mios

HOW TO CREATE A PLUGIN

Steps to Create and Deploy your Plugin

A - Create your Plugin Code

Step 1. Choose your plugin type

Step 2. Create your code files.

Step 3. Create a tar.gz package from your code files.

B - Upload and Install your Plugin

Step 1. Upload your plugin package to the EZLogic web platform

Step 2. Install your package to a controller directly from EZLogic platform.

C - Use your Plugin in EZlogic

Step 1: Login to EZLogic and select your plugin devices as a node in a meshbot trigger.

You can also create a tile for your device in your dashboard.

Step 2. Interact with your devices in a meshbots and in your dashboards.

General help to use the EZlogic interface is available here.

Types Of Plugins

MiOS has 3 types of plugin.

1 - Gateway Plugin:

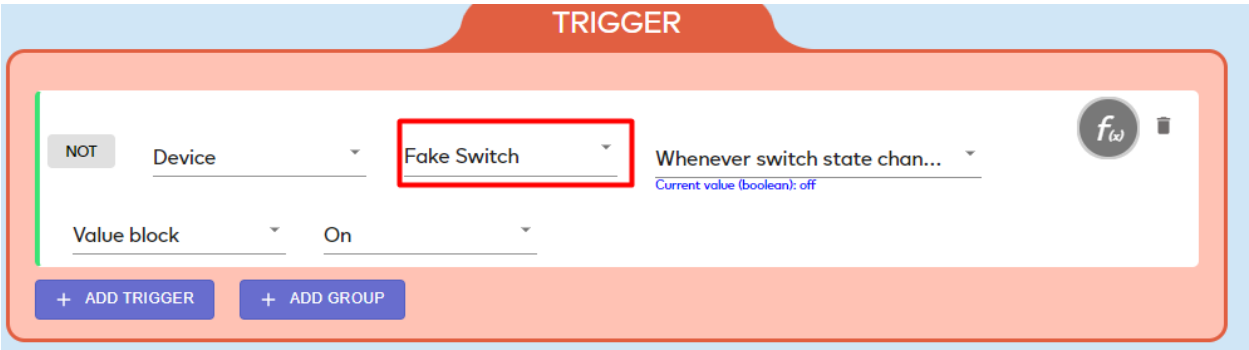
Plugins that can create logical (virtual) devices, items and settings.

config.json example:

```
{
  "id": "example",
  "version": "1.0",
  "type": "gateway",
```

<snip - rest of config.json file >

Once you have created a gateway plugin, you can select the devices created by the plugin as a (logical) device in a meshbot trigger. This screenshot shows a target device called ‘Fake Switch’ which was created by plugin:



Logical / Virtual devices - You can create any type of virtual/logical device via gateway plugins. Virtual devices work just like physical devices and can save you time via automation. They can also act as a ‘universal translator’ for unsupported devices. You can read more about the advantages of logical / virtual devices here.

Items (aka ‘capabilities’) and settings

- **Item** is the word used throughout our API to describe a capability of a device. For example ‘Thermostat Fan State’ is a capability (item) of a thermostat device. ‘Start Recording’ and ‘Stop Recording’ are capabilities (items) of a camera device.
  Items can be set or just read (get). Capabilities which can be set are exposed as ‘Actions’ in meshbots. Capabilities which can be read are exposed as ‘Triggers’ in meshbots.

- **Setting**
  Settings are configuration values for device capabilities/items. With gateway plugins, you can create custom settings for your logical devices.

You can read more about devices, settings and items in the Concepts and Terminology section.

2 - Library Plugin:

Plugins that are used as a dependency by other plugins. Library plugins are usually called as a module by another plugin script. Here’s an example of Library plugin usage with a library called ‘example’:

```
local lib = require( "example" )
lib.print( "LIB" )
lib.hello()
```

config.json example:

```
{
  "id": "example",
  "version": "1.0",
  "type": "library",
  "permissions": [],
  "dependencies": {
    "firmware": "1.0",
    "addons": [
      {
        "id": "lua",
        "version": "1.0"
      }
    ]
  },
  "library": {
    "entryPoint": "HUB:example/scripts/main",
  }
}
```

In the example above, the `entrypoint` script is run once the first time the library is required:

```
local result = {}
result.print = function(...)
     print(...)
end
result.hello = function()
     print( "hello" )
end
result.add = function(a,b)
     return a + b
end
result.print_time = function(a,b)
     local core = require "core"
     print( core.get_local_time() )
end
return result
```

3 - Extension Plugin:

These plugins add functionality to an existing plugin without modifying the original plugin. Extensions can work with existing devices but cannot create their own devices. Extension plugins are not yet ready.

Lua is the language used for MiOS plugins.

This document explains how to create a 'Gateway' plugin. You must use `gateway` as the type in your config.json file.

Before we look at creating a gateway plugin, let's first look at their structure.

Plugin Structure

Developers should use the plugin structure outlined below as their starting point. A minimal plugin structure is as follows:

File quantity and names:
MiOS expects a minimum of 4 files with these exact names as part of a 'Gateway Plugin':

File 1: config.json
File 2: interface.json
File 3: startup.lua
File 4: teardown.lua

Format:
Place all files in a .tar.gz archive - <Myplugin_Name>.tar.gz.

