metrological

Thunder

API Reference

History

Version	Date	Author	Description
0.1	7-10-2017	P. Wielders	Initial version
0.2	30-05-2018	P. Wielders	Subsystems
0.3	04-07-2018	C.Custers	Added Reload Configuration API
0.4	05-07-2018	M.Fransen	Updated Controller Configuration
0.5	12-10-2018	C.Custers	Update References + Adjust Layout
0.6	31-10-2018	C.Custers	Add root-object + Fix typos
0.7	26-02-2019	P.Wielders	JSONRPC support.
0.8	07-03-2019	P.Wielders	Update after review on JSONRPC
0.9	14-04-2019	M. Fransen	Small JSONRPC corrections
0.91	22-08-2019	M. Fransen	Describe resumed state option for services
0.92	07-11-2022	V. Aslan	Minor update to state section
1.0	13-11-2024	V.Aslan	Thunder Renaming

Table of Contents

1. Introduction

1.1 Scope

This document describes the Thunder API interface which can be used to control/query plugins configured and build for the Thunder product range. This document describes the Thunder concepts and provides a definition of the API.

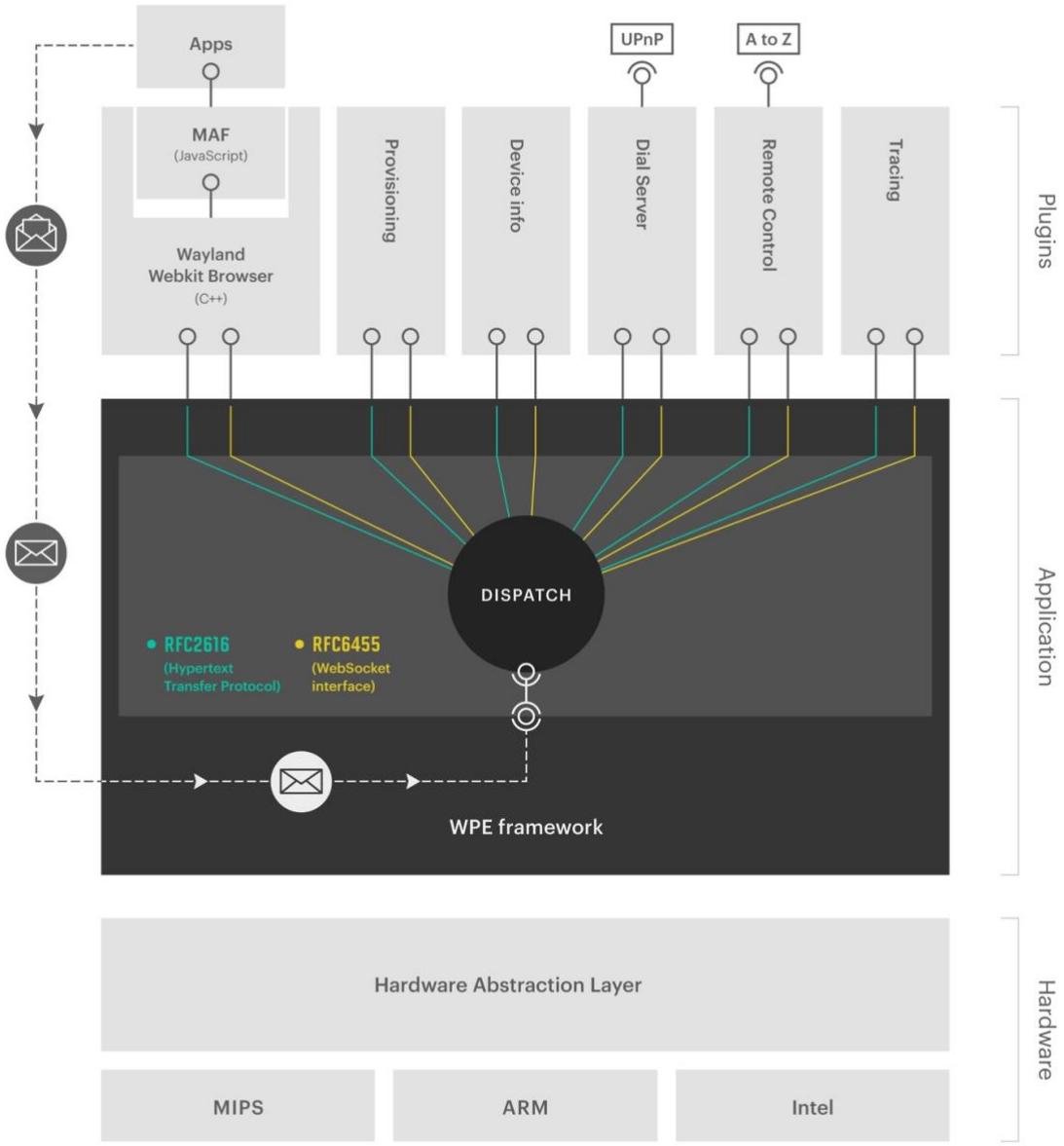

Figure 1. Global overview of the Thunder

1.2 Context

Thunder provides a unified web-based interface with a consistent navigation model. In this model, plugins (custom or generic) are controlled and queried, through the Thunder application.
The main responsibilities of Thunder application are:

- Modular loading and unloading of plugins.
- Plugin process localization. In or out-of-process communicating with the framework over a lightweight RPC communication channel.
- Runtime enabling/disabling of tracing information within the plugins and the Thunder application.
- Light-weight implementation of the HTTP [RFC2616] specification.
- Light-weight implementation of the WebSocket [RFC6455] specification.

Each instance of a plugin in the WPE id identified by a name. This name is referred to as ***Callsign*** of the plugin. The callsign must be unique in the context of all configured plugins.

1.3 Case sensitivity

All identifiers on the interface described here are case-sensitive. e.g. an id known in the plugin as 'C0FFEE' is not the same as 'c0ffee'.
All keywords, entities, properties, relations and actions should be treated as case-sensitive.

