AMX MUSE Programming Guide

Version 1.2

Contents

Revision History

Revision	What's new
1.0	Initial Release

1.1	- Python update to version 3.10
	    - https://docs.python.org/3/whatsnew/3.10.html
	- LDAP integration
	- Platform Service
	    - allows querying for information like FW version, serial number, etc.
	- Auth Session Service
	    - enables scripts to allow basic login credential validation against the current user database
	- I/O Manager Support for CE-COM2 devices for use in Duet Modules
	- ICSLan DNS Lookup support
	    - Allows IP network device name resolution for devices on the ICSLan network
	- Python PIP install supported via *requirements.txt* file
	- Controller to Controller communication
	    - enabled thru a new HControl device type
	- Exposing Scripts as Devices
	    - Allows scripts to expose an API to the MUSE control environment for access and interaction by other scripts or controllers
	- NetLinx Device Driver support
	    - Supports full range of NetLinx control constructs (multiple ports, channels, levels, strings, commands, custom event generation)
	    - Deprecates support for the netlinxClient

1.2	Added details on third-party Python modules support

Firmware & software versions must be matched for features to function. Please update your devices and software applications to these latest versions:

Device/Application	Initial Release	Latest Release
MUSE Firmware	1.1.43	1.2.65
MUSE Automator	1.0.54	1.1.12
VS Code Extension	1.2.16	1.4.7
Minimum Required Firmware		
CE Control Extenders		1.2.7
Varia Touch Panels		1.11.42

Note: Attempting to downgrade from MUSE firmware 1.2.to 1.1.x may have unexpected results. To downgrade, the user must reset the MUSE to factory firmware, and then load 1.1.x firmware.

A unit shipped from the factory with 1.2.x firmware cannot be downgraded to 1.1.x.

Script Versions & Compatibility

MUSE	Python	JavaScript	Groovy	Node-RED
Firmware 1.2.65 VS Code Extension 1.4.7 Automator 1.1.12	3.10	Nashorn	4.0.6	4.0.2

VS Code Extension

AMX has created a lightweight VS Code extension to facilitate the configuration & programming of the MUSE series of controllers. Using the extension, you can:

- Discover MUSE controllers on the network
- Connect to view Devices and Programs
- Write scripts for MUSE
- Upload scripts
- Attach to the standard output of the scripts for debugging

VS Code Extension installation

The MUSE VS Code Extension is available as a standalone .vsix installation file.   To install or update the extension:

1. Launch VS Code from Microsoft:
   https://code.visualstudio.com/download

2. Download the VS Code Extension from AMX.com:
   https://www.amx.com/products/muse-extension-for-vs-code

3. Choose the Extensions tab from the vertical ribbon.
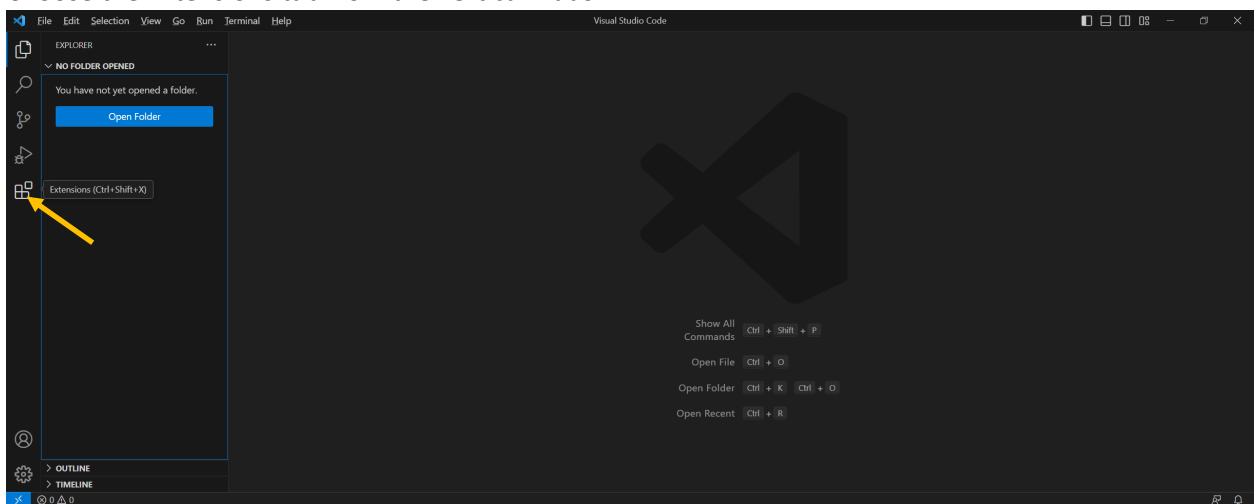
4. Click on the ellipses to show the Extensions context menu and select "Install from VSIX…"
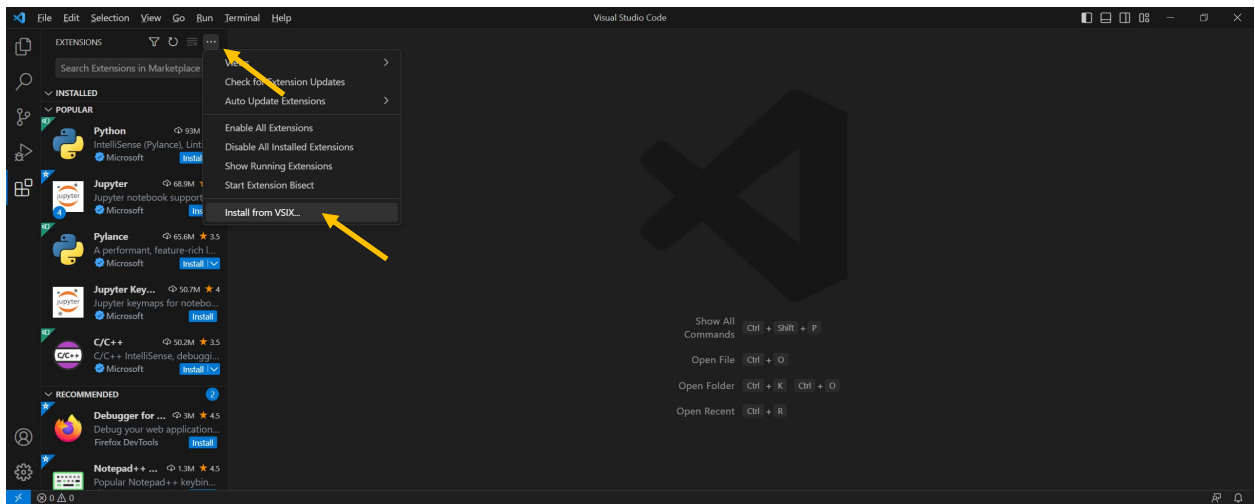
5. Choose the mojo-<version>.vsix file and click Install…

6. Click the Explorer tab on the vertical ribbon.
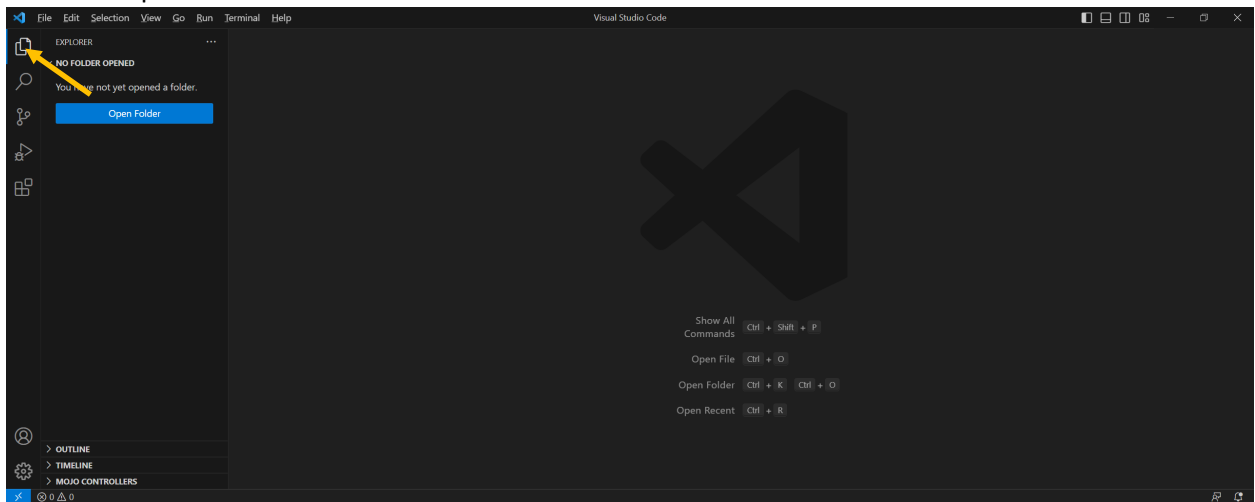
7. If this is the first time you have run the VS Code application, you will be prompted to open a folder.   This folder will contain all the projects you want to group together.   Each project will be in a separate folder and can contain any files your project or script needs.   Choose a folder to create a place for your projects.

Discovering nearby controllers

In the Explorer there is a node labelled MOJO CONTROLLERS. This tree contains all the controllers you have interacted with recently as well as any controllers that have been discovered nearby. The Mojo VS Code extension sends an HControl discovery broadcast message. Any controllers that respond will appear in the list.
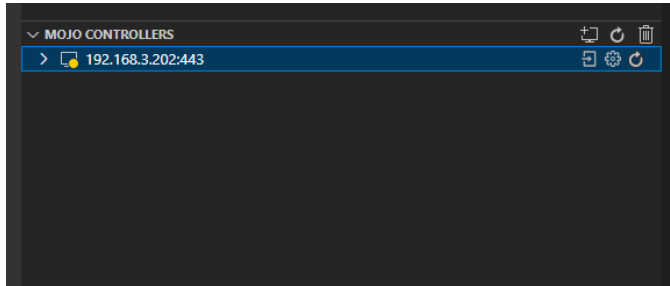
In the above example, a MUSE controller has been discovered at the IP address 192.168.3.202. Note: the yellow pip on the controller icon indicates that it is present but not authorized.

To log in to the controller, click the login icon next to the IP address:
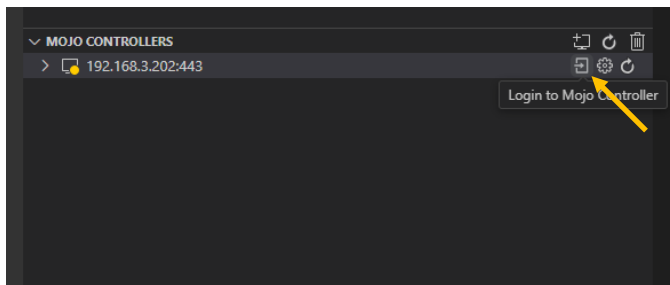
A login prompt will appear at the lower-right portion of the screen:
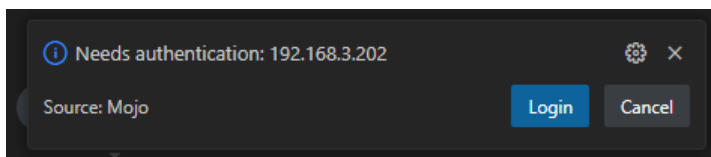
Select 'Login' and follow the prompts at the top of the screen.
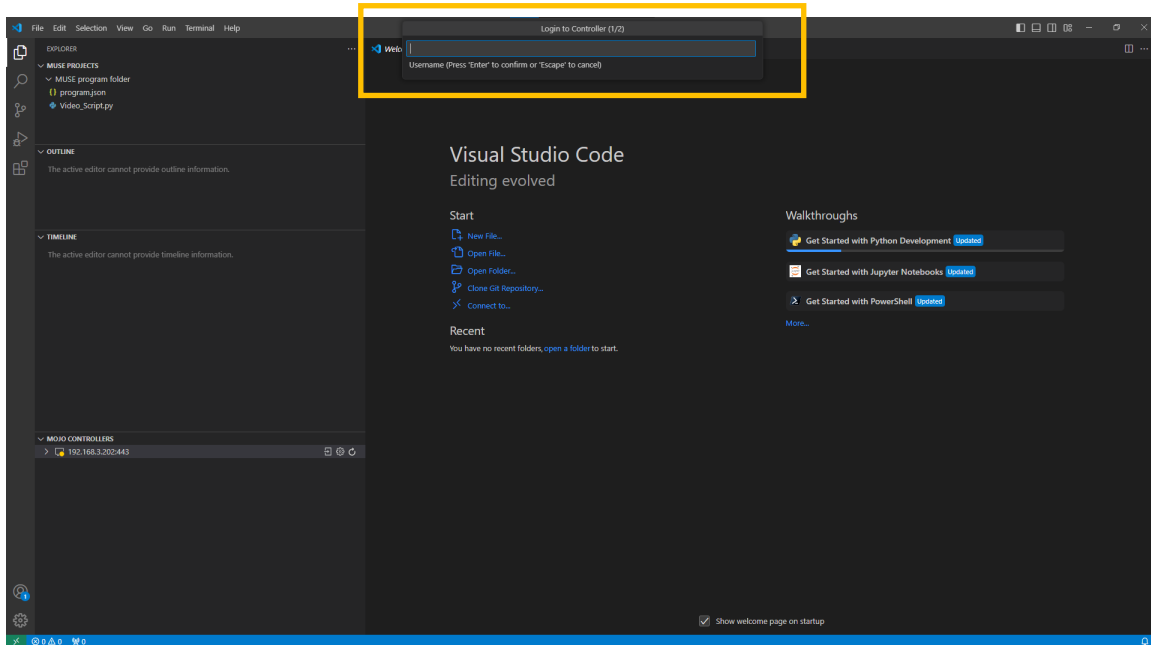
The default credentials for a MUSE controller are **admin | password**

If this is the first time you have logged in to this controller, you must choose a new password. By default, password security is set to 'low', so any password besides 'password' is acceptable.

