How to create your first plugin with Ezlo LUA API

- Overview
- Plugin configuration
- Plugin initialization
- Events handling
- Device adding
    - Device and item mappings
    - Full device generation process
    - Start device adding
- Device removing
    - Single device removing
    - ZWave net reset
- Item value update
- Device controlling
- Global state saving
    - Save simple data
    - Save table data
    - Get saved data
    - Remove data
- Full plugin example
- API documentation

Overview

This document describes main tips to create your first lua plugin. This simple plugin will be based for ZWave protocol. But all steps will be equivalent for other protocols.

Plugin configuration

First thing we should do is to create a "config.json" file with plugin configuration.

config.json

```
{
  "id": "my_first_plugin",
  "version": "1.0",
  "type": "gateway",
  "dependencies": {
    "firmware": "1.0",
    "addons": [
      {
        "id": "zwave",
        "version": "1.0"
      },
      {
        "id": "lua",
        "version": "1.0"
      }
    ]
  },

  "permissions": [
      "zwave",
      "core",
      "storage",
      "timer"
  ],

  "startup": "scripts/startup",
  "teardown": "scripts/teardown",

  "gateway": {
    "name": "my_first_plugin",
    "label": "my gateway",
    "reason": "Start bus",
    "unreachableReasons": [
      {
        "reason": "1. No power.",
        "advice": "Please, check the device is properly powered. If it is a battery device, check the battery."
```

```
      },
      {
        "reason": "2. Out of range.",
        "advice": "Maybe device is out of Zigbee range?"
      },
      {
        "reason": "3. Device malfunction.",
        "advice": "Try to exclude/include the device."
      }
    ],
    "unreachableActions": [
      {
        "label": "Delete",
        "method": "start_exclude"
      },
      {
        "label": "Replace",
        "method": "replace_failed_node"
      }
    ],
    "setItemValueCommand": "HUB:my_first_plugin/scripts/set_item_value"
  }
}
```

Parameter name	Description
id	Unique id of plugin or unique name of plugin(used for dependencies resolving)
version	Plugin version(used for dependencies resolving)
type	"gateway" - implementation of "smart" protocol(ZWave, ZigBee, Nest, etc) "extension" - additional functionality plugin
dependencies	Required plugins and addons for this plugin
firmware	Required firmware version

addons	Required addons
id	Addon ID
version	Addon version
permissions	Script modules which needs for working this plugin
startup	Script which will be run when all dependent modules will be started
teardown	Script which will be run before removing of plugin
gateway	Gateway functionality description
name	Gateway name
label	Gateway label
reason	
setItemValueCommand	Script for setting items value of this gateway
unreachableReasons	
unreachableActions	

Plugin initialization

For correct Initialization of plugin startup script should be created. There should be placed all initialization steps. In our example we will subscribe on ZWave addon events to receive them

in our plugin.

startup.lua

```
local zwave = require "zwave"
zwave.subscribe( "HUB:my_first_plugin/scripts/events/events_handling" )
```

If you want to use any script from you plugin, it's path should be in format "*HUB:<plugin_name>/<path_to_file>*"

To register this script as a startup script, it should be added to plugin's "*config.json*" file.

config.json

```
...
"startup": "scripts/startup",
...
```

Events handling

During the plugin initialization process we registered a handling file. Now it's time to implement it.

events_handling.lua

```
local params = ...

local handler = loadfile( "HUB:my_first_plugin/scripts/events/" .. params.event )

if handler then
    handler( params )
else
    print( "Unsupported event " .. params.event )
end
```

This code tries to find a handler for the received event. If a file with an event name exists in the "events" folder, then it will be executed.

Device adding

Now its time to create our first device. It will be a simple switch based on Binary Switch command class.

After the device interview process finished, ZWave addon sends a "**node_added**" event. All ZWave events and their structure you can see in API documentation.