1.4 Acronyms, Abbreviations and Terms

The next list provides an overview of acronyms and abbreviations used in this document and their definitions.

Acronym	Definitions
API	Application Programming Interface
JSON	JavaScript Object Notation
UTC	Coordinated Universal Time

Below terms are listed with their definitions, as used in this document.

Term	Definitions
Callsign	The callsign is the name given to an instance of a plugin. One plugin can be instantiated multiple times, but each instance the instance name, callsign, must be unique.
Proxy	An object in one process space representing the “real” object in another process space. The Proxy takes care of marshalling the parameters.
Stub	An object in the process space that contains the actual object. The stub takes care of un-marshalling the request from the Proxy and executes the call, on behave of the Proxy object, on the real object

1.5 Standards

Date time formats between the systems shall be in UTC time and W3C ([ISO-8601] profile) formatting [ISO-8601], e.g.: 2004-11-05T13:15:30Z. This way time discontinuities can be avoided due to daylight savings. Note that all interfacing systems must decode/encode the date time to the correct local time. Languages used in the Thunder will be conform [ISO-639-2] using two letter language codes. If Thunder encounters a language code it does not recognize, it will use 'xx' instead. For a list of available two letter ISO language codes, please visit: [ISO-639-2].

1.6 References

This section lists the references made in this document:

[HTTP]	Hypertext Transfer Protocol
[ISO-8601]	Date and time format
[ISO-3166]	Country code specification
[ISO-639-2]	Language code specification (Alpha-2 code)
[JSON]	JavaScript Object Notation
[URLENC]	URL Encoding

1.7 Open Issues

This is a list of open issues that needs to be resolved:

- This document is still a work in progress.

1.8 Limitations

The information described in this document is preliminary and subject to change in the future.
Legend:

Be aware of: implementation choice is needed or side-effect needs to be handled.

