframework-plugins

In wpeframework-plugins workspace :

(cloned from https://github.com/WebPlatformForEmbedded/ThunderNanoServices/)

Create PluginName folder

Inside PluginName directory:

1. Create “**CmakeLists.txt**” to compile the Plug-in code and to generate the shared library (“.so”)

This will handle all the dependencies as well

2. **Module.h:** This header file includes the support for JSON request, response, logging etc.,

3. **Module.cpp:** This file is used to declare the module name for the Plug-in

4. <**PluginName>Plugin.json:**

This file contains the plugin's information like schema, information and interface json file (defined earlier)

Ex:-

```
{
  "$schema": "plugin.schema.json",
  "info": {
    "title": "Plugin Name Plugin",
    "callsign": "PluginName",
    "locator": "libWPEFrameworkPluginName.so",
    "status": "production",
    "description": "The PluginName plugin allows retrieving of various plugin-related information.",
    "version": "1.0"
  },
  "interface": {
    "$ref": "{interfacedir}/PluginName.json#"
  }
}
```

5. <**PluginName>.config:** This file is used to set configurations of the Plug-in

Ex:- *set (autostart true)*

Used to make the Plug-in to start automatically along with wpeframework daemon

We can set some other parameters based on our need

6. <**PluginName>.h**

Declare the plugin class in this which should contains all the structures, variables and methods which are needed for plugin implementation. The interface header auto-generated earlier will be used here,

Ex:- Declare the class in the following name space with constructor and destructor:

```
namespace WPEFramework {
        namespace Plugin {

                class PluginName : public PluginHost::IPlugin, public PluginHost::IWeb, public PluginHost::
JSONRPC {
                public:
                                PluginName()
                                        : _skipURL(0)
                                        , _service(nullptr)
                                        , _subSystem(nullptr)
                                {
                                        RegisterAll();
                                }

                                virtual ~PluginName()
                                {
                                        UnregisterAll();
                                }
                }
```

Declare the methods in the above class, required to implement the functionality of the plugin

These methods are used to place collection of plugin JSON interface methods to register & unregister with JSON RPCs APIs

```
void RegisterAll();
void UnregisterAll();
```

These methods are used to initialize and deinitialize the handlers for the plug-in service

```
virtual const string Initialize(PluginHost::IShell* service);
virtual void Deinitialize(PluginHost::IShell* service);
```

These are are the JSON interface (get/set) methods to communicate with plugin

```
uint32_t get_method(JsonData::Plugin::ClassName& response) const;
uint32_t set_method(JsonData::Plugin::ClassName& response) const;
```

This method is used to process the REST APIs request such as GET/POST/SET and return the response

```
virtual Core::ProxyType<Web::Response> Process(const Web::Request& request) override;
```

7. <**PluginName>.cpp:** This class does contains all the definitions for the methods declared in the Plugin.h and those definitions should be defined inside the below namespace.

The plugin should register using service registration MACRO as declared below:

```
namespace WPEFramework {
        namespace Plugin {
                SERVICE_REGISTRATION(Plugin, 1, 0);
                //All the methods declared in Plugin.h should be defined here
        }
}
```

8. <**PluginName>JsonRpc.cpp***: This class is used to register the methods with JSON RPC interface as below*

Ex:-

```
namespace WPEFramework {

namespace Plugin {

        using namespace JsonData::PluginName;

    void Plugin::RegisterAll()
    {
                //Can register any number of methods in this way
                Property<className>(_T("parameter1"), &PluginName::get_method, nullptr, this);
                Property<className>(_T("parameter2"), &PluginName::set_method, nullptr, this);
    }

        void Plugin::UnregisterAll()
    {
            Unregister(_T("parameter1"));
        Unregister(_T("parameter2"));
        }
}
}
```

The registered (get / set) methods are defined in the same file

```
uint32_t Plugin::get_method(ClassName& response) const
{
        //body of the method
}

uint32_t Plugin::set_method(ClassName& response) const
{
        //body of the method
}
```

9. <**PluginName>HAL.cpp:** Used to communicate to driver layer in order to get some information or to set some properties.

Reference

Please refer to any existing plugins in https://github.com/WebPlatformForEmbedded/ThunderNanoServices/

or in workspace of RDK4.0 Beta build:

(***rdkv_4.0_beta/build-raspberrypi-rdk-hybrid-thunder/tmp/work/cortexa7t2hf-neon-vfpv4-rdk-linux-gnueabi/wpeframework-plugins/3.0+gitrAUTOINC+886f9025b3-r1/git/***)

for further clarifications

Compilation and Install

Enable the plugin in the main CMakeLists.txt of wpeframework-plugins

bitbake wpeframework-plugins

will generate <PluginName>.json and libWPEFrameworkPlugin.so

Copy the plugin library (libWPEFrameworkPlugin.so) to **“/usr/lib/wpeframework/plugins”**

Copy the Plugin.json file to “**/etc/WPEFramework/plugins”** so that the controller plugin identify it and list it in the WebUI (controller UI)

Restart the service

systemctl restart wpeframework

Using the command **“$ journalctl – u wpeframework | grep <plug-in name>”** we can identify, if the newly added plug-in got activated or not

If the plug-in got activated, it will be listed in the WebUI and using Controller plug-in we can control (activate / deactivate) the newly added plugin

Validation

REST APIs based curl command:

Request:

$ *curl --request GET http://127.0.0.1:9998/Service/<pluginName>/*<function>

eg: $ curl --request GET http://127.0.0.1:9998/Service/Picture/Brightness

Response:

{"brightness":100}

JSON RPC APIs based curl command:

Request:

$ curl --data-binary '{"jsonrpc": "2.0", "id": 3, "method": "Picture.1.brightness"}' http://127.0.0.1:9998/jsonrpc

Response:

{"jsonrpc":"2.0","id":3,"result":{"brightness":100},"success"}