First of all we should get full node information from ZWave addon. To do that we executed the "**zwave.get_node**" method. All ZWave methods you can see in API documentation.

node_added.lua

```
local zwave = require "zwave"
local zwave_node = zwave.get_node( params.node_id )
...
```

Now we can iterate through channels and command classes to find Binary Switch command class.

node_added.lua

```
...
for _, channel in ipairs( zwave_node.channels ) do
    for _, cc in ipairs( channel.classes ) do
        if cc.id == 0x25 then -- Binary Switch command class ID
            -- generate device and item
        end
    end
end
...
```

When we found a needed command class, let's generate a device. We are using the "**core.add_device**" method. All Core methods you can see in API documentation.

node_added.lua

```
...
local core = require "core"

local device = {
    name = "MySwitch",
    gateway_id = core.get_gateway()._id,
    category = "switch",
    subcategory = "interior_plugin",
    type = "switch.outlet",
    device_type_id = math.floor( zwave_node.manufacturerId ) .. '_' ..
                     math.floor( zwave_node.productType ) .. '_' ..
                     math.floor( zwave_node.productId ),
    battery_powered = false
}

local device_id = core.add_device( device )
...
```

Then we can create items. We are using the "**core.add_item**" method.

node_added.lua

```
...
local item = {
    device_id = device_id,
    name = item_names.switch,
    value_type = "bool",
    value = false,
    has_getter = true,
    has_setter = true
}

local item_id = core.add_item( item )
...
```

Device and item mappings

For future use of this device, we will need to store the relationship between ZWave node id and device id. For this purpose we are using a "**storage**" module.

node_added.lua

```
...
storage.set_string( zwave_node._id, device_id )
storage.set_string( device_id, zwave_node._id )
...
```

Same thing is required for an item. To identify item id we will need a composite key, containing "**item_name**", "**node_id**", "**channel_id**", "**class_id**". "**storage**" module can receive only string as a key, so we should generate a string for that key.

item_descriptor_utils.lua

```
local utils = {}

function utils.to_string( descriptor )
    return descriptor.item_name .. '_' ..
           math.floor( descriptor.node_id ) .. '_' ..
           math.floor( descriptor.class_id ) .. '_' ..
           math.floor( descriptor.channel_id )
end
return utils
```

Now, using this utility we can create mappings for items.

item_descriptor_utils.lua

```
...
storage.set_string(
    loadfile( "HUB:my_first_plugin/scripts/utils/item_descriptor_utils" )().to_string(
item_descriptor ),
    item_id
)
storage.set_table( item_id, item_descriptor )
...
```

Full device generation process

Item_descriptor_utils.lua

```
local params = ...

local zwave = require "zwave"
local core = require "core"
local storage = require "storage"
```

```
local zwave_node = zwave.get_node( params.node_id )
for _, channel in ipairs( zwave_node.channels ) do
   for _, cc in ipairs( channel.classes ) do
      if cc.id == 0x25 then -- Binary Switch command class ID
         -- add device
         local device = {
            name = "MySwitch",
            gateway_id = core.get_gateway()._id,
            category = "switch",
            subcategory = "interior_plugin",
            type = "switch.outlet",
            device_type_id = math.floor( zwave_node.manufacturerId ) .. '_' ..
                        math.floor( zwave_node.productType ) .. '_' ..
                        math.floor( zwave_node.productId ),
            battery_powered = false
         }
         local device_id = core.add_device( device )
         storage.set_string( zwaveitem_descriptor_utils.lua_node._id, device_id )
         storage.set_string( device_id, zwave_node._id )

         -- add item
         local item = {
            device_id = device_id,
            name = "switch",
            value_type = "bool",
            value = false,
            has_getter = true,
            has_setter = true
         }
         local item_id = core.add_item( item )
         local item_descriptor = {
            item_name = "switch",
            node_id = zwave_node._id,
            class_id = cc.id,
            channel_id = channel.id
         }
         storage.set_string(
            loadfile( "HUB:my_first_plugin/scripts/utils/item_descriptor_utils" )().to_string(
item_descriptor ),
            item_id
         )
         storage.set_table( item_id, item_descriptor )

         return
      end
   end
end
```

Start device adding

We don't create any scripts to init device adding. We are using ZWave protocol, which default plugin already has a "start_include" script. To init this process, should be executed:

```
{
    "method": "extensions.plugin.run",
    "id": "_ID_",
    "params": {
        "script": "HUB:zwave/scripts/start_include"
    }
}
```

Device removing

Another major thing is to remove devices and items, when the device left ZWave net.

In ZWave protocol there are two triggers for that. First is when the device excluded the net, and second when the whole ZWave net was reseted.