Implementation advice: Guide line for implementation mostly related to performance.

2. Basic Concepts

2.1 JSONRPC API calls

This chapter discusses the basic concept of the Thunder API interface from a client perspective. The concept can be applied using the body of the RESTFul API (as described in §2.2 RESTful API calls [deprecated]) or as a JSON message over the Web Socket(as described in §2.3 Web sockets).

The JSONRPC message format is conform https://www.jsonrpc.org/specification and this specification is using JSON conform RFC4627.

2.1.1 Designator definition

The designator is split up in three elements:

The callsign is a text that is made up of [‘A’..’Z’, ‘a’..’z’,’0’..’9’,’.’]
The version is a number that is made up of [‘0’..’9’]
The method is a text that is made up of [‘A’..’Z’, ‘a’..’z’,’0’..’9’]

The callsign, with or without the version, indicate that the method should be handled by the plugin indicated by the given callsign. The optional version number, indicates that it is expecting that the destination does support that version. If the version is specified, but not supported, an error will be returned.
It is allowed to send a [version.]<method> only, if the communication endpoint resides on the given callsign.

2.1.2 Synchronous Invocation

Checking for the existence of a method:

```
{
  “jsonrpc”: “2.0”,
  “id”: 3,
  “method”: “firebolt.displaysettings.1.exists”,
  “params”: “setresolution”
}
```

```
{
  “jsonrpc”: “2.0”,
  “id”: 3,
  “result”: 0
}
```

Result values are:

ERROR_NONE:	0	// All OK!
ERROR_UNAVAILABLE:	2	// The plugin does not exist
ERROR_UNKNOWN_KEY	22	// The method does not exist
ERROR_INVALID_SIGNATURE:	38	// The plugin exists but the version is **not** supported.

Example for setting the display to another surface geometry:

```
{
  “jsonrpc”: “2.0”,
  “id”: 4,
  “method”: “firebolt.displaysettings.1.setresolution”,
  “params”: { “width”: 1920, “height”: 1080 }
}
```

```
{
  “jsonrpc”: “2.0”,
  “id”: 4,
  “result”: 0
}
```

2.1.3 A-Synchronous notifications/callback

```
{
  “jsonrpc”: “2.0”,
  “id”: 5,
  “method”: “firebolt.ioconnector.1.register”,
  “params”: { “event”: “powerpin”, “id”: “client.events.1” }
}
```

```
{
  “jsonrpc”: “V2.0”,
  “id”: 5,
  “result”: 0
}
```

This triggers a registration on the server. In case the pin is changing state, a method invocation is triggered from the server. This will be an id-less invocation so no answer expected:

```
{
  “jsonrpc”: “2.0”,
  “method”: “client.events.1.powerpin”,
  “params”: { “state”: “low” }
}
```

To end the notifications, the event needs to be unregistered. This can be done by invoking the unregister on the server side.

```
{
  “jsonrpc”: “2.0”,
  “id”: 6,
  “method”: “firebolt.ioconnector.1.unregister”,
  “params”: { “event”: “powerpin”, “id”: “client.events.1” }
}
```

```
{
  “jsonrpc”: “2.0”,
  “id”: 6,
  “result”: 0
}
```

2.1.4 JSONRPC over HTTP RESTFul API

JSONRPC can use the HTTP RESTFul interface. This communication channel can only realize synchronous invocation (see §2.1.2 Synchronous Invocation). The JSONRPC message is carried in the body. The RESTFul API call is specified as:

Request:	`POST /jsonrpc[/Callsign]` `BODY: JSONRPC according to RFCXXXX`
Success:	`HTTP/1.1 200 OK`

Failure	`HTTP/1.1 403 Method does not exist`
	`HTTP/1.1 404 Callsign does not exist`
	`HTTP/1.1 405 Method version does not exist`
	`HTTP/1.1 500 No JSONRPC interface available on the given Callsign`

The HTTP RESTFul call can be directed to the required plugin by adding the call sign in the HTTP path header. If the request is directed to the specific plugin that should invoke the requested method, the method in JSONRPC message can be limited to method name only. If the Callsign is **not** present in the HTTP path header, the request will be handled by the controller plugin. If the method needs to be invoked by a different plugin, please use the designator definition, as found in §2.1.1.