- We have no restriction on the contents, number of folders or folder structure inside the .tar.gz file. You can place script files in folders however you please. The only rules are that the config.json and interface.json files are on the root, and all paths mentioned in these two files are valid.

Structure:
Place and name your files as follows:

File 1: config.json
- Must be in the top level directory
- Name cannot be changed

File 2: interface.json
- Must be in the top level directory
- Name cannot be changed

File 3: startup.lua
- This file can be in any directory
- Name can be changed

File 4: teardown.lua
- This file can be in any directory
- Name can be changed

What do they do:

File 1: config.json

This file is where you define configuration attributes, dependencies and metadata:

- Name
- Version
- Type
- Metadata (User friendly name, description, author)
- Dependencies
- References
- Execution Policy
- Entry and Exit points

File 2: interface.json

Define user input fields for configuration. Define user input fields for data source interaction

File 3: startup.lua

A script which is executed every time the firmware is started or rebooted after the plugin is installed.

File 4: teardown.lua

teardown.lua is a script that is executed when the plugin is uninstalled. It contains cleanup logic to remove devices, temporary files, timers etc created by the plugin on the hub. This saves resources on the hub.

See our API docs https://developer.mios.com/api/hub/user-functionality/custom-scripts/ for full details.

Our documentation contains examples and explanations for all APIs and modules that we make available for use in plugins. For example, under ‘Scripting > Modules’ you can find the timer module: https://developer.mios.com/api/scripting/modules/timer/timer-module-description/

How to Create a config.json file

Let's first take a look at what a skeleton default config.json looks like:

Example config.json file

```
{
    "id": "test_plugin",
    "version": "1.0",
    "meta": {
        "name": { "text": "Test plugin"  },
        "description": {"text": "This plugin is Test" },
        "author": {"text": "Ezlo Plugin Team" },
        "type": "node",
        "language": "lua"
     },
     "permissions": [
        "core",
        "logging"
      ],
     "startup": "startup",
     "teardown": "teardown",
     "gateway": {
        "name": "Test plugin",
        "label": "Test plugin",
     }
}
```

The example above is the minimum code required for a config.json file. You cannot change the name of this file - it must be named 'config,json'.

Now let's take a look at a real life example of a config.json:

```
{

    "id": "node_test_plugin",
    "version": "1.1",
    "meta": {
        "name": {
            "text": "Node test plugin"
```

```
        },
        "description": {
            "text": "This plugin will create a fake switch with an
item upon installation."
        },
        "author": {
            "text": "Ezlo Plugin team"
        },
        "type": "node",
        "language": "lua",
        "placement": {
            "static": true,
            "custom": true
         }
    },
    "type": "gateway",
    "dependencies": {
        "firmware": "2.0",
        "addons": [
            {
                "id": "lua",
                "version": "1.0"
            }
        ]
    },
    "permissions": [
        "core",
        "http",
        "json",
        "logging",
        "storage",
        "timer"
    ],
    "executionPolicy": "restoreLastScriptState",
    "startup": "scripts/startup",
    "teardown": "scripts/teardown",
    "gateway": {
        "name": "Node test plugin",
```

```
        "label": "Node test plugin",
        "forceRemoveDeviceCommand":
"HUB:node_test_plugin/scripts/delete_device",
        "setItemValueCommand":
"HUB:node_test_plugin/scripts/set_item_value",
        "setItemValueResponsePolicy": "auto"
    }
}
```

- See our API docs https://developer.mios.com/api/hub/user-functionality/custom-scripts/plugin-structure/ for more information on config.json.

id - The name of your plugin as it will be known in the file-system.

- This id should be unique among your plugins on a specific hub.
- The plugin install folder on the firmware and the tar.gz archive name should have the same name as the config.json “id” value.
- Your custom plugin is referenced in the API by the name you assign to it in this “id” line. For example, ‘node_test_plugin’ is the name of the plugin defined here:

```
{
    "id": "node_test_plugin",
    "version": "1.1",
```

<rest of config.json…>

This identifier is used in the path when calling scripts and other API components. For example,

```
 constants =
require(HUB:node_test_plugin/scripts/definitions/constants”),
```

Meta - Object which contains public-facing/general information about the plugin. This object is described in our API documents here.

- “meta” > “name”/”description”/”author” - public-facing information about the plugin.
- “meta” > “type” - this should always be “node”.

- “meta” > “language” - must be set to “lua”.

- “meta” > “placement” - *“static”: true* and *“custom”: true* are required. The plugin will fail if these are omitted or are specified as a different type.

Type - The type of plugin as explained here. This should be “`gateway`” to create a data source node plugin of the type described in this document.

Dependencies

Lists the minimum firmware version and versions of addons that are required for your plugin to run. Example:

```
"dependencies": {
        "firmware": "2.0.23.1818",
        "addons": [
            {
                "id": "lua",
                "version": "1.0"
```

Permissions

The modules that you want to include in the plugin. Users will have to agree to let the plugin use those permissions when they install it.