Single device removing

To handle the first trigger, we should add a handler for the "**node_removed**" event.

node_removed.lua

```
local params = ...

local core = require "core"
local storage = require "storage"

local node_id_string = tostring( math.floor( params.node_id ) )

local device_id = storage.get_string( node_id_string )
if device_id then
    local items = core.get_items_by_device_id( device_id )
    -- remove items mappings
    for _, item in ipairs( items ) do
        local descriptor = storage.get_table( item._id )
        storage.delete( item._id )
        storage.delete( loadfile( "HUB:my_first_plugin/scripts/utils/item_descriptor_utils"
)().to_string( descriptor ) )
    end

    -- remove device mappings
    storage.delete( node_id_string )
    storage.delete( device_id )
```

```
    -- remove device with items
    core.remove_device( device_id )
end
```

In this code first we remove items and device mappings using "**storage.delete**" method. And after that we delete devices from core using "**core.remove_device**" method. All device items will be deleted automatically.

ZWave net reset

To handle the second trigger, we should add a handler for the "**module_reset**" event.

module_reset.lua

```
local params = ...

local core = require "core"
local storage = require "storage"

if params.status == "finished" then
    -- clear all databases
    core.remove_gateway_devices( core.get_gateway()._id )
    storage.delete_all()
end
```

In this code we are using "**core.remove_gateway_devices**" to delete all devices that were created from the current plugin. To clear all bindings, we are using "**storage.delete_all**" method.

Item value update

Now we have created our first device and item. It's time to add logic for viewing actual device state. To receive a value update for the Binary Switch command class we will have to handle the "**value_updated**" event.

value_updated.lua

```
local params = ...

local core = require "core"
local storage = require "storage"

if params.class_id == 0x25 then -- Binary Switch command class ID
   local item_descriptor = {
      item_name = "switch",
      node_id = params.node_id,
      class_id = params.class_id,
      channel_id = params.channel_id
   }

   local item_id = storage.get_string( loadfile("HUB:my_first_plugin/scripts/utils/item_descriptor_utils"
)().to_string( item_descriptor ) )
```

```
    if item_id then
        core.update_item_value( item_id, params.value ~= 0 )
    end
end
```

In this code we are searching "item_id" using previously created bindings, and if it is found, updating its value using "core.update_item_value" method.

Device controlling

The last thing left, is to have the opportunity to control devices from the plugin side. We have to create a script "set_item_value" and register it in "confi*g.json*" file.

config.json

```
 ...
 "setItemValueCommand": "HUB:my_first_plugin/scripts/set_item_value"
 ...
```

set_item_value.lua

```
 local params = ...

 local zwave = require "zwave"
 local storage = require "storage"

 local item_descriptor = storage.get_table( params.item_id )
 if item_descriptor then
     if item_descriptor.class_id == 0x25 then -- Binary Switch command class ID
         zwave.set_value( item_descriptor.node_id, item_descriptor.class_id,
 item_descriptor.channel_id, params.value and 255 or 0 )
         zwave.request_value( item_descriptor.node_id, item_descriptor.class_id,
 item_descriptor.channel_id )
     end
 end
```

In this code we are checking that we are trying to control the Binary Switch command class and then set a new value using the "**zwave.set_value**" method. To receive updated device state and update item value we request new value using "**zwave.request_value**" method.

Global state saving

Small remark about global states. If you were working with plugins on Vera controllers, you might know that to save any global state between scripts execution, you may simply save those data into a global variable and it will be available on every lua execution. In Ezlo firmware it was changed. Now, to save any global data you should use "**storage**" module functionality. In the API documentation described all methods available in this module. Here are some examples.

Save simple data

If you want to save *bool*, *number* or *string* data you can use corresponding methods "**set_bool**", "**set_number**" and "**set_string**".

example.lua

```
local data = "some_data"

local storage = require "storage"
storage.set_string( "data_key", data )
```

Save table data

If you need to save complicated table data, you can use the "**set_table**" method.

example.lua

```
local data = {
    key1 = { value1, value2 },
    ...
}

local storage = require "storage"
storage.set_table( "data_key", data )
```

Get saved data

Any time, when you need those global data, you can use "**get_...**" methods. Here is an example for string data.

example.lua

```
local storage = require "storage"
local data = storage.get_string( "data_key" )
```

And example for table data.

example.lua

```
local storage = require "storage"
local data = storage.get_table( "data_key" )
```

If no data with such key available, "**get_...**" methods will return **null**.

Remove data

To remove any saved data you should use the "**delete**" method.

example.lua

```
local storage = require "storage"
local data = storage.delete( "data_key" )
```