2.1.5 JSONRPC over WebSocket API

JSONRPC can use the WebSocket interface. This communication channel can realize synchronous invocation (see §2.1.2 Synchronous Invocation) and a-synchronous communication (see §2.1.3 A-Synchronous notifications/callback). The JSONRPC message are send as text over the websocket.
A Websocket can be opened directly to the plugin or it can be opened to the Controller. Directly means that socket can only communicate with the designated (Callsign) plugin. The method, in the JSONRPC message, can be issued without the callsign, if the websocket is opened directly. If the socket is opened towards the controller, the method should use the designator definition, as found in §2.1.1 Designator definition) to reach a specific plugin for method invocation.

Request:	`ws://<IP Address and port of Thunder>/jsonrpc[/Callsign]`
Success:	`HTTP/1.1 200 OK`

2.2 RESTful API calls [deprecated]

This chapter discusses the basic concepts of the Thunder API interface.
All request and response bodies should use the JSON format for data.

Methods:

Method	Function
GET	Retrieve information from Thunder or a plugin
POST	Update new information or new objects at Thunder or a plugin
PUT	Create new information or new objects at Thunder or a plugin
DELETE	Delete information at Thunder or a plugin

For example, the following call retrieves information from the WebKitBrowser plugin:

```
GET / Service/WebKitBrowser HTTP/1.1
```

Web API paths:

All Thunder commands start with the “Service” prefix followed by the Plugin name:

```
(GET|POST|PUT|DELETE) /Service/<PluginName>[/OptionalPaths] HTTP/1.1
```

For example, to retrieve information from the Controller plugin:

```
GET /Service/Controller HTTP/1.1
```

This returns a JSON list with plugins and their state as managed by the controller:

```
HTTP/1.1 200 OK

Content-Type: application/json

Content-Length: 5036

Cache-Control: no-cache, private, no-store, must-revalidate, max-stale=0,
post-check=0, pre-check=0 Access-Control-Allow-Origin: *


{"plugins": [ ... ]}
```

2.3 Web sockets

Not all actions initiated on a plugin need to be deterministic. For example, getting a time in an NTP plugin requires network access and the response of the network is not deterministic. The network server might respond in 100ms but it might also take some seconds.

The RESTful API, using the request->response algorithm, is not suited for non-deterministic actions, as a rule of thumb, a request should return a response within a second. But what if the network connection is slow, or takes several retries and the network time is only received after 10 seconds?

The architecture of the Thunder offers event based feedback to clients. This functionality is realized using web socket connections to a plugin. Over these web socket connections, the plugin will notify the client on the other side of the web socket of events specific to the plugin.

The syntax for opening a web socket to the plugin is equal to the syntax of the RESTful API:

Request:	`ws://<IP Address and port of Thunder>/Service/<Callsign>` `protocol: notification`
Success:	`HTTP/1.1 200 OK`

In JavaScript, the socket would be opened like:

```
socket = new WebSocket("ws://192.168.1.100:9999/Service/Controller", "notification");
```

Over this connection, the JavaScipt application, opening this socket will receive events from the controller plugin. See plugin details in the sections: 0 .

2.4 Subsystems

A subsystem is a unit of functionality that might be required by plugins before that plugin can operate properly. As an example, a Provisioning plugin, receiving its provisioning information from the cloud, can only work properly if there is internet connectivity. The internet subsystem represents functionality in the system that allows network connectivity towards the open internet (and thus the cloud). This subsystem is only available, if the system has detected network connectivity with the internet.