Click 'Change Password' to continue:
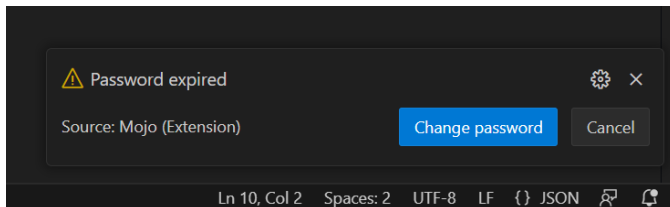
Follow the prompts to change the password and log in.

Mojo Controllers tree view

Once authenticated, the tree view for a Mojo controller contains three nodes:

- Devices – Known AMX devices or modules that the script can access
- Programs - Any scripts loaded to the controller, running or disabled
- HControl – Any HARMAN HControl devices that the MUSE controller has discovered
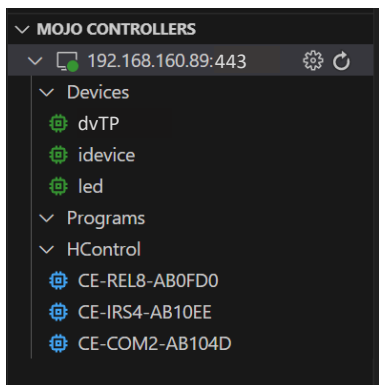
In this example there are three devices; dvTP (a touch panel), idevice (the MUSE controller's built-in control ports), and led (LEDs).

There are no programs currently loaded.

There are three nearby HControl devices. Each of these are the CE family of control extenders.

Note: selecting nearby HControl devices will display their IP address along with other properties.

Creating new scripts

1. To create a new project to contain a script for MUSE to run, right-click on the empty space in the pane and "Create Mojo Local Program" from the menu.
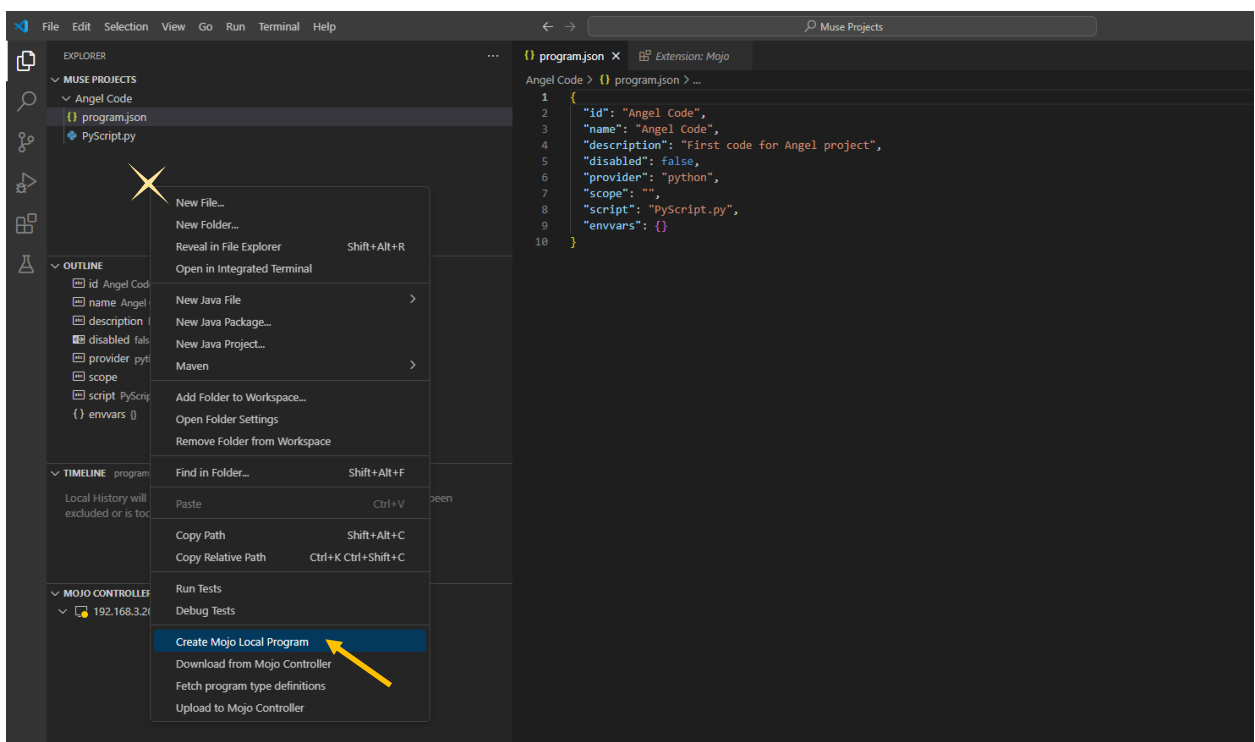
2. Follow the prompts at the top of the screen to create your project:
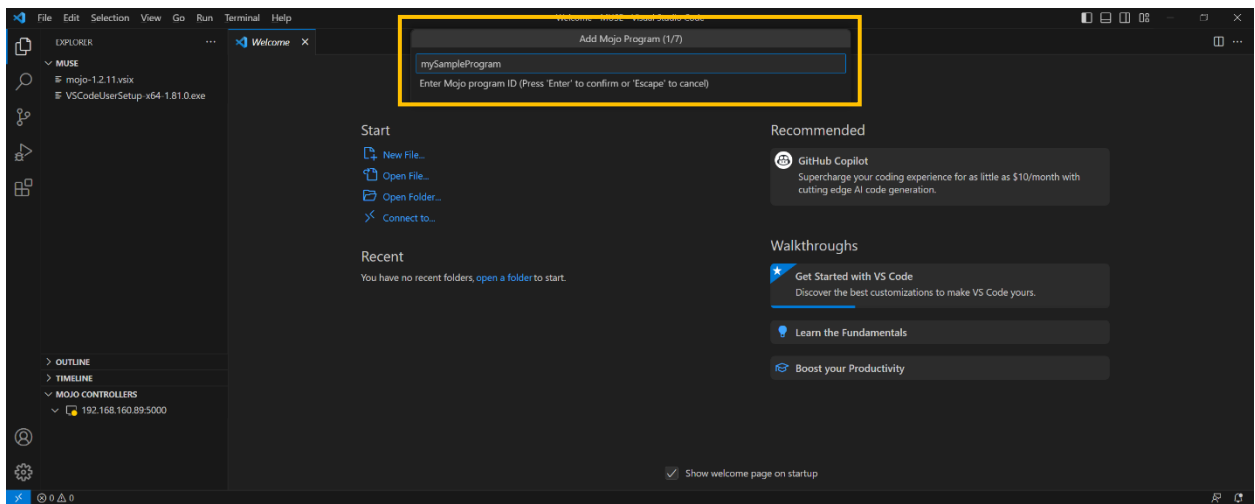
You will be prompted for:

- Mojo Program ID – The friendly name displayed within VS Code for the script
- Program Description – Programmer-provided metadata for the project
- Program Enable/Disable – A flag to determine if the project should run if loaded
- Program Provider – Choose the scripting language for the project
- Program Scope – Future use: logically divide devices and scripts into groups
- Program Entry Script – The name of the script to use as a starting point

At this stage a folder and two files have been created for your MUSE script based on your input.
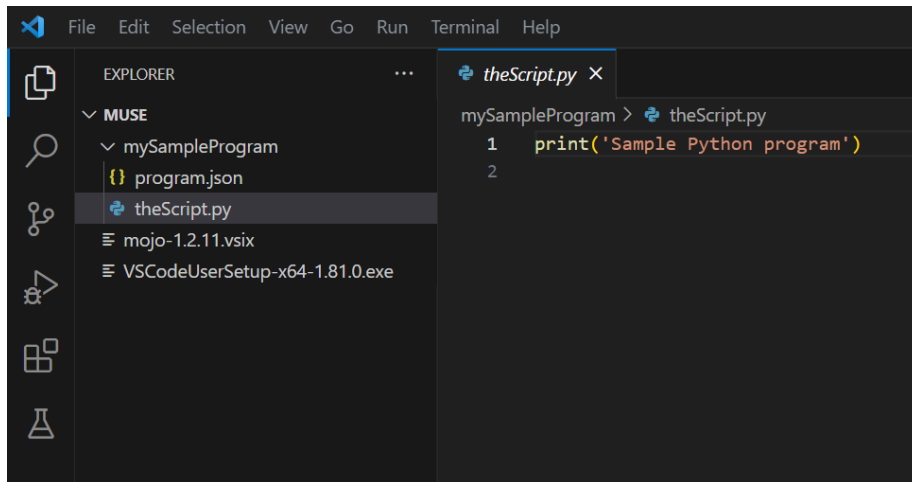
In this example, 'theScript.py' is the Python script that has been created. The *program.json* file contains the MUSE script's target information. If at some later time you create a different Python file in this project and would like to choose it as the starting point, you could edit *program.json* to make it so.
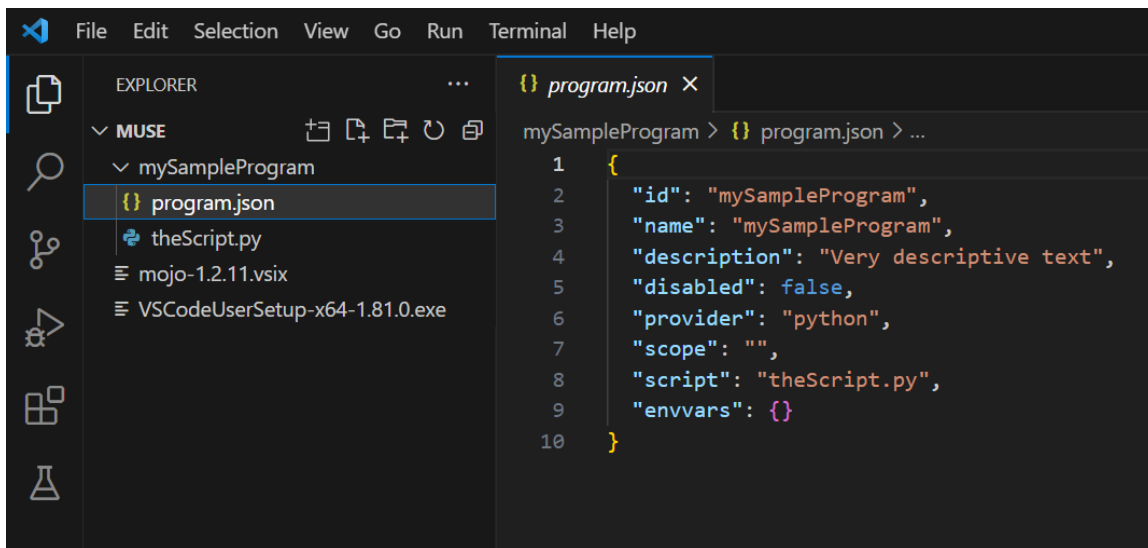
For a basic script that runs at startup, no further intervention is needed.

NOTE: For Python scripts only, the following two lines need to be added to your script to properly use all the available MUSE resources:

At the top:
```
from mojo import context
```

At the bottom:
```
context.run(globals())
```

Loading Scripts

To quickly load a script to a controller, right click on the script's main folder.   In the example above, **mySampleProgram** is the main folder.