- For example, a plugin which requires http requests will need to specify the HTTP module and its events as listed here.

- See a full ‘List of Lua modules’ here.

Execution Policy

Developers can specify that a plugin saves a global state and restores it when the plugin is re-started.

Gateway

Configure the name, label and commands for the plugin. ‘Gateway’ is the entity created to orchestrate all the devices and items created by the plugin.

All commands and fields you can add to the “`gateway`” section are listed at https://developer.mios.com/api/hub/plugins/api/gateway/.

Some are required and others optional. You should include all ‘required’ commands and any ‘optional’ commands that are needed by your plugin. The following section explains all required fields/commands and a sample of optional commands:

- **Name** [Required] - the internal id of the plugin. This name identifies the plugin in the list of hub plugins at hub.gateways.list, allowing the system to call and reference it. The gateways list contains references to generic scripts about each plugin. These include gateway value set commands, dictionary value set commands, ready status, unreachable actions and so forth.

- **Label** [Required] - the public-facing name of the plugin for end-users.

- **forceRemoveDeviceCommand** [Required] - command to call a script to uninstall a device. Example:

```
"forceRemoveDeviceCommand": "HUB:node_test_plugin/scripts/delete_device",
```

In the example above, 'delete_device' calls 'delete_device.lua', which uses the following command to remove a device:

hub.device.force_remove

- **setItemValueCommand** [Required] - command to call a script which defines a device capability (aka 'item'). For example, 'Take a snapshot' on a camera device. Example:

```
"setItemValueCommand": "HUB:node_test_plugin/scripts/set_item_value",
```

In the example above, 'set_item_value' calls 'set_item_value.lua' which uses one of the following commands to specify a device capability (item):

hub.item.value.set (single item)
hut.item.value.set (multiple items, one value)
hub.item.value.set (multiple items, different values)

- **setSettingValueCommand** [Optional] - command to call a script to modify the value of a device capability (aka 'item'). Example:

```
"setSettingValueCommand": "HUB:node_test/scripts/set_setting_value"
```

In the example above, 'set_setting_value' calls a script named 'set_setting_value.lua' which uses the following command to specify a capability (item) value:

hub.device.setting.value.set

- **setItemValueResponsePolicy** [Optional] - specifies the response type for requests sent by `hub.item.value.set` (single or multiple version). The response policy field applies to Linux firmware only. Example:

```
"setItemValueResponsePolicy": "auto"
```

Possible values:

"auto" - The firmware sends the response immediately after receiving the hub.item.value.set request. This is the default setting.

"custom" - The plugin is responsible for sending the response to the hub.item.value.set request. It must call `core.send_response()` to do this. You must specify an additional parameter, **"operation id"**, in the script you call in setSettingValueCommand. The firmware will send a timeout error if the plugin was unable to send a response within 2 minutes.

More info about gateway is available in the 'Concepts and Terminology' section here.

Startup - startup.lua.

Specify a path to a script that is called every time the firmware is started or rebooted. In our example, 'startup.lua' is loaded next after config.json and can call other scripts from within it. For example, you can call 'constants.lua' with:

```
local _constants = require("HUB:node_test_plugin/configs/constants")
```

Example startup.lua:

```
local _logger = require("logging")

_logger.info("node_test_plugin starting up...")

loadfile("HUB:node_test_plugin/scripts/functions/create_device")()

local _constants = require("HUB:node_test_plugin/configs/constants")

_G.constants = _constants or {}
```

- Users must first login to their account before we can begin adding devices to the plugin. This is because we need to know how many devices they already have on the plugin.

- Once access rights have been verified, the script will call the device creation script with the following command:

  ```
  loadfile("HUB:node_test_plugin/scripts/functions/create_device"")()
  ```

- This loads 'create_device.lua' which contains a call to core.add.device in the core module.

Create Device - create_device.lua

This script lets you call an API function to create a device on your plugin. You can also use it to specify device type, category, id, battery requirements etc.

Note - 'create_device.lua' is just our name for the script in our example. You can name it however you please.

Example with test parameters:

```
local _core = require("core")
local _logger = require("logging")
local _storage = require("storage")

local credentials = _storage.get_table(_G.constants.STORAGE_ACCOUNT_KEY)

if not credentials then
    _logger.warning("No account is configured... The user did not log in yet.")
end

local function CreateDevice ()
    local my_gateway_id = (_core.get_gateway() or {}).id
    if not my_gateway_id then
        return nil
    end

    local count = 0
    for _, device in pairs(_core.get_devices() or {}) do
        if device.gateway_id == my_gateway_id then
            count = count + 1
            if not credentials or count >=2 then
                return device.id
            end
```

```
        end
    end

    _logger.info("Create new fake device")
    return _core.add_device {
        type = "switch.inwall",
        device_type_id = "switch.inwall.fake",
        category = "switch",
        subcategory = "interior_plugin",
        battery_powered = false,
        gateway_id = _core.get_gateway().id,
        name = not credentials and "Fake Switch (install)" or "Fake
Switch (login)",
        info = {
            manufacturer = "Ezlo",
            model = "1.0"
        }
    }
end

local function CreateItem (device_id)
    if not device_id then
        _logger.error("Cannot create item. Missing device_id...")
        return
    end
    for _, item in ipairs(_core.get_items_by_device_id(device_id) or
{}) do
        return
    end

    _logger.info("Create new fake 'switch' item")
    _core.add_item({
        device_id = device_id,
        name = "switch",
        value_type = "bool",
        value = false,
        has_getter = true,
        has_setter = true
```

```
    })
end

local device_id = CreateDevice()

CreateItem(device_id)
```