Currently the Thunder supports the following subsystems (PluginHost::ISubSystem):

```
enum subsystem {
    PLATFORM = 0,          // platform is available.
    NETWORK,               // Network connectivity has been established.
    IDENTIFIER,            // System identification has been accomplished.
    GRAPHICS,              // Graphics screen EGL is available.
    INTERNET,              // Network connectivity to the outside world has been
    LOCATION,              // Location of the device has been set.
    TIME,                  // Time has been synchronized.
    PROVISIONING,          // Provisioning information is available.
    DECRYPTION,            // Decryption functionality is available.
    WEBSOURCE,             // Content exposed via a local web server is available.
}
```

Thunder offers an event driven interface framework for subsystem availability signaling to the plugins. This framework is utilized to realize concurrent startup scenario using a conditional evaluation only in case a condition changes. This allows for startup without requiring sleeps, or resource consuming polling mechanisms.
The startup dependency can be set in the precondition field of the plugin configuration, as can termination conditions be set in the termination field. Note that the Thunder will by default set all subsystems to activated when starting up to prevent plugins waiting indefinitely for the conditional evaluation to be triggered. In the Controller configuration the subsystems provided by the system can be configured to disable the automatic subsystem signaling by the Thunder for these subsystems.
The value associated with each subsystem is depicted below the negated value is the same but than the value should be prefixed with an exclamation mark (!).

SUBSYSTEM	CONFIGNAME
PLATFORM	Platform
NETWORK	Network
IDENTIFIER	Identifier
GRAPHICS	Graphics
INTERNET	Internet
LOCATION	Location
TIME	Time
PROVISIONING	Provisioning
DECRYPTION	Decryption
WEBSOURCE	WebSource

Table 1 Subsystem to configname

3. Thunder Application

3.1 Startup parameters

3.2 Configuration

The configuration for the Thunder covers for a flexible deployment. Most behavior and file locations can be configured.

3.2.1 Main configuration

version	String stating the Human Readable string associated with this delivery. **Default**: “”
model	Model identification that is returned in multicast broadcasts to identify this device. **Default**: “”
port	Number identifying the port on which the TCP RESTful server should be listening. **Default**: 80
binding	IP address to associate the TCP RESTful server with. This parameter is mutual exclusive with the interface parameter. **Default**: AnyInterface
interface	Interface to associate with the TCP RESTful server with. This parameter is mutual exclusive with the binding parameter.
prefix	This is the prefix to indicate the default sstart path for accessing the RESTful service. **Default**: “Service”.
persistentpath	Path to the location where information can be stored over reboots. The path is always postfixed by the callsign of the plugin. <persistentpath>/<callsign>/ under the callsign PluginHost, a file called override.json can be found. Here the Thunder stores the configuration of all configured plugins together with the autostart flag. These settings “overrule” the once found in the configuration files (read only)
datapath	Path to the location where read-only data is stored, this path is post-fixed with the plugins classname. <datapath>/<classname>/
systempath	Path to the location where the plugins will be searched, if not found on the persistent path.
volatilepath	Path to use in case volatile storage is required. This storage is cleared after reboot.
proxystubpath	Path to where the Proxy Stubs can be found. All the files found on this path, with the extension *.so will be loaded
configpath	Path to the JSON configuration files. Each JSON file is describing the configuration for a plugin instance. If no value is given, the Thunder will search the directory called plugins, found next to the location where this configuration file is located.
ipv6	Boolean to force IPv4 networking only, if set to false. Default: false.

Tracing	Array of traces that should be enabled or disabled from startup.
Redirect	If no path is given, if a RESTful API is called, it will be redirected to the path given here. **Default**: /Service/Controller
Process	JSON object to specify process properties for the Thunder parent process to be started. See 3.2.3 for the fields
Input	Enumerate specifying how inserted keys should be forwarded.
communicator	Default path to reach the RPC communication channel of Thunder. Each newly created out-of-process instance will hookup, using this identifier to the Thunder. Over this channel, RPC invoke actions will take place.
Plugins	Array of potential plugin configuration. The semantics of these JSON structs is equal to the ones found in the configpath, with the exception of the callsign. Where the name of the JSON file found in the configpath, will be the callsign, here it is the callsign keyword identifying the callsign to be used for this instance.