Choose *Upload to Mojo controller*.   A dialog will appear at the top center of the VS Code application. First, confirm the script to be loaded.   Next, choose any controller you are currently attached to as a target for uploading.   Clicking **OK** will upload the script and start it running.
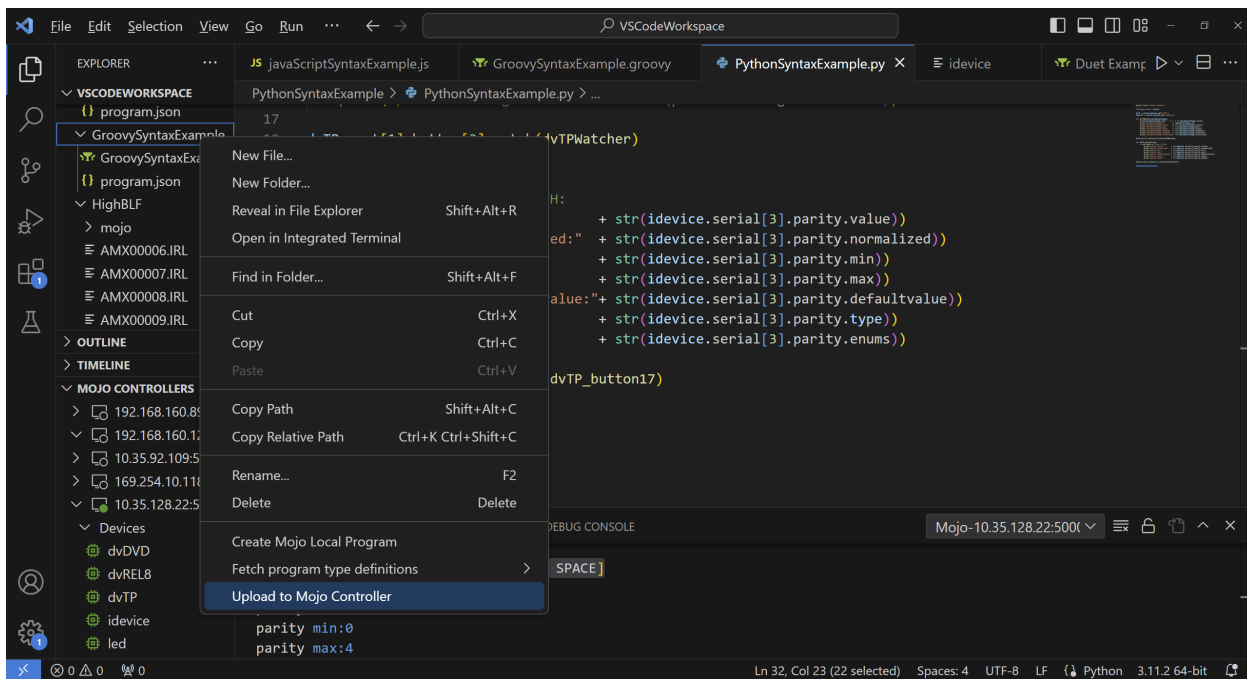
Install MUSE Controller Firmware

To use the latest VS Code Extension with an AMX MUSE controller, you will need to update the MUSE controller firmware available on amx.com. This table shows the required compatible firmware & application versions:

Device/Application	Initial Release	Latest Release
MUSE Firmware	1.1.43	1.2.65
MUSE Automator	1.0.54	1.1.12
VS Code Extension	1.2.16	1.4.7
Minimum Required Firmware		
CE Control Extenders		1.2.7
Varia Touch Panels		1.11.42

Firmware can be loaded via the MUSE web interface. Browse to the IP of the MUSE controller and navigate to the *System >> Firmware >> Upgrade Firmware* window. Browse your file system for the firmware .SWU file and select the [Upgrade Firmware] button.

Note: Attempting to downgrade from MUSE firmware 1.2.to 1.1.x may have unexpected results. To downgrade, the user must reset the MUSE to factory firmware, and then load 1.1.x firmware.

A unit shipped from the factory with 1.2.x firmware cannot be downgraded to 1.1.x.

Device Declarations

MUSE uses a dynamic device binding system that allows each running script to access any device that is attached to the MUSE Controller. Touch panels, keypads, modules, and other ICSP, HiQnet, & HControl devices are configured outside of the script.

Devices are acquired by the scripts by name. The naming of the attached devices depends on the connection type.   Some are assignable at the time of configuration, and some are automatic.
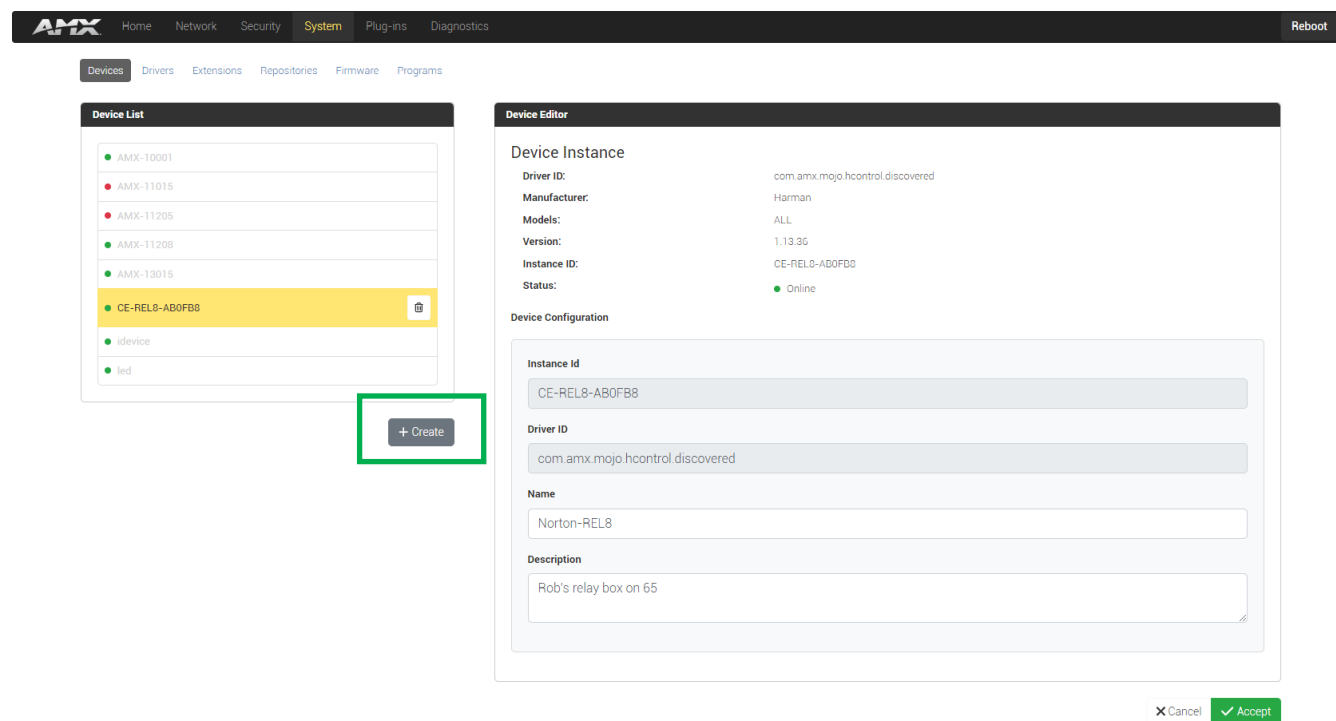
The primary method to declare a device is through the MUSE Controller's web interface. Depending on the connection type, some devices may auto-register. The following is a description of each device type:

ICSP

ICSP connections have multiple behaviors that may be supported. URL mode, Listen mode, NDP each behave slightly differently.

URL Mode

The most common connection method is URL mode. The peripheral devices (a touch panel, keypad, etc.) is given the IP address or DNS resolvable hostname of the MUSE Controller as a target to connect. The device reaches out on TCP port 1319 (or 1320 for Secure ICSP) and negotiates a connection with the controller.

Once connected, URL Mode devices will be automatically assigned a unique name based on the device number. If you have a touch panel set to device number 10001, the device name will be AMX-10001. This **Instance ID** is what is used to access the device from a script.

NDP

MUSE supports the NDP method of connection.   Devices in NDP mode can be discovered and bound to the controller. The controls are available in the CLI and the controller's web server.

Once created, it will behave as a URL mode device and receive a unique name based on the device number. This **Instance ID** is what is used to access the device from a script.

Listen Mode

ICSP Listen Mode reverses the connection initiation. A device in Listen mode opens a socket to listen to incoming port 1319 connections. It is the controller's responsibility to initiate a connection with the device. To inform the controller of the IP address or DNS resolvable hostname of the target device, a Device needs to be configured. The primary method of configuring the device is on the MUSE Controller's web interface. Point a web browser to the URL of the MUSE Controller and perform the login as prompted.

Next, navigate to the Devices page by choosing the System tab, then selecting Devices.   Click the **Create+** button to start configuration of the device.

For each device configuration, a unique name is required.   The **Instance ID** is what is used to access the device from a script.

Device Names must consist of only the following characters:

- Upper-case alphabetic characters (A - Z)
- Lower-case alphabetic characters (a - z)
- Numerals (0 - 9)
- Hyphen ("-"), underscore ("_") and space (" ")

Device Names must begin with an alpha-numeric character.
Device Names must not end with a space.

To create an ICSP Listen Mode device definition, use the Driver ID pulldown and choose the *com.amx.thing.ICSP* entry.   This will reveal two more fields that are specific to the ICSP driver.   The IP address field is used to convey the target IP address for the device in Listen mode. The target port is read-only information containing the default ICSP port.

Create a New Device

Instance Id:

dvTP

Driver ID:

com.amx.thing.icsp

Name:

Varia TP

Description:

Varia SL50

Device Configuration

Port

1319

IP Address

198.18.0.2

X Cancel   ✓ Accept

Regardless of connection method, **Name** and **Description** are metadata fields reserved for friendly names & other information. They do not affect the functionality of the ICSP device.

HControl

There are three methods to create an HControl device: controller, direct, or discovered. Please note the com.amx.mojo.*hcontrol.generic* device connection has been replaced, and your HControl devices will need to be connected using one of the following more specific connection types:

Controller

Selecting *com.amx.mojo.hcontrol.controller* creates a Controller-to-Controller device where the MUSE controller can control a device bound to a second MUSE controller.

Direct

Selecting *com.amx.mojo.hcontrol.direct* creates a direct connection to an HControl device using its HControl name and IP address. This is useful if a device is not automatically discovered (e.g., if it is on a different subnet).

Discovered

Selecting *com.amx.mojo.hcontrol.discovered* creates a device using just the HControl name of an already-discovered device. All HControl devices on the same subnet as the controller will be automatically discovered as part of the HControl discovery protocol. Discovered devices can be found either in the HControl device list in the Mojo VS Code Extension, or by using Manager software.

Create a New Device

Instance Id:*

dv-REL8

Driver ID:*

com.amx.mojo.hcontrol.direct

Name:

CE-REL8

Description:

CE Series Relay Expansion Box, 8-port

Device Configuration

Port

4197

IP Address

192.168.3.115

✕ Cancel   ✓ Accept

Instance ID is the HControl name of your intended device, and must match the device to establish an HControl connection. The HControl name can be found either in the HControl device list in the Mojo VS Code Extension, or by using Manager software.

Device Names must consist of only the following characters:

- Upper-case alphabetic characters (A - Z)
- Lower-case alphabetic characters (a - z)
- Numerals (0 - 9)
- Hyphen ("-"), underscore ("_") and space (" ")

Device Names must begin with an alpha-numeric character.
Device Names must not end with a space.

Regardless of connection method, **Name** and **Description** are metadata fields reserved for friendly names & other information. They do not affect the functionality of the ICSP device.

LDAP Integration

The LDAP page provides configuration and tests connection to a remote directory service via LDAPv3. The controller supports the option of an insecure or secure connection. The secure option is supported via the StartTLS command in LDAP and also via "LDAPS", or LDAP over SSL/TLS on port 636. Select the LDAP option on the Security Page to access the LDAP Settings page. The options on this page allow authorized users to enable and modify LDAP security settings.

LDAP Options

Please note that some LDAP parameters are case sensitive and must be entered exactly as they are entered into the LDAP database. You can also perform LDAP Client Configuration via terminal commands to the MUSE controller's Program Port.