Teardown - teardown.lua. This is called once when the plugin is uninstalled. It contains cleanup logic to remove devices, temporary files, timers etc created by the plugin on the hub. This saves resources on the hub.

How to create an interface.json file (coming soon)

interface.jason defines a set of inputs that are displayed on the UI. It handles the configuration of the plugin after installation. For example, if the plugin needs credentials to work, then this requirement is listed in interface.json.

The `configuration` section lists all elements required to configure the plugin. You can call other Lua config scripts that you have created from here. For example

“script”: “HUB:node_test_plugin/configs/account”,

The `account` part calls ‘account.lua’ and requests the inputs as shown below. accounts.lua collects the actual UN/PW from the account. In this case, the username and password are passed and used to populate the ‘local args’ part of account.lua.

Example interface.json file:

```
"configuration": {
        "type": "static",
        "script": "HUB:node_test_plugin/configs/account",
        "placement": "device,plugin",
        "inputs": [
            {
                "name": "Username",
                "description": "Fake Account Username",
                "required": true,
                "field": "username",
                "type": "string"
            },
            {
```

```
                "name": "Password",
                "description": "Fake Account Password",
                "required": true,
                "field": "password",
                "type": "string"
            }
        ]
    }
}
```

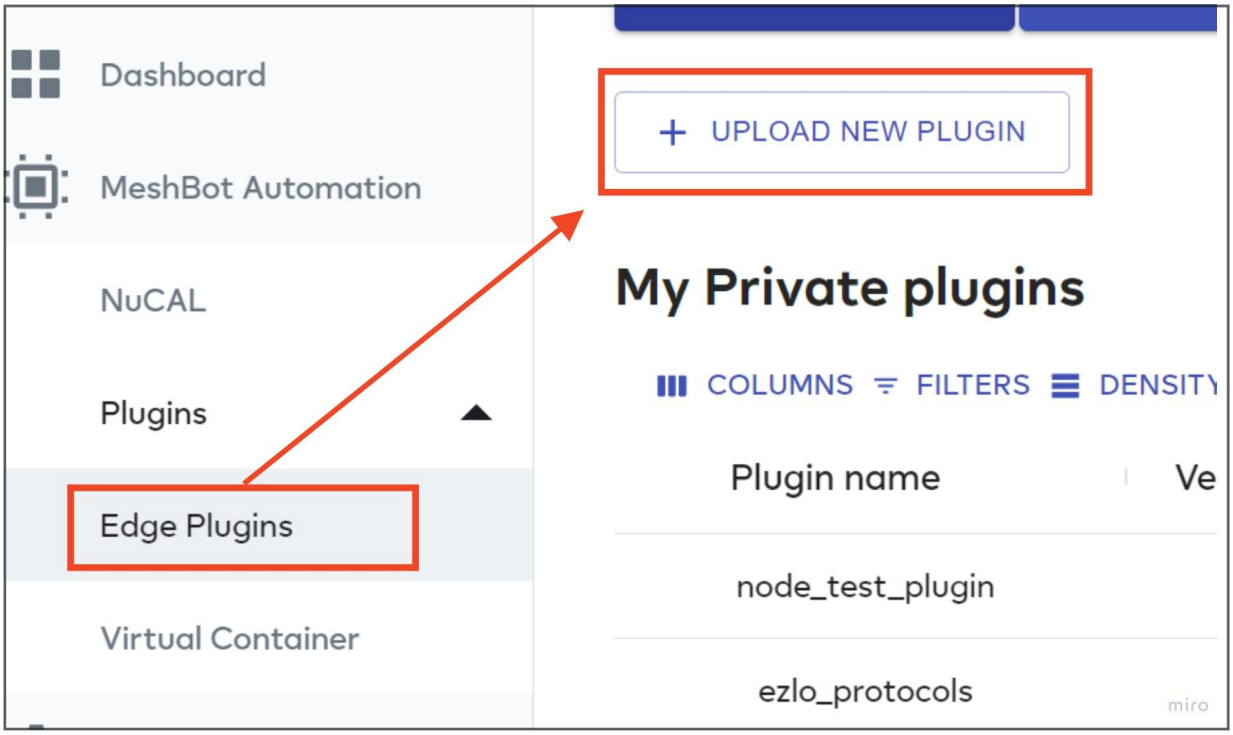
- interface.json must be included in your .tar.gz for your custom plugin to operate with the new UI.
- Type must be “static”. It will fail if you use “none”.
- "script": "HUB:node_test_plugin/configs/account"