3.2.2 Plugin

callsign	Name identifying this instance of the plugin.
classname	Class name that should be instantiated to get the default IPlugin interface.
locator	Name of the SO (shared object) to be loaded to find the given classname.
autostart	Should this plugin be activated during startup time, or will it be started later via the RESTful API. **Default**: true. As can be seen in paragraph 4.1 a plugin will start in the Suspended state by default. A plugin will start on system startup in the Resumed state however when autostart is set to true and the callsign of the plugin was added to the resumes list of the Controller (see paragraph 4.2) **or** resumed for this plugin is set to true (see below)
resumed	Determines if the plugin when being started, is started in the Resumed state. **Default:** off (so meaning by default the plugin starts in the Suspended state). Note this is independent whether the plugin is started because of system startup or started by any other means (e.g. by the RESTful API). If resumed is set to true it will always start in Resumed state.
precondition	List of subsystem states to comply to before activating this plugin (AND).
termination	List of subsystem states that trigger the Deactivation of this plugin (OR).
configuration	JSON object specifying the exact configuration for this plugin. See the documentation of the specific plugin for details.
root	JSON object specifiying RPC configuration of this plugin. This object is contained inside configuration-object. See the documentation of the specific plugin for details.

3.2.3 Process

priority	Relative adjustment of the niceness level of the parent process (-20 : +19)
policy	Scheduler policy to be applied (Batch, Idle, FIFO, RoundRobin or Other)
oomadjust	Relative adjustment of the OOMScore (-10 : +10)
stacksize	The default stack size to be pre-allocated for a new thread to be started.

4. Controller Plugin

Thunder is the application that loads, initializes, de-initializes and unloads modules, by configuration. Each module realizes a specific functionality. Depending on the deployment different modules will be deployed/configured and loaded. One of the plugins that is the exception to the rule of loading/unloading, is the controller.
The controller plugin is part of the Thunder and cannot be activated/deactivated and thus not be unloaded. This is the plugin that Controls (Activate/Deactivate) the configured plugins. As this is an integral part of Thunder, it will be described in this document. All other plugins have their own document, describing their configuration and their API.

4.1 Activate/Deactivate state-diagram

Each plugin in the Thunder goes through a sequence of states when it is being Activated or Deactivated. The Activation and Deactivation of a plugin can be triggered via the Controller Plugin API (see Controller API**Error! Reference source not found.** for details). The next diagram depicts the state diagram for Activation and Deactivation of a plugin.