LDAP Options	
Option	**Description**
LDAP Enabled	This parameter enables the LDAP configuration parameters described below. **NOTE:** When LDAP is enabled, you can only create device users. If the administrator user has been deleted, you must perform a factory reset of the controller via pushbutton to restore the administrator user.
LDAP.LDAPS URL	This parameter has the syntax **ldap[s]://hostname:port**. • The **ldap:// URL** is used to connect to LDAP servers over unsecured connections. • The **ldaps:// URL** is used to connect to LDAP server over Secure Sockets Layer (SSL) connections. • The **hostname** parameter is the name or IP address, in dotted format, of the LDAP server (for example, *LDAPServer01* or *192.202.185.90*). • The **port** parameter is the port number of the LDAP server (for example, 696).

LDAP Options	
	NOTE: The standard unsecured port number is 389 and the standard secured port number is 636.
LDAP/LDAPS BASE DN	This parameter specifies the Distinguished Name (DN) of an entry in the directory. It identifies the entry that is the starting point of the user search.
BIND DN	This parameter specifies the Distinguished Name (DN) to use to bind to the LDAP server for the initial search for the user's DN.
User Query Attr	This LDAP attribute is used for the AMX equipment user search (e.g. UID). **NOTE:** This attribute MUST be unique in the context of the LDAP BASEDN or the search will fail.
Search Password	This is the password used for the initial bind to the LDAP server - it is the password associated with BIND DN.

Click the **LDAP enabled** check box to make the LDAP options available for selection.

- When LDAP is enabled, users are authenticated using the configuration set up on the LDAP server.
- The "*admin*" user is handled by the local MUSE, and does not connect to the LDAP server for user verification.
- As users log onto a controller, their user name and access privileges are displayed on the User Security Details page (Security >> Users). This information is stored in the controller's RAM but is not written to nonvolatile memory, and is lost after rebooting the controller.
- If a user is removed from the LDAP directory tree & access is denied. If that user name is on the controller's User Security Details web page, it is removed.

Accepting Changes

Click the **Test/Accept** button to save changes on this page. Accepting changes is instantaneous and does not require rebooting the controller.

Testing Connection to the LDAP Server

After entering and accepting the parameters, the Accept/Test button can be used to test the connection to the LDAP server. This test does a bind to the BIND DN using the Search Password entered.

- If connection test successful, displays "LDAP Settings saved and confirmed"
- If connection test fails, displays "LDAP Settings saved but not confirmed. "
  - In this case the admin should check all entries to make sure correct.

Device Online/Offline Behavior

There are two ways to determine the online/offline status of a Device.

The get the current state, you can call the .isOnline() or .isOffline() method.

To be informed of changes to the online/offline status of the Device, use the .online() or .offline() method to register a callback.

Interpreting Descriptor Files

Any Device in MUSE has its capabilities described in a Descriptor File. This document contains information about the interface you can use to interact with the device.   In the case of HControl, this document is provided by the device itself during the connection process.   In the case of other communication methods like ICSP and HiQnet the document is dynamically generated based on information provided during the connection process.

The Descriptor File is a .json formatted document. This makes it reasonably human readable and available to parse programmatically.

There are several ways to access this information. The VS Code extension provides a quick way to see an expanded view of the information.

Expand the tree view of the controller to reveal the Devices node:
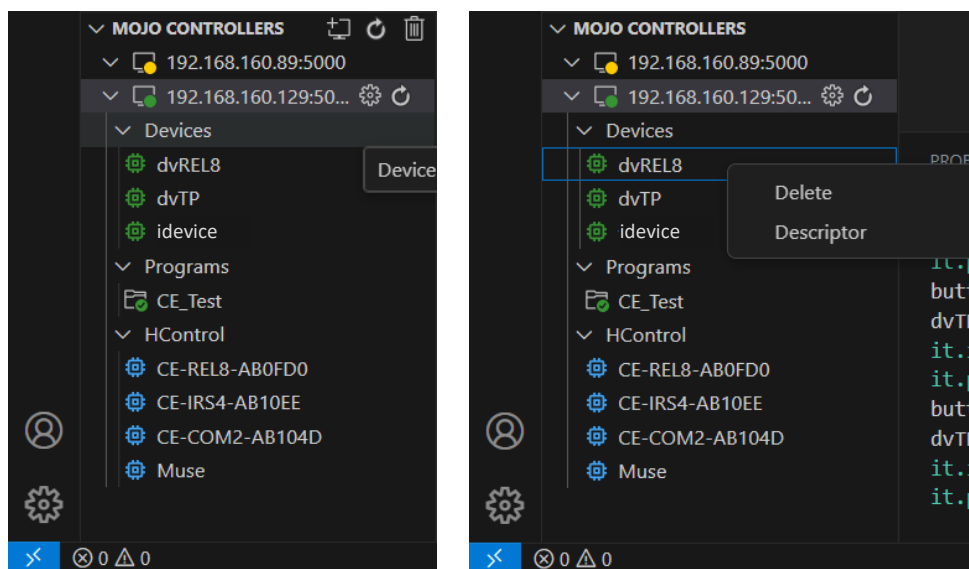
Right click on the device in question and choose Descriptor from the context menu. A file will open that is an interpretation of the Descriptor file, not the file itself.

```
JS TascamDVD.js      🐍 MojoToNetLinx.py      🐍 blf.py      ☰ CE-REL8-AB0FD0      🟢 CE.groovy      ☰ dvREL8   ✕

 1   |------------------------------------
 2   com.amx.mojo.hcontrol.generic
 3   ------------------------------------
 4   |
 5   +-- id:           com.amx.mojo.hcontrol.generic
 6   +-- name:         CE-REL8
 7   +-- manufacturer: Harman
 8   +-- models:       [ALL]
 9   +-- version:      1.11.107
10   +-- description:  HControl based 8 relay expansion box
11   +-- dynamic:      false
12   +-- oneBased:     false
13   |
14   +- PARAMETERS (1)
15   |  +-- version <string> (ro)
16   |       +- [ <string>  min=0
17   |
18   +- COMMANDS (1)
19   |  +-- locate
20   |
21   +-- OBJECTS (1)
22   |  +-- configuration
23   |      +- PARAMETERS (1)
24   |      |  +-- commands <string> (rw)
25   |      |       +- [ <string>  min=0
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

The file tends to start with configuration information that may not apply to the programming task at hand. The typical control-related information will be in the Arrays portion at that end of the file.

```
JS TascamDVD.js   🐍 MojoToNetLinx.py   🐍 blf.py   ☰ CE-REL8-AB0FD0   🟢 CE.groovy   ☰ CE-REL8-AB0FD0   ☰ dvREL8   ✕

137   |
138   +- ARRAYS (1)
139   |  +-- relay
140   |       +- ELEMENTS (8)
141   |            +-- [0]
142   |            |  +- PARAMETERS (1)
143   |            |      +-- state <boolean> (rw)
144   |            |           +- [ <boolean>  min=0 max=1
145   |            |           +- [ +- METADATA (1)
146   |            |           +- [    +--- desc = Parameter to set a relay port on/off
147   |            +-- [1]
148   |            |  +- PARAMETERS (1)
149   |            |      +-- state <boolean> (rw)
150   |            |           +- [ <boolean>  min=0 max=1
151   |            |           +- [ +- METADATA (1)
152   |            |           +- [    +--- desc = Parameter to set a relay port on/off
153   |            +-- [2]
154   |            |  +- PARAMETERS (1)
155   |            |      +-- state <boolean> (rw)
156   |            |           +- [ <boolean>  min=0 max=1
157   |            |           +- [ +- METADATA (1)
158   |            |           +- [    +--- desc = Parameter to set a relay port on/off
159   |            +-- [3]
160   |            |  +- PARAMETERS (1)
161   |            |      +-- state <boolean> (rw)
```

PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

From this example, we can see that the CE-REL8 has an array named 'relay' with 8 elements. Each relay has a parameter named 'state' that is a Boolean. If we want to engage the relay, we would set the 'state' parameter to true.

Here are some syntax examples in the supported languages. In each, assume we have a Device reference held in a variable named dvREL8.

Groovy syntax example:

```
dvREL8.relay[0].state = true
```

Python syntax example:

```
dvREL8.relay[0].state = True
```

JavaScript syntax example

```
dvREL8.relay[0].state = true
```

For this activity, there is very little difference between the syntax in any particular language.

More ways to get the Descriptor and related information

The MUSE controller offers an SSH connection that has many useful CLI commands.   To get the same interpretation of the descriptor is available in VS Code, the CLI command is:

```
admin@mojo>driver:describe dvREL8
```

To obtain the actual JSON document, use:

```
admin@mojo>driver:describe -j dvREL8
```

To get the list of commands or parameters that pertain to the device, use:

```
admin@mojo>device:commands
admin@mojo>device:parameters
```

To obtain a list of current devices, use:

```
admin@mojo>device:list
```

Event-based Programming

Adding a watch/listen

One of the most basic requirements of A/V control system programming is reacting to user input from some form of user interface. Typically, touch panels & keypads provide the user the ability to inform the system of their desired functionality.

MUSE has two methods to be notified about changes: **.watch( )** and **.listen( )**

The **.watch( )** method is for parameters.   Parameters are attributes that keep their state and can be queried for their value.

The **.listen( )** method is for events.   Events, once fired, are gone. You cannot query the value of an event outside of the .**listen( )** callback.

In MUSE, to react to button presses you create a ***watch,*** because buttons are defined as parameters. The watch argument contains a callback function that will be called to process the button event.

Groovy Syntax:

```
dvTP.port[1].button[255].watch( {
    println("button is "+it.value)
    println("dvTP button "+it.id+" "+ ((it.value == true)?"pressed":"released"));
    println("it.id=="+it.id)
    println("it.path=="+it.path)
    })
```

Groovy provides a convenient mechanism for writing complex code within the watch. This Groovy Closure includes the passed parameter containing the event information. If unnamed, the passed parameter is named *it*.   In this example, *it* is a Parameter Update Structure that contains everything about the parameter change notification.

For ICSP events, the path contains the port and channel of the button that caused the event in the form port/<p>/button/<b>. So, if button 1 on port 1 was pressed, the event.path value would contain:

```
port/1/button/1
```

In languages that support regex, this lends itself to a convenient pattern match.   For example, in Groovy you can use:

```
def eventPathPattern = ~"port/[0-9]+/button/[0-9]+"
```

…then extract the event data with:

```
def eventMatches = it.path =~ eventPortAndChannelPattern
eventMatches.find()
```

`EventMatches` then becomes a collection with the same port & channel numbers as the two elements.

Python Syntax:

```
dvTP.port[1].button[1].watch(dvTP_port1_button1_watch)
```

…where dvTP_port1_button1_watch is a user defined function used as a callback.   When the event is processed, this function will be called and passed a parameter update structure containing the parameter change information.

Elsewhere in your code, you need to implement this function:

```
def dvTP_port1_button1_watch(theButtonPress):
    #do stuff with the data in theButtonPress
    print('theButtonPress.path==' +theButtonPress.path)
```

All the event parameters are available in each language. See The Parameter Update Structure for details.

JavaScript syntax:

```
dvTP.port[1].button[4].watch(dvTP_button1_watch)
```
…where dvTP_button1_watch is a user define function used as a callback.

Elsewhere in your code, you need to implement this function:

```
function dvTP_button1_watch(theButtonPress) {
    context.log('dvTP_button_watch(theButtonPress) called')
    context.log(path:'+theButtonPress.path+" is "+theButtonPress.newValue)
```

where theButtonPress is a structure passed into the callback. All the event parameters are available in each language. See The Parameter Update Structure and The Event Structure for details.

Timelines

MUSE provides a centralized timing mechanism to use within the scripts. To ensure that events are processed in the order that they are received, events are processed one at a time. Consequently, invoking a *sleep* method on a thread that is processing an event will delay the processing of the next event until the *sleep* returns.   Using a Timeline side-steps this issue by running the delayed code in the centralized timing thread.

To use a Timeline, an instance of a timeline needs to be retrieved from the context.

In Python, the syntax is:

```python
tick = context.services.get("timeline")
```

In this case, the variable *tick* will contain the timeline object. To start the timeline, the syntax is:

```python
tick.start([1000],True,-1)
```

…where [1000] is an array of times in milliseconds.

The Boolean is the 'relative' flag.   If there are multiple times in the array, it determines whether the timings are treated as relative delays between triggers, or as an independent list of times that may trigger out of sequence relative to their order in the list.

The final parameter is the 'repeat' value. A value of -1 indicates that the timeline should run forever. A timeline started with a value of 0 will run once. The value indicates the number of repetitions.

To receive events from the timeline, use this syntax:

```python
tick.expired.listen(tickListener)
```

Where *tickListener* is the function that will be called when the timer expires. This is not a fixed name. The script writer creates and names this function:

```python
def tickListener(tlEvent):
     # do the timed stuff
```

The *tlEvent* parameter contains information specific to the timeline event that just triggered.