Upload, install and manage Plugins

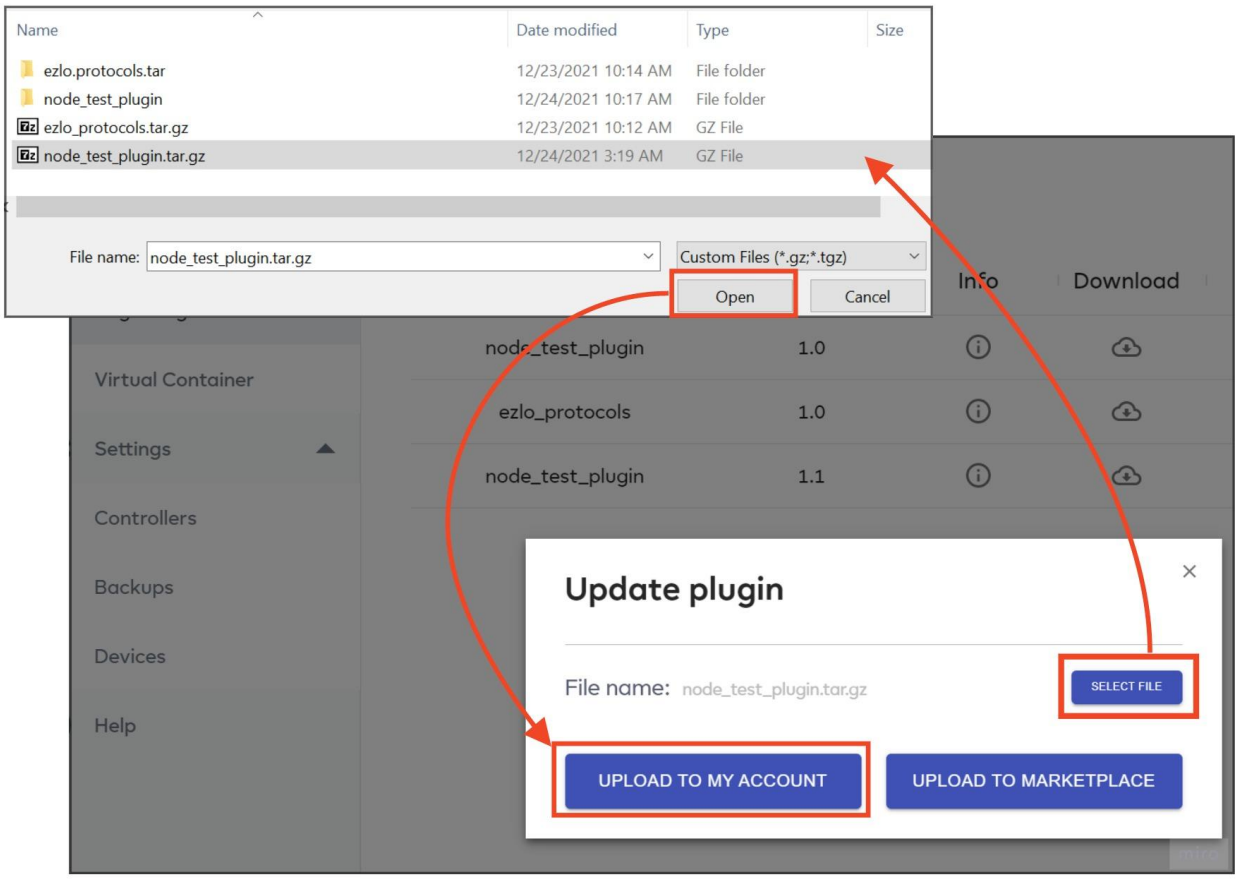
Upload Your Plugin

You can upload your plugin to EZlogic as follows:

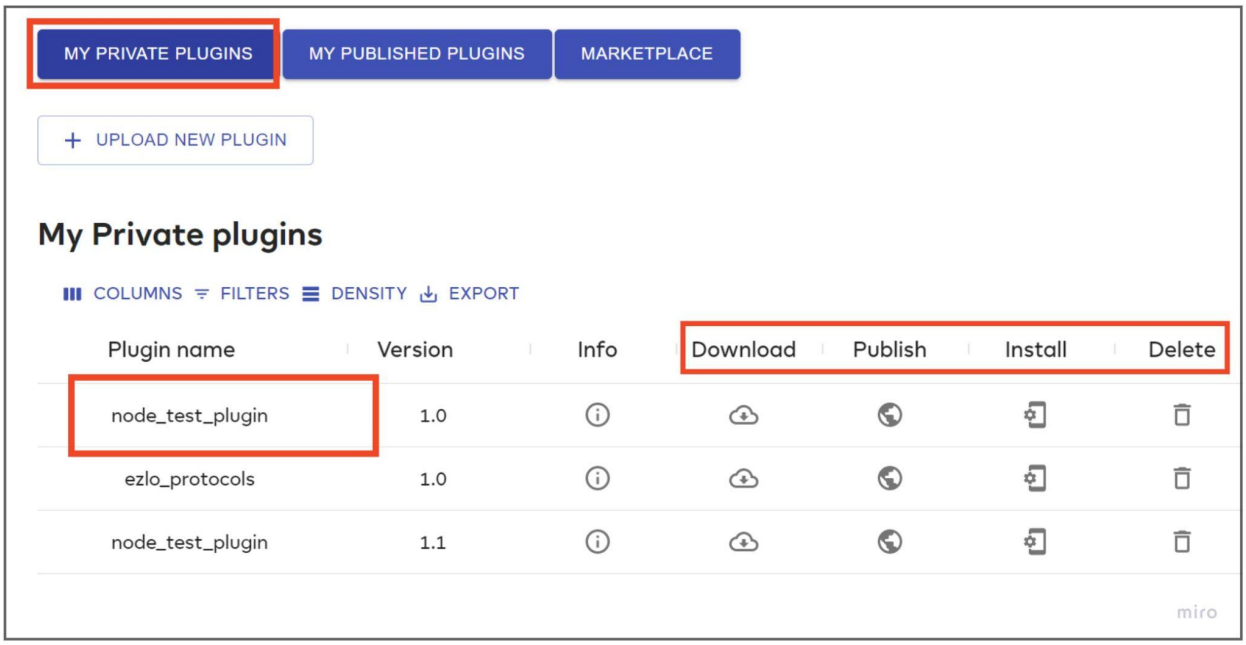
- Login to EZlogic at https://ezlogic.mios.com.
- Click ‘**Plugins**’ > ‘**Edge Plugins**’ in the left-hand menu.
- Make sure you are in the ‘**My Private Plugins**’ area
- Click the ‘**Upload New Plugin**’ button.



- Click '**Select File**' and choose your plugin package (tar.gz)
- Click '**Upload to my account**':

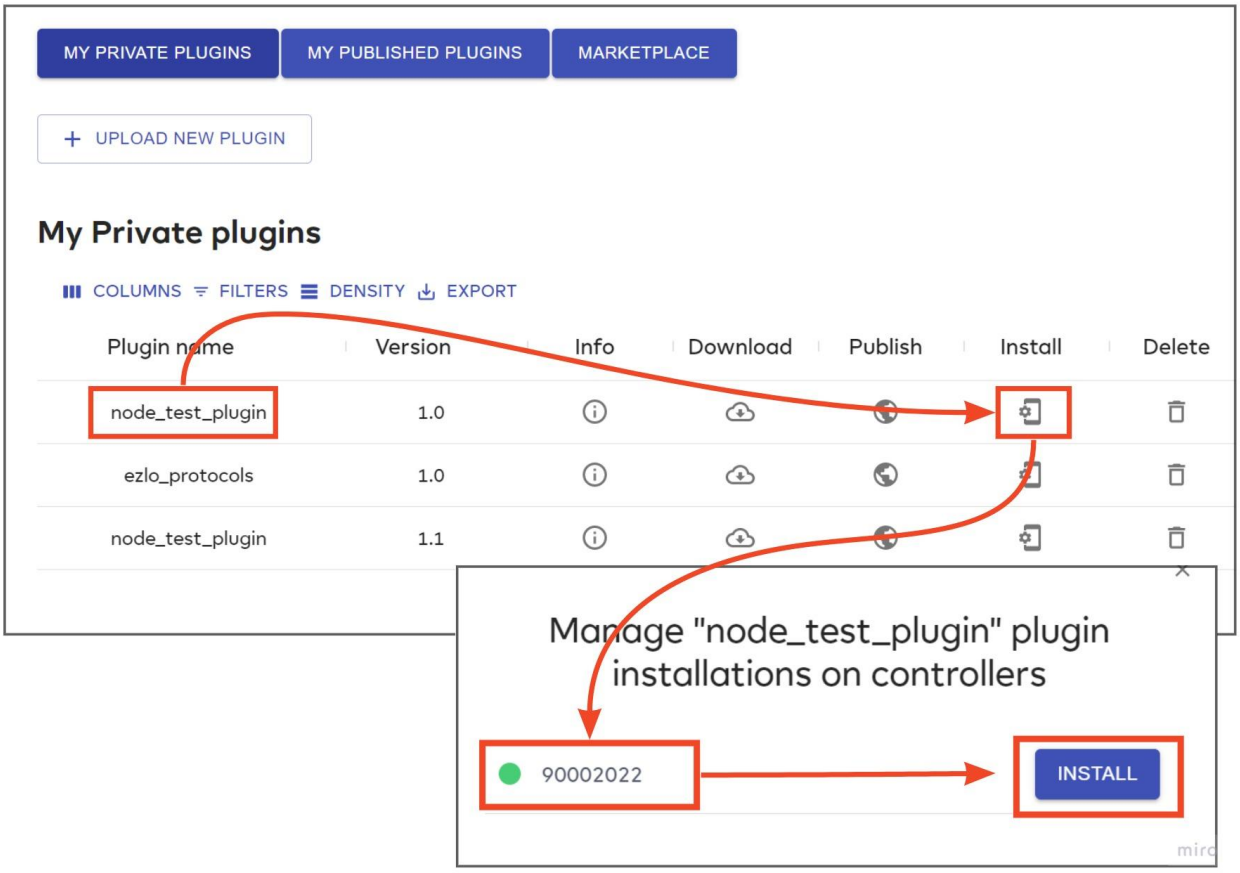


- This adds the plugin to the '**My Private Plugins**' area.
- Use the 'Download', '[Publish](#)', '[Install](#)' and 'Delete' controls to manage your plugin:



Install Your Plugin on a Controller

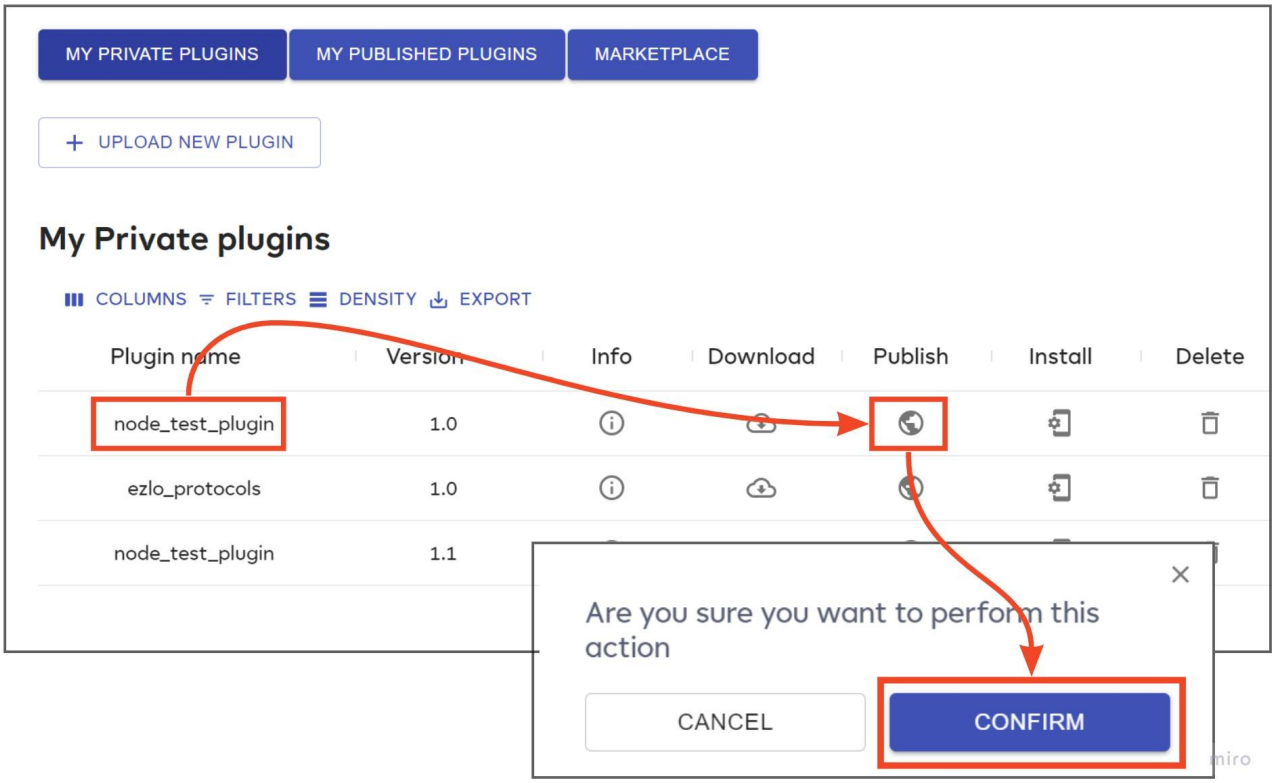
- Upload your plugin to '**My Private Plugins**' as explained above.
- Click the '**Install**' icon in the row of the plugin you want to deploy.
- Select the controller on which you want to install your plugin.
- Click '**Install**' to install it on your controller:



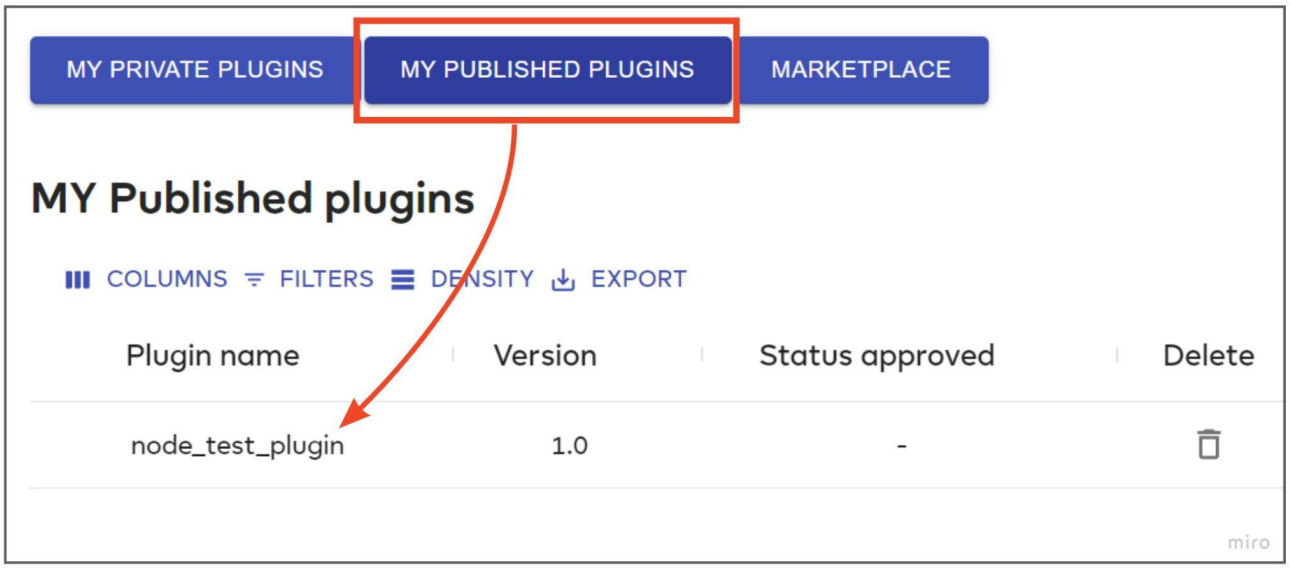
- Check to see if your plugin is installed by creating a test meshbot. Select ‘Device’ as your node type. All devices created by your plugin should then be visible in the ‘Node’ drop-down. See the trigger interface for more help with meshbot triggers.

Share Your Plugin on the Marketplace

- Upload your plugin to ‘**My Private Plugins**’ as explained above.
- Click the ‘**My Private Plugins**’ button.
- Click the ‘**Publish**’ icon in the row of the plugin you want to upload.
- Click ‘Confirm’ to publish your plugin to the marketplace:



- Click '**My Published Plugins**' to verify the upload and manage the plugin going forward:



Use public plugins from the Marketplace

- Click **‘Plugins’ > ‘Edge Plugins’** in the left-hand menu.
- Click the ‘**Marketplace**’ button to view and install publicly available plugins
- **Certified** - Plugins that have been tested and approved by the MiOS team are listed as ‘Ezlo’ plugins. Plugins that have been uploaded by the community but have not been tested by MiOS are listed as ‘Community’ plugins.
- **Copy to my Ezlo** - Copy the plugin to ‘My Private Plugins’. From there you can download, inspect and/or modify the plugin before installation.
- **Install to my Elzo** - Directly install the plugin to a controller of your choice.

Known Issues

- interface.json > configuration > *type* = *“none”* will fail. You must use “static” as the type.
- config.json > meta > language > placement. *“static”: true* and *“custom”: true* are required. The plugin will fail if these are omitted or are specified as a different type.

Concepts and terminology

Gateway - The entity created to orchestrate all the devices and items created by the plugin.

- 'Gateway' is an object that is bound and created when you specify "`type`": "`gateway`" in config.json. You must use 'gateway' as the type in a node plugin.

  To clarify, in config.json there are two "type" fields. You should specify them as follows:

  "meta" > "type" = "node"

  "type" = "gateway"

- You can check all gateways on a controller with the following websocket request - {"id": "_AUTO_74107","method": "hub.gateways.list","params": {}} using our API tool at https://apitool.ezlo.com/dashboard . This tool lets you simulate the requests that are made by our mobile and web apps so you can test and debug your plugins.

- The UI will be notified via a hub.gateway.added broadcast when you install a plugin

Device - a physical or virtual (logical) device, a service, or a component which is included in the plugin.

- Each gateway can have zero or more devices.

- Devices can have a parent-child relationship. For example, a humidity sensor can be a child of a thermostat. Once you remove the parent device, all child devices are also removed.

Item - a device capability. For example, 'Start Recording' and 'Stop Recording' are capabilities (items) of a camera device.

- Each device can have zero or more items (capabilities).

- A list of our defined items is available in the API docs here. The 'Enum' column links to allowed values/settings for the item.

- Use the defined items in your plugin if you want the item to be visible in the UI. Plugin items not shown on this list will not be visible in the UI.

Setting - Settings are configuration values for device capabilities/items. With gateway plugins, you can create custom settings for your logical devices.

- Each device can have zero or more settings. Each setting can change the behavior of the device.

- Device settings can also be used to add new settings that are managed solely by the plugin. For example, to let users change the polling frequency of a device's status. As another example, a setting to change the color saturation of a camera can be forwarded to the physical device via an API call, just as you would do with an item call.

- You can view available settings for each of your devices in the 'Devices' section of the EZlogic portal. Click 'Function' on a device to see the settings list.

- With a few exceptions, our mobile apps do not currently show device settings in the UI.

- The web UI will be notified with a hub.device.setting.added broadcast when you add/remove/modify a device setting from a plugin.

- The settings object is documented in the core objects section of our API docs here.

- API's to work with device settings from plugin code are listed here.

- API's to work with device settings from the API tool are listed here.