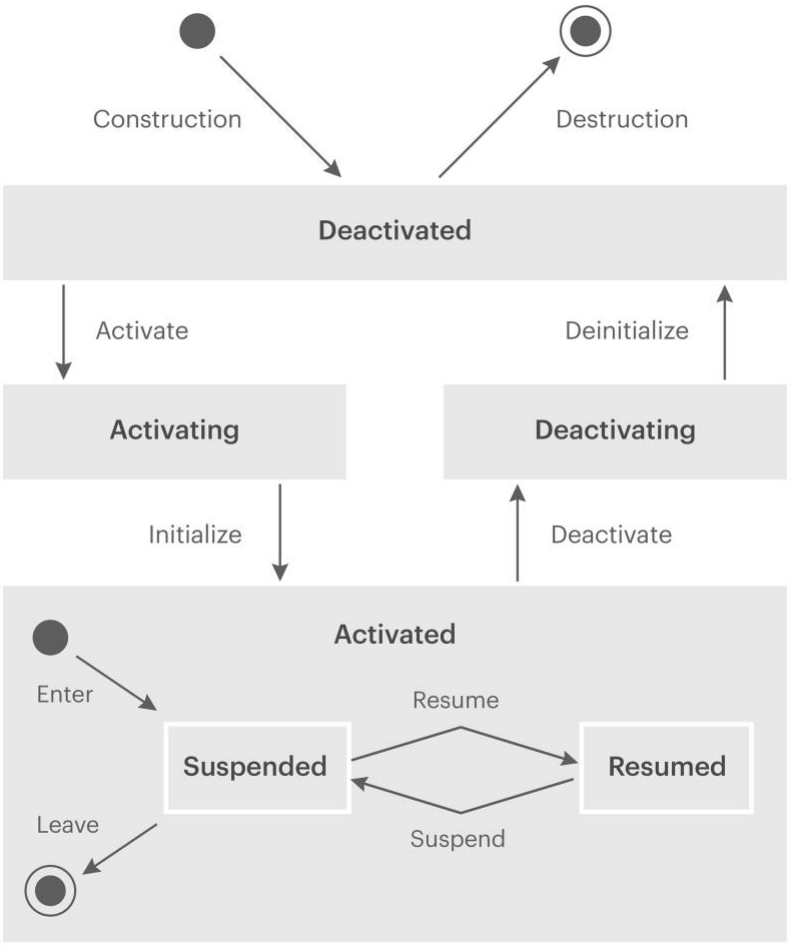

Figure 2. Plugin state diagram

The states Deactivated, Activating, Activated and Deactivating are controlled by the Controller plugin. The sub-states, suspended and resumed are controlled by the plugin self. To change these states, the API on the plugin needs to be called. For the details, see the specific plugin that have suspend and resume behavior.

4.2 Configuration

callsign	[string]	the instance name for the controller plugin. If no name is given, the default name will be “Controller”.
classname	[string]	should not be available or is an empty string.
locator	[string]	should not be available or is an empty string.
autostart	[bool]	should not be available or true. The controller is **always** activated.
configuration	[JSON]	JSON object specifying the exact configuration for this plugin. See the next paragraph for details.

Configuration of the controller:

ttl	[uint8]	Defines the TTL to be set on the broadcast package for Thunder discovery in the network. Default: 1.
resumes	[string]	JSON Array of call signs that have an IStateControl interface (suspend/resume behavior) and need a resume at startup. By definition, if a plugin supports the IStateControl interface and the “autostart” is set to true, it means that the plugin gets activated at startup, but remains in the suspended state. If the call sign of such a plugin is in this list, during startup, it automatically gets resumed.
subsystems	[string]	JSON Array of Subsystems configured for this system. The Thunder will not automatically signal these as active when the Thunder is being started in case for example when this signal is already provided by an actual plugin which implements the subsystem functionality.

4.3 Application Programming Interface (API)

4.3.1 Thunder status information

Using this method all information, related to a links, plugins, and running servers can be requested.

Request:	`GET /Service/Controller`
Success:	`HTTP/1.1 200 OK` `{ { link_info}, { plugin_info }, {process_info} }`

4.3.2 Plugin Information

Using this method all information, related to a plugin can be requested.

Request:	`GET /Service/Controller/Plugin/<Callsign>`
Success:	`HTTP/1.1 200 OK` `{ plugin_info }`

4.3.3 All Plugins Information

Using this method information on all plugins can be requested

Request:	`GET /Service/Controller/Plugins`
Success:	`HTTP/1.1 200 OK` `{ plugins_info }`

4.3.4 Link information

Using this method all information, related to a link can be requested.