```python
def tickListener(tlEvent):
     print("ticklistener expired: sequence=" + tlEvent.arguments["sequence"] +
       ", time=" + tlEvent.arguments["time"] +
       ", repetition=" + tlEvent.arguments["repetition"])
     Repetition = tlEvent.arguments["sequence"]
```

Timelines have the following methods: start(timeArray,relative,repetition), stop(), pause(), restart()
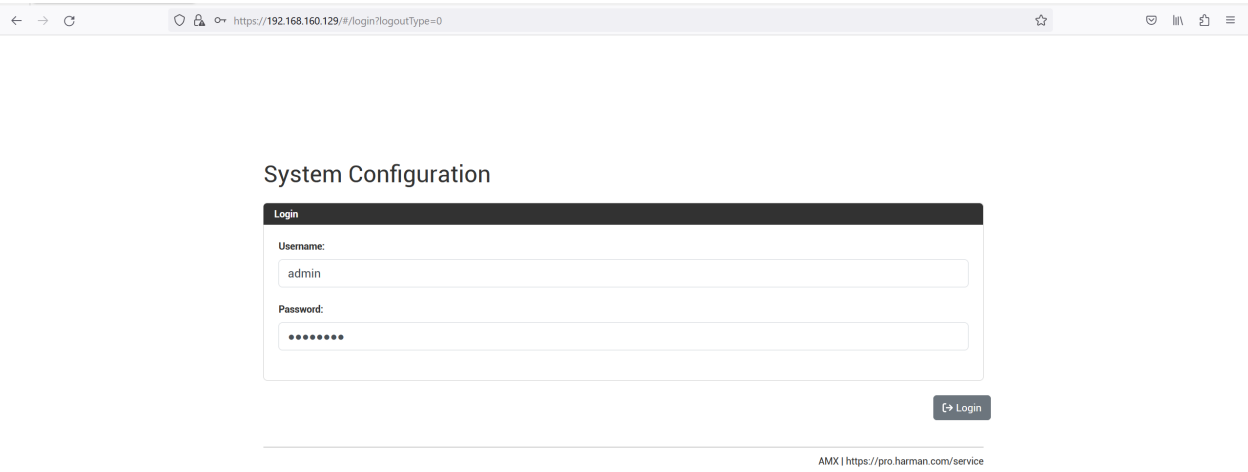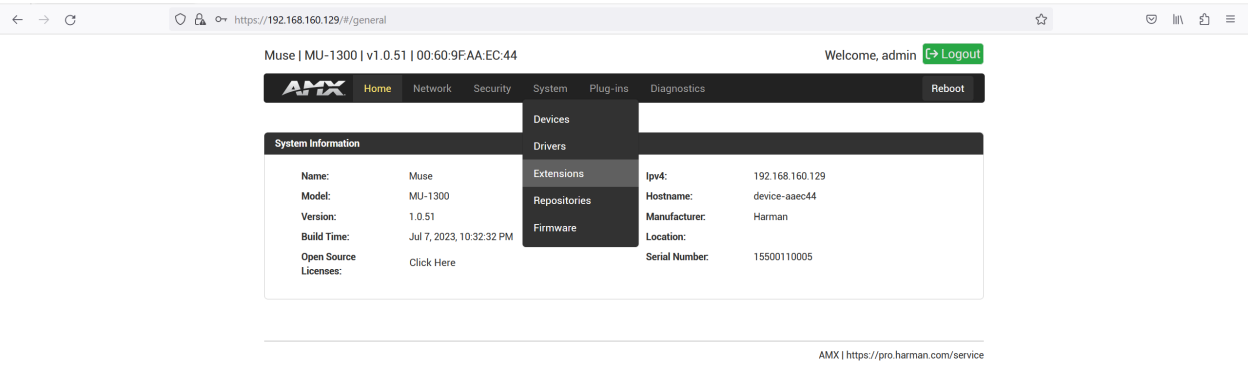
Duet Modules

Initial setup

Adding the Duet Extension

By default, the Duet Driver Provider is not installed on the MU-series controller. This Extension must be enabled to load and run Duet-based drivers.

Log in to your MUSE controller's configuration web page:

In the menu bar, choose System -> Extensions

Find **mojo-duet** in the list of available extensions:

We can see that the Extension's status is 'uninstalled'. With **mojo-duet** selected, click **Install**

After a few moments, the 'busy' screen will disappear, and the **mojo-duet** extension is enabled.

Loading a Duet module's .jar file
Choose Plug-ins -> Duet in the MUSE controller's web configuration menu bar
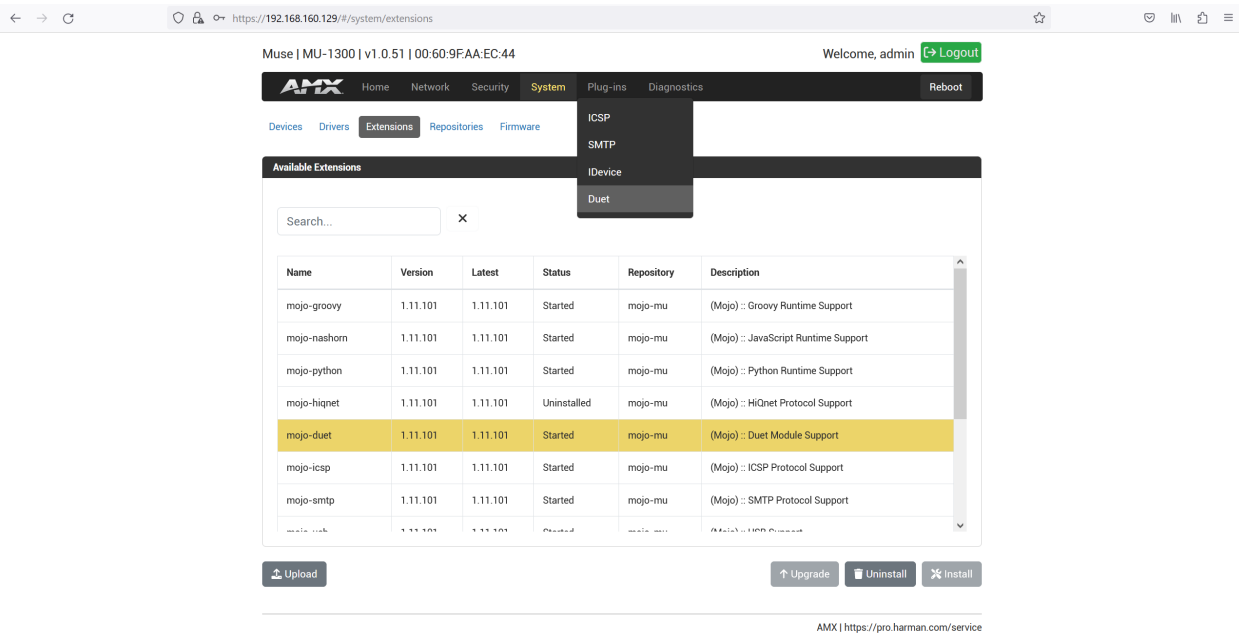
Click Upload to find and load a Duet .jar file
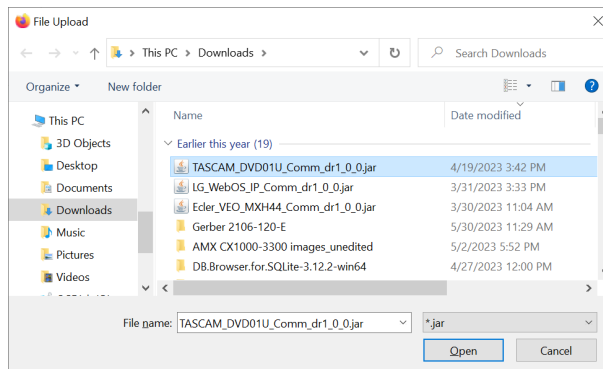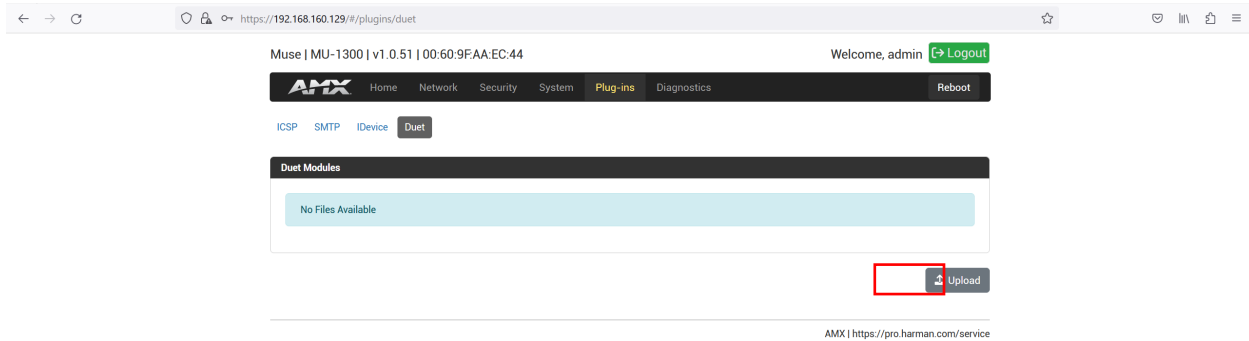
The Duet Driver is now available for use.
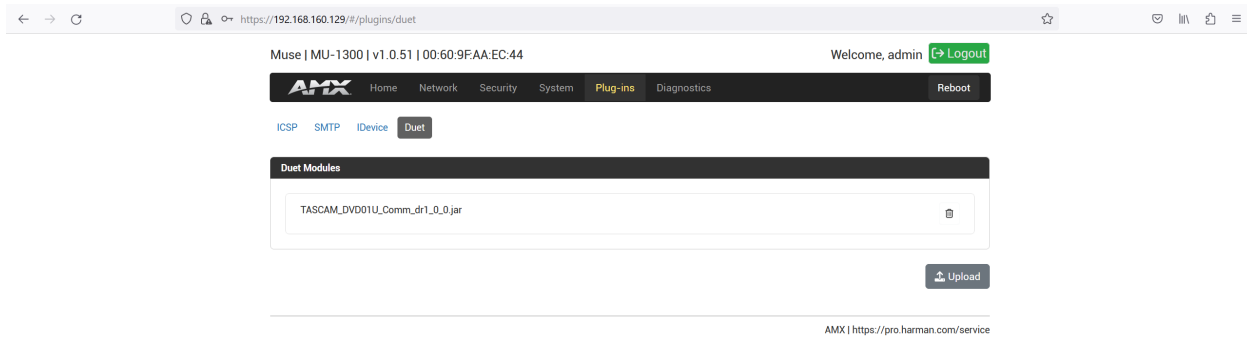
Make a Driver instance

Choose System -> Devices to navigate to the Devices page:
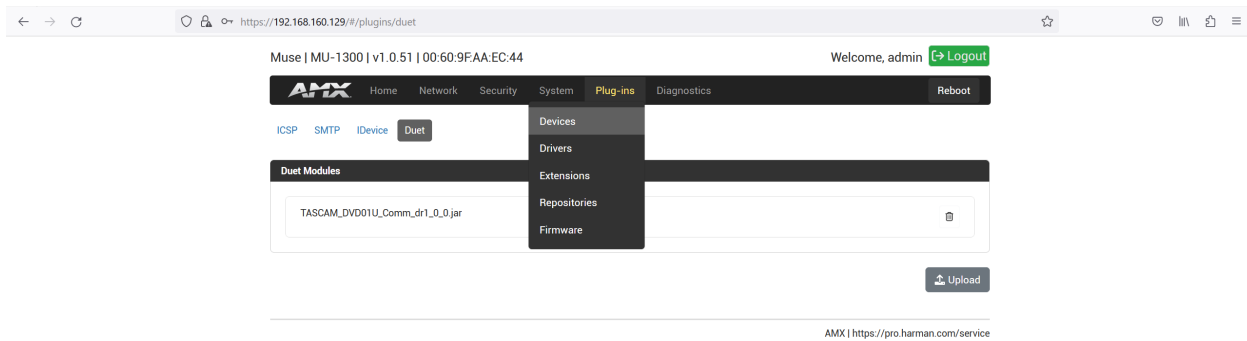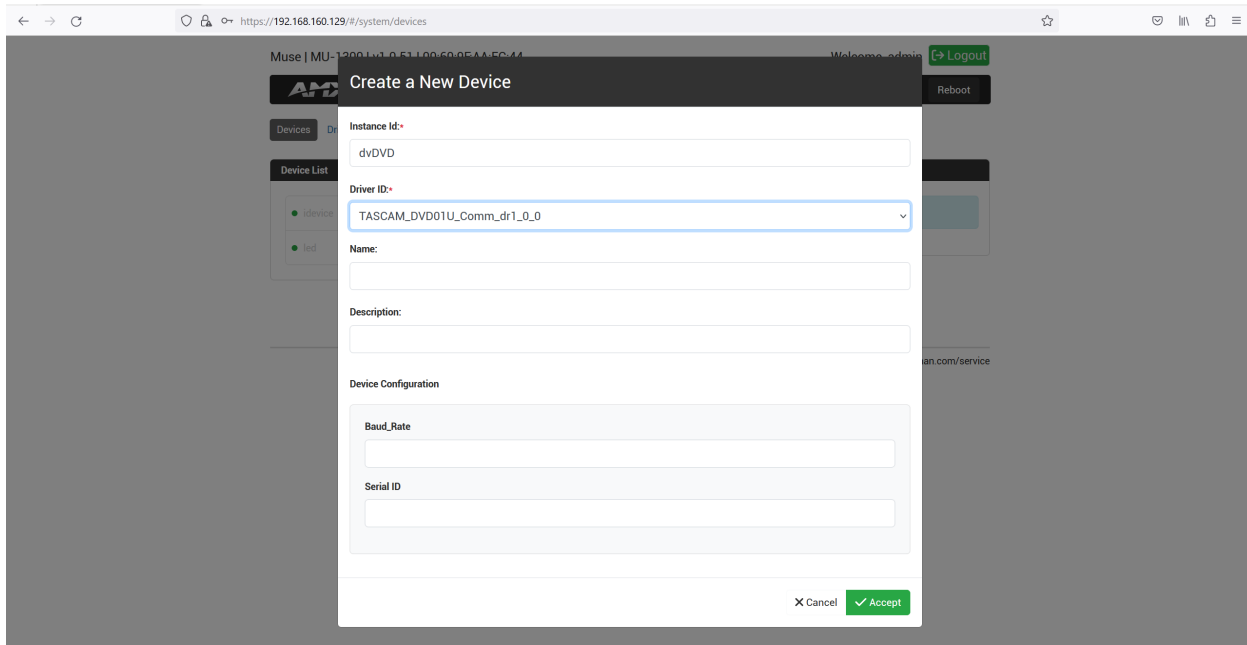
On the Devices page, click the Create button:
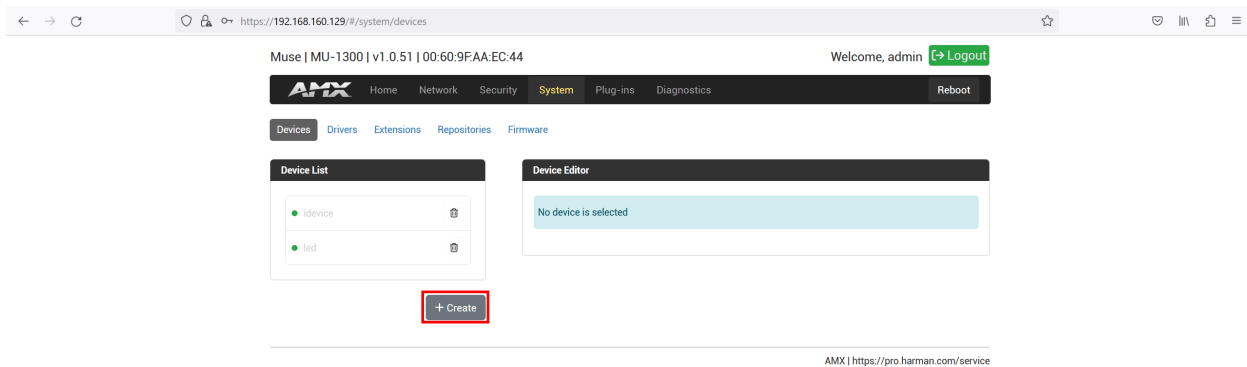
In the resulting dialog box, name the Device and choose the Duet Module to use:
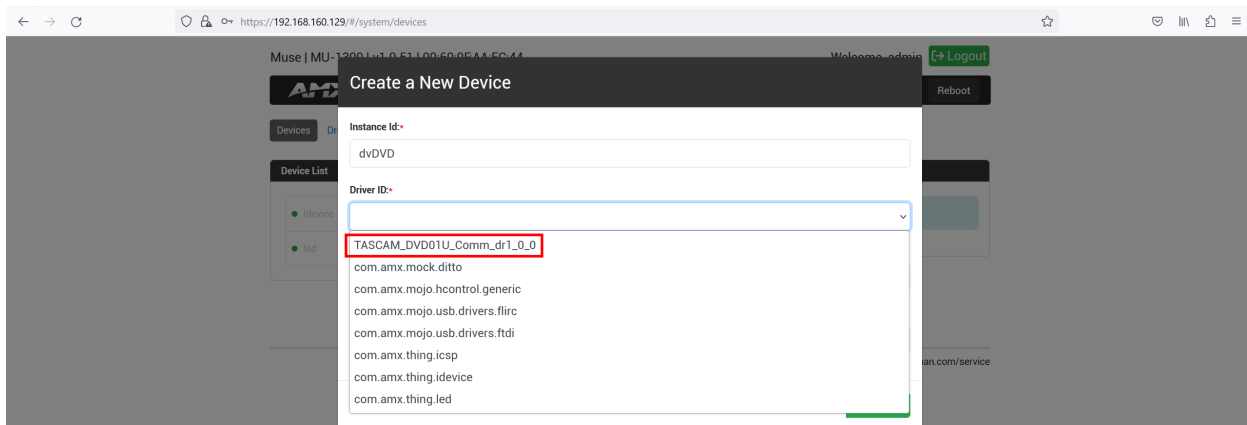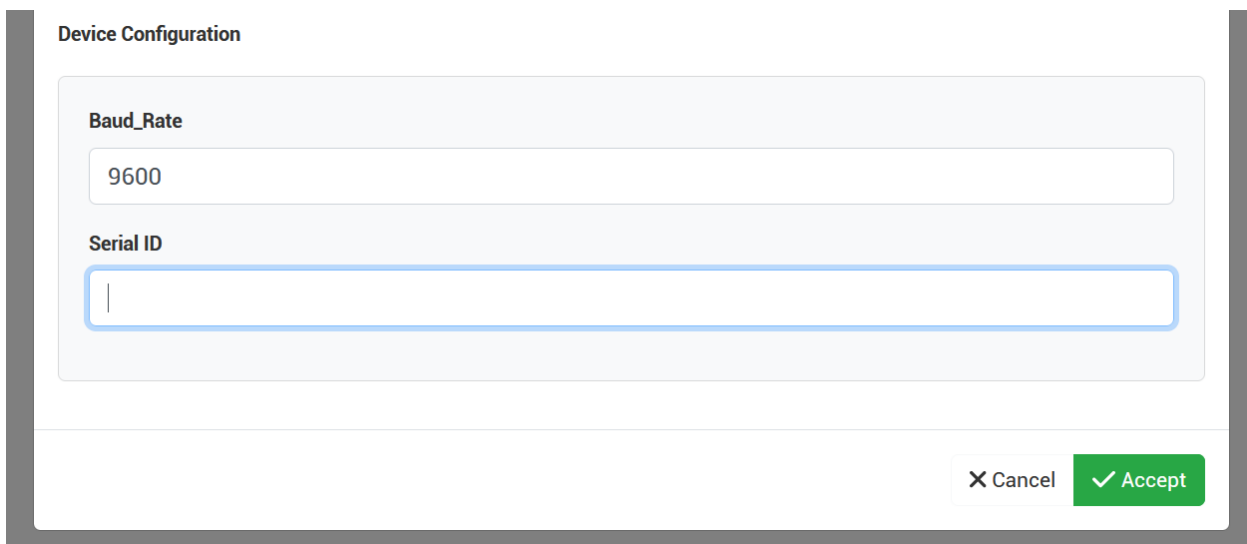
Once selected, the dialog box will now contain new fields relevant to the Duet Module:

Fill out the fields in the Device Configuration section to inform the Duet module about the setup

For the Serial ID, the MUSE controller has a list of all available COM ports on the controller. In future, this will be populated in a pulldown. Until that time, we need to ask the MUSE controller for that list.

In an SSH session with the controller, log in and type the following command:

```
io:list
```

From this list, choose the COM port that the device is physically wired to.

```
192.168.160.129 - PuTTY                                    —    □    ✕

End of keyboard-interactive prompts from server


      /\ \/ /\      ___  ( )    ___
     /  \/  \ \    /   \     /      \
    /   |   \ \   |    \/   |  /| |\ \
    \   \___/  \  |    |   /| / | | \ \
     \__/\_/\__/  /\_| |  /\_|   | |  \
        \/          \__ /      /\_|  |
                                  \__/

   Mojo Controller  (1.11.101)

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown Karaf.

Password change not required for user admin

admin@mojo()> io:list
ID                 | Type
-------------------+-----------------------------
idevice-serial-0 | com.amx.mojo.api.io.SerialPort
idevice-serial-1 | com.amx.mojo.api.io.SerialPort

admin@mojo()>
```

Note: Mojo keeps the COM ports organized in an array. Consequently, the list is numbered in a zero-based index as most modern programming environments do.   Please note this differs from the actual silkscreen on the MUSE unit, which is one-based.   In the example above, an MU-1300 is being used. There are two serial ports available on the controller. To use Serial Port 1 as labelled on the controller, choose:

```
idevice-serial-0
```
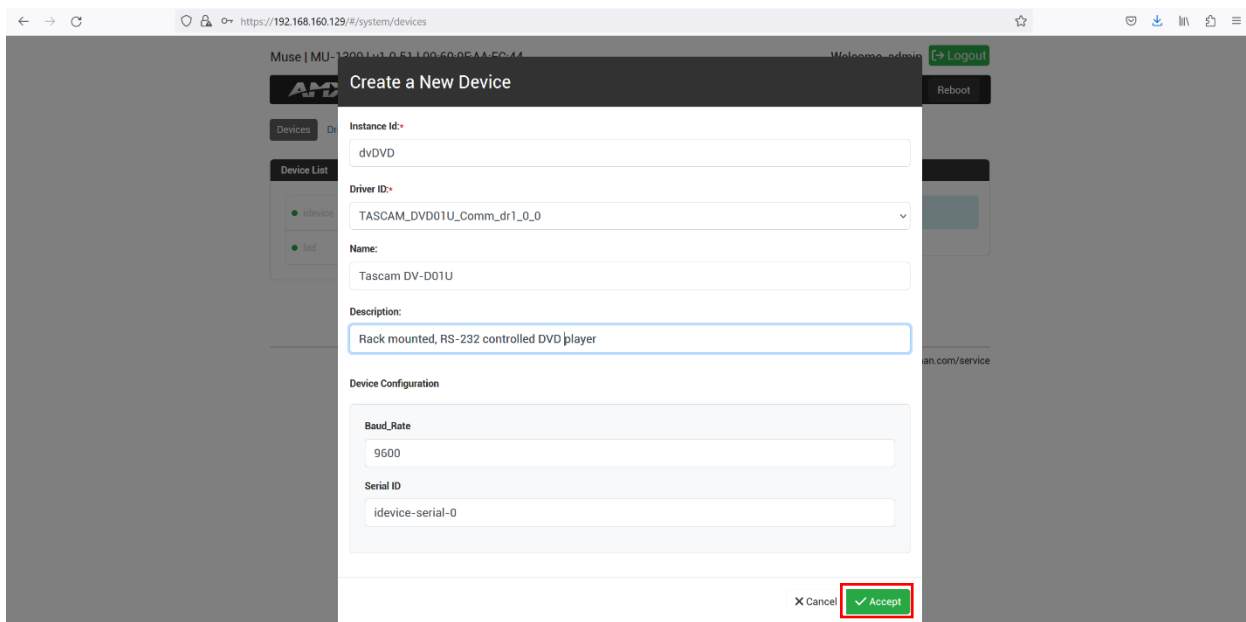

Click Accept to bind the driver to the COM port.
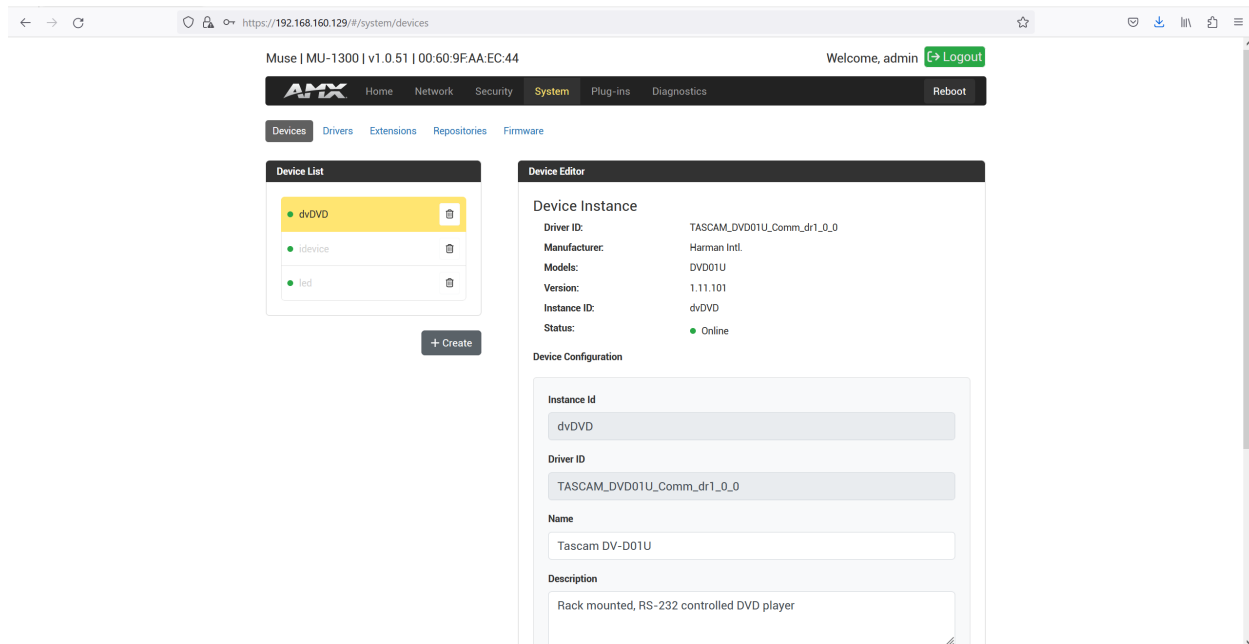
Using a module instance

Once the module instance is created, the module itself will start. This may include connecting to the device, polling, and other startup behaviors. The module instance will be listed by name in the Device List portion of the MUSE Controller's web server and the device:list CLI command:

```
admin@mojo()> device:list
ID       | Scope     | Status | Driver ID                     | Version
---------+-----------+--------+-------------------------------+--------
dvDVD    | <GLOBAL>  | Online | TASCAM_DVD01U_Comm_dr1_0_0     | 1.12.12
dvTP     | <GLOBAL>  | Online | com.amx.thing.icsp            |
idevice  | <GLOBAL>  | Online | com.amx.thing.idevice         |
led      | <GLOBAL>  | Online | com.amx.thing.led             |

admin@mojo()>
```

The CLI provides some focused insight to the control of the device through the module. Two helpful CLI commands are: device:commands and device:parameters.   The essential difference between commands and parameters is persistence.   Parameter values stay and can be queried. Commands happen and then are gone.

Asking the CLI what command our DVD player has results in this output:

```
  admin@mojo()> device:commands dvDVD
------------------------------------------------
DEVICE COMMANDS (dvDVD)
------------------------------------------------
> "discDevice/0/cycleDiscTray"
> "discDevice/0/cycleRepeat"
> "discDevice/0/queryDiscProperties"
> "discDevice/0/queryDiscProperty" (1 arguments)
```

```
> "discDevice/0/queryTitleProperties"
> "discDevice/0/queryTitleProperty" (1 arguments)
> "discDevice/0/setPlayPosition" (2 arguments)
> "discTransport/0/cycleScanSpeed"
> "discTransport/0/getTrackInfo"
> "discTransport/0/queryTrackProperties"
> "discTransport/0/queryTrackProperty" (1 arguments)
> "menu/0/moveCursor" (1 arguments)
> "menu/0/pressButton" (1 arguments)
> "menu/0/selectItem"
> "module/channel" (3 arguments)
> "module/command" (2 arguments)
> "module/getInstanceProperty" (1 arguments)
> "module/intLevel" (3 arguments)
> "module/passThru" (1 arguments)
> "module/reinitialize"
> "module/setInstanceProperty" (2 arguments)
> "power/0/cycle"
-------------------------------------------------

Use device:invoke to execute a command

admin@mojo()>
```

The commands can be executed through the CLI as stated by this output.

For syntax help with this or any other CLI command, use the **man** or **help** CLI command:

```
admin@mojo()> man device:invoke
DESCRIPTION
        device:invoke

    Invoke a command on a specific runtime device.
 Arguments are presented as name/value pairs separated by '='.  e.g. day=tuesday time=12:00
where day and time are the name of the arguments

SYNTAX
        device:invoke id command [arguments]

ARGUMENTS
        id
                Instance ID
                (required)
        command
                Command Name
                (required)
        arguments
                Command Arguments (name=value format)
```

```
admin@mojo()>
```

So, to toggle the power on this DVD Player, we can say:

```
admin@mojo()> device:invoke dvDVD power/0/cycle
-------------------------------------------------
DEVICE INVOKE COMMAND
-------------------------------------------------
 INSTANCE  : dvDVD
 COMMAND   : power/0/cycle
-------------------------------------------------
Invoked Successfully!


admin@mojo()>
```

For parameters, the device:parameters command lists both the available parameters and their current value.

```
admin@mojo()> device:parameters dvDVD
-------------------------------------------------
DEVICE PARAMETERS (dvDVD)
-------------------------------------------------
> "configuration/device/classname" = com.amx.thing.duet
> "configuration/device/container" =
> "configuration/device/description" = This is a Duet module driver
> "configuration/device/descriptorlocation" = descriptor.json
> "configuration/device/devicestate" = Running
> "configuration/device/manufacturer" = HARMAN Intl.
> "configuration/device/name" = Duet Driver
> "configuration/device/protocolversion" =
> "configuration/device/serialnumber" =
> "configuration/device/softwareversion" =
> "configuration/device/venue" =
> "configuration/device/version" =
> "discDevice/0/discCapacity" = 1
> "discDevice/0/titleCounterNotification" = false
> "discTransport/0/discTransport" = PLAY
> "discTransport/0/trackCounterNotification" = false
> "module/dataInitialized" = true
> "module/debugState" = 2
> "module/deviceDateTime" = Thu Jan 01 00:00:00 UTC 1970
> "module/deviceOnline" = true
> "module/fwVersion" = 1.0.0
> "module/passBack" = false
> "module/version" = 1.0.1
> "power/0/power" = ON
```

```
------------------------------------------------------
admin@mojo()>
```

Parameter values can be set and get individually with the device:set and device:get CLI commands:

```
admin@mojo()> device:set dvDVD discTransport/0/discTransport PLAY
--------------------------------------------------
DEVICE PARAMETER SET
--------------------------------------------------
 INSTANCE : dvDVD
 PARAMETER : discTransport/0/discTransport
--------------------------------------------------
NEW PARAMETER VALUE: PLAY (1)


admin@mojo()>
```

This particular parameter has caused the DVD to start playing. For a full list of values, we can consult the descriptor file for dvDVD.   Typing driver:describe dvDVD gives the entire control set for the dvDVD device. The discTransport parameter specifically contains:

```
+- ARRAYS (4)
|   +-- discTransport
|   |     +- ELEMENTS (1)
|   |         +-- [0]
|   |             +- PARAMETERS (2)
|   |             | +-- discTransport <enum> (rw)
|   |             | |   +- [ <enum> [ INVALID, PLAY, STOP, PAUSE, PREVIOUS, NEXT, SCAN_FWD,
SCAN_REV, SLOW_FWD ]
```

If we stopped the DVD player and wanted to query the discTransport parameter, we can type:

```
admin@mojo()> device:get dvDVD discTransport/0/discTransport
--------------------------------------------------
DEVICE PARAMETER/METADATA GET
--------------------------------------------------
 INSTANCE          : dvDVD
 PARAMETER/METADATA : discTransport/0/discTransport
--------------------------------------------------
PARAMETER VALUE: STOP (2)


admin@mojo()>
```

These commands and parameters give us insight into how to use our device in control scripts. Duet modules have the concept of Components (typically arrays of Components) and methods/properties. For the above example, the discTransport component array has one element (index 0). This component has a property called discTransport, which is called to set or get the current state of the DVD.

Groovy syntax example:

```
dvDVD.discTransport[0].discTransport = "PLAY"
```

Python syntax example:

```
dvDVD.discTransport[0].discTransport = "PLAY"
```

JavaScript syntax example

```
dvDVD.discTransport[0].discTransport = "PLAY";
```

…as before, the individual bits of Mojo-related syntax barely differ between language.

Exporting Script Functionality

It is possible to expose the functionality of a script to the Mojo system. These exposed (or exported) touch points may then be access from HControl, scripts, or other control modules as a virtual device in the system. When any touch point is exported by the script, a virtual device is created in the system with the same ID as the script.

Note: This implies that scripts and real devices must have unique IDs amongst themselves. These touch points can be modeled as Thing parameters, commands, & events. They can also be contained in objects and arrays just like any other Thing

Script Descriptors

All devices in the Mojo / HControl ecosystem advertise their capability using the descriptor file. To expose the touch points in the script, the programmer must expand the *program.json* file to advertise its parameters, commands, & events.

```
Full Script Descriptor
{
    "$schema": "https://developer.amx.com/schema/amx-mojo-program-1.0.json",
    ".metadata" : {
        "id" : "jstestapp",
        "name" : "JavaScript Driver Program",
        "disabled": false,
        "description" : "JavaScript Driver Program",
        "provider" : "javascript",
        "script": "test.js"
    },

    "led": {
        ".kind": "array",
        ".prototype": {
            ".kind": "param",
            ".type": "enum",
            ".enums": ["OFF", "ON"],
            ".metadata": {
                "setter" : "setLED"
            }
        },
        ".size": 5
    },

    "setAllLeds": {
        ".kind": "command",
        ".arguments": {
            "value": {
                ".type": "enum",
                ".enums": ["OFF", "ON"]
            }
        }
    },

    "power": {
        ".kind": "param",
        ".type": "boolean"
    },
```

```json
    "cyclePower": {
        ".kind": "command",
        ".metadata": {
            "method" : "cyclePower"
        }
    },

    "goodbye": {
        ".kind": "event",
        ".arguments": {
            "who": {
                ".type": "string"
            }
        }
    }
}
```

In the above example, there are two parameters and two commands. One of the parameters (led) is actually an array of parameters. This is standard descriptor content with the exception of two new .metadata keys: "setter" and "method". If the .metadata of a parameter contains a "setter" key, the system will look for a top-level function in the script with the same name as the key's value to call whenever a request to set the parameter is received. Likewise, if a "method" key is found in the .metadata of a command, the associated value will be assumed to be a top-level function to call to invoke the command. If a parameter or command doesn't have the "setter" or "method" keys in .metatdata, the function names are assumed to be "set" and "call".

The setter and method functions are required to have the following signatures (as appropriate for each language):

set(path, value) where path is a string denoting the path of the parameter and value is the new value to set. If a custom setter is specified, the function may not need the path argument but it is still required in the signature. Also, the system will guarantee that value is the correct type and value range as described in the descriptor so the script author doesn't need to do type or range checking.

call(path, args) where path is a string denoting the path of the command and args is a map (or specific language equivalent) of named arguments for the command.

So, the outline of a script to handle the *program.json* file above would be (example in JavaScript):

```javascript
function setLED(path, value) {
    // calculate index from path
    // set the LED to new value
    // update the system with the new value
}

function set(path, value) {
    switch (path)
    {
    case "power":
        // set the power to the new value
        // update the system with the new value
        break;
```

```
        }
}

function call(path, args) {
    let value;
    switch (path)
    {
    case "setAllLeds":
        value = args.get("value");
        for (let i=0; i<5; i++) {
            // set led[i] to new value
            // update the system with the new value
        }
        break;
    }
}