Request:	`GET /Service/Controller/Links`
Success:	`HTTP/1.1 200 OK` `{ link_info }`

4.3.5 Environment Value

Using this method the value of an Environment variable can be requested

Request:	`GET /Service/Controller/Environment/<Variable Name>`
Success:	`HTTP/1.1 200 OK` `{ variable value }`
Failure:	`HTTP/1.1 204 Environment variable does not exist`

4.3.6 Configuration String

Using this method the configuration for a service can be requested

Request:	`GET /Service/Controller/Configuration/<Service Name>`
Success:	`HTTP/1.1 200 OK` `{ configuration JSON string }`

4.3.7 Process information

Using this method information on the process can be requested

Request:	`GET /Service/Controller/Process`
Success:	`HTTP/1.1 200 OK` `{ process information }`

4.3.8 Discovery Information

Using this method discovery information can be requested

Request:	`GET /Service/Controller/Discovery`
Success:	`HTTP/1.1 200 OK` `{ Discovery information }`

4.3.9 Subsystems

Using this method the status of the Subsystems can be requested

Request:	`GET /Service/Controller/SubSystems`
Success:	`HTTP/1.1 200 OK` `{ Status of Subsystems }`

4.3.10 Activate Plugin

Using this method, a plugin can be activated, move from Deactivated, via Activating to Activated state. If a plugin is in the Activated state, it can handle RESTful requests coming in. Only if it gets activated, the plugin gets loaded into memory.

Request:	`PUT /Service/Controller/Activate/`*`<Callsign>`*
Success:	`HTTP/1.1 200 OK`

Failure	`HTTP/1.1 304 Current state of the plugin does not allow an activate`
	`HTTP/1.1 403 You cannot activate the controller`
	`HTTP/1.1 500 Plugin could not be started, configuration is incorrect`

4.3.11 Deactivate Plugin

Using this method, a plugin can be deactivated, move from Activated, via Deactivating to Deactivated. If a plugin is Deactivated, the actual plugin (so) is no longer loaded into the memory of the process. In a deactivated state, the plugin will not respond to any RESTful requests.

Request:	`PUT /Service/Controller/Deactivate/<Callsign>`
Success:	`HTTP/1.1 200 OK`
Failure	`HTTP/1.1 304 Current state of the plugin does not allow an activate` `HTTP/1.1 403 You cannot activate the controller`

4.3.12 Reload Configuration

Using this method to reload Configuration.

Request:	`PUT /Service/Controller/Configuration`
Success:	`HTTP/1.1 200 OK`
Failure	`HTTP/1.1 304 Configuration could not be reloaded, due to malformed syntax`

4.4 Events

Events are autonomous events, triggered by the internals of the plugin. These events will be broadcasted as JSON to all the connected web socket connections that where opened to this plugin.

4.4.1 Controller notification forwarder event

The Controller plugin is an aggregator of all the events triggered by the specific plugin. All notifications send by a specific plugin are forwarded over the Controller socket as an event. The event is capsulated in the following JSON:

Callsign	[string]	callsign of the originator of this event.
data	[JSON]	the JSON that was broadcasted as an event by the plugin, designated by the callsign.

4.4.2 State change event

Requesting a state change is A-synchronous. The actual state transition (2 of the outer states, see 4.1) are reported as a state change.

Callsign	[string] callsign of the plugin that changed state.
State	[enum] new state of the plugin: • Activated • Deactivated
Reason	[enum] what caused the state change: • Requested • Automatic • Failure • MemoryExceeded • Startup • Shutdown

4.5 JSON definitions

4.5.1 Plugin information (plugin_info)

state	[enum] state of this plugin: • Deactivated • Deactivation • Activated • Activation • Suspended • Resumed
processedrequest	[uint32] the number of RESTful API requests that have been processed by this plugin.
processedobjects	[uint32] the number of objects that have been processed by this plugin.
observers	[uint32] the number of observers currently observing this plugin (web sockets)
module	[string] name of this plugin from a module perspective. Used in tracing.
hash	[hash] sha256 hash identifying the sources from which this plugin was build.

4.5.2 Link information(link_info)

state	[enum] state of this link: • WebServer • WebSocket • RawSocket • Closed • Suspended
Id	[uint32] a unique number identifying this connection.
remote	[string] the IP address (or FQDN) of the other side of the connection.
activity	[bool] indicates if there was activity during this timeslot.