// function must have arguments of path and args even though they aren't needed (language
dependent)
function cyclePower(path, args) {
    // cycle the power state
    // update the system with the new value
}
```

Notice that in the example above there is a comment to "update the system with the new value". This requires a mechanism for the script to call into the system. You will also need a similar capability if we want to generate events from script. To facilitate this functionality, a new object has been added to the context global variable in the script called **export**. The export object contains two methods that allow the script act on the virtual device that it has created.

export.update()

The *export.update()* method allows a script to update the value of a parameter. When updated, any listeners to that parameter will automatically be updated with the new value. The update method takes two required arguments and an optional third argument.

1. The first argument is the path of the parameter that is being updated. In the simple case, this is just the name of the parameter. If the device descriptor contains objects and arrays, this path argument is a slash ('/') separated list of nodes in the descriptor tree.
2. The second argument is the new value of the parameter.
3. The third (optional) argument is the normalized value of the parameter. If no normalized value is given, the system will automatically calculate a normalized value using linear interpolation between the minimum and maximum of the parameter.

In JavaScript, our example would read:

```
let export = context.export  // get the export API from context
```

```
function setLED(path, value) {
    // calculate index from path
    // set the LED to new value
    export.update(path, value)  // update the system with the new value
}
```

export.dispatch()

The *export.dispatch()* is similar to the export.update() method except that it generates events instead of parameter changes.

The dispatch method has one required argument which is the path of the event in the descriptor. The dispatch method also takes an optional argument which is a map of name/value pairs representing the arguments of the event that will be generated. The map can be empty, null, or omitted completely for events without arguments.

In Groovy, our example would read:

```
export = context.export

function goodbye(name) {
    // generate a goodbye event
    export.dispatch("goodbye", {"who" : name})
}
```

Python Virtualization / PIP Installation

Third-party Python modules can be installed to MUSE using a *requirements.txt* file, which lists out the module (or modules) to download & install. **Note:** MUSE requires Internet access to download modules.

Method 1

In your program, create a text file named *requirements.txt* and enter your list of modules:
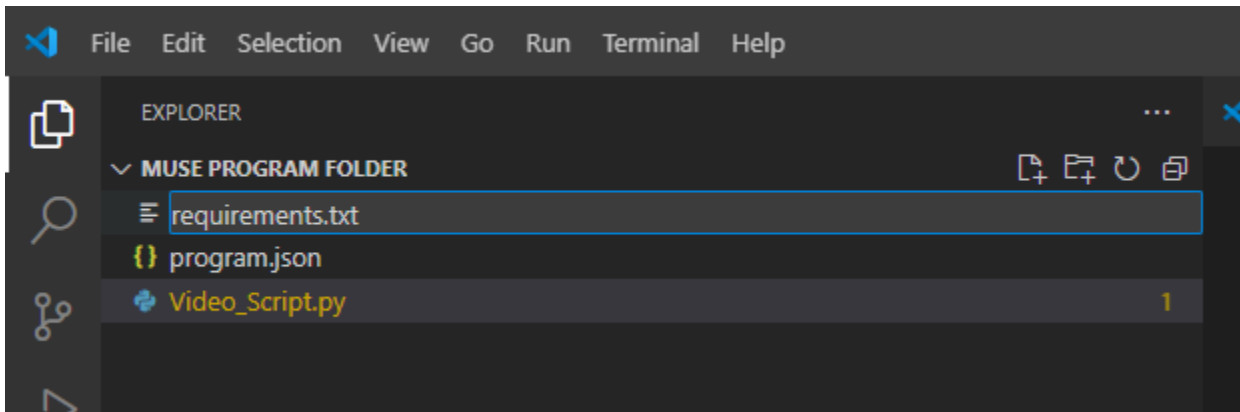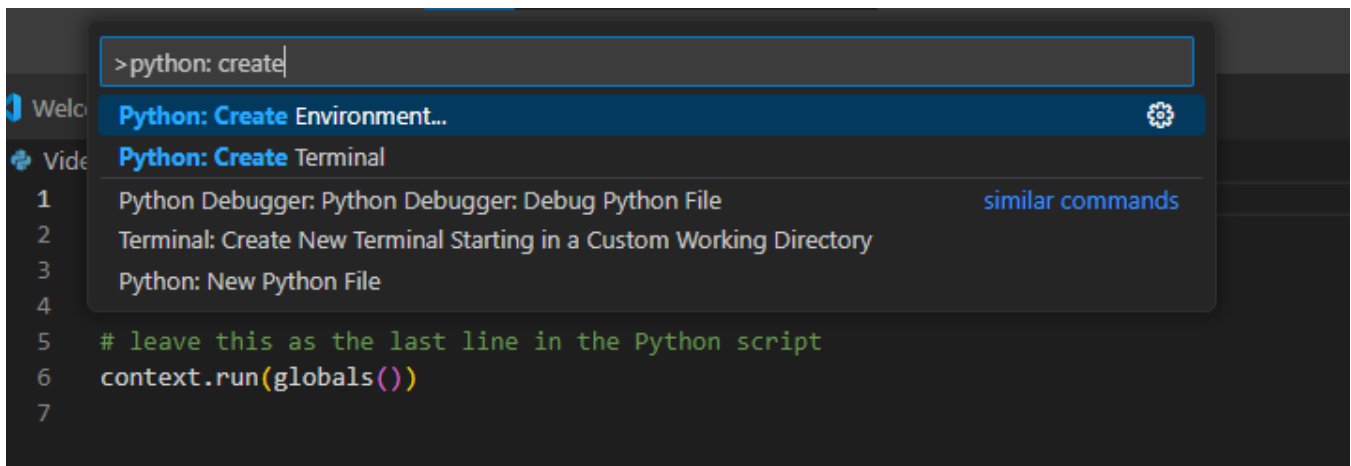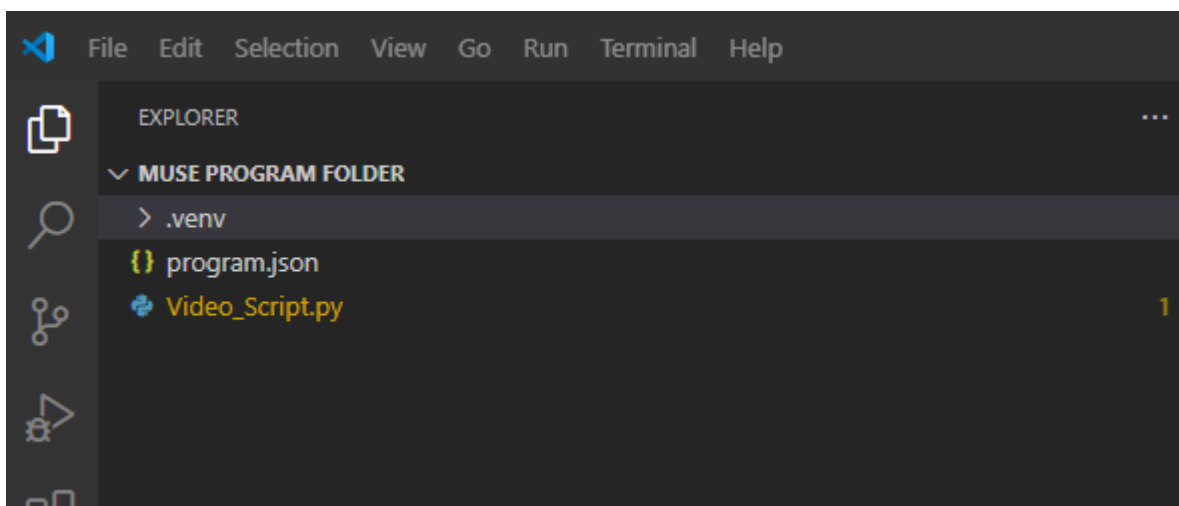

Type in the required module (or modules)
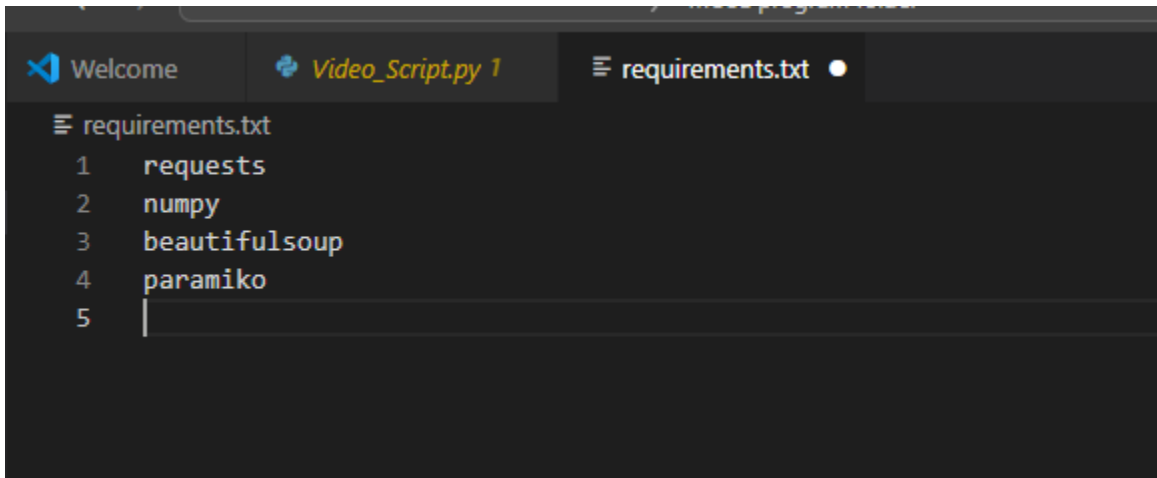
The *requirements.txt* file should then be loaded with your scripts. It will download modules into the code.
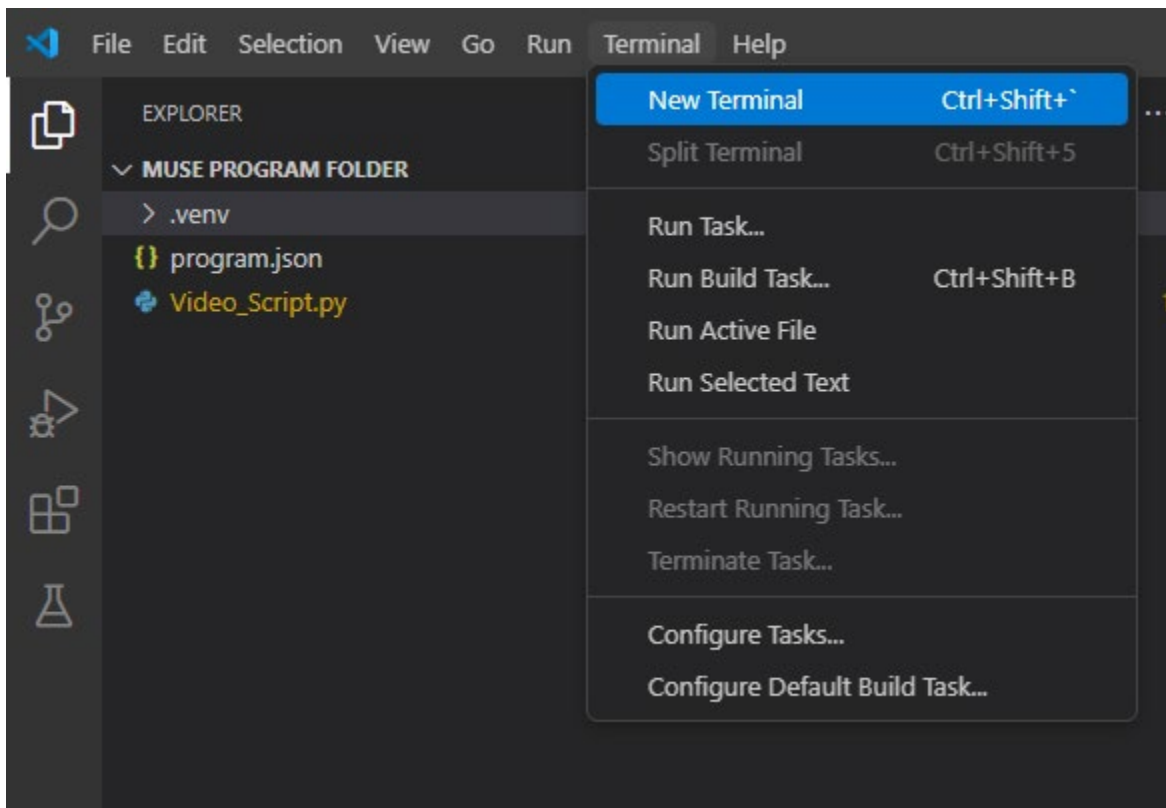
Method 2

Run a new command named Python: Create Environment.

Select VENV and find your Python executable on your machine. This will create a *.venv* folder within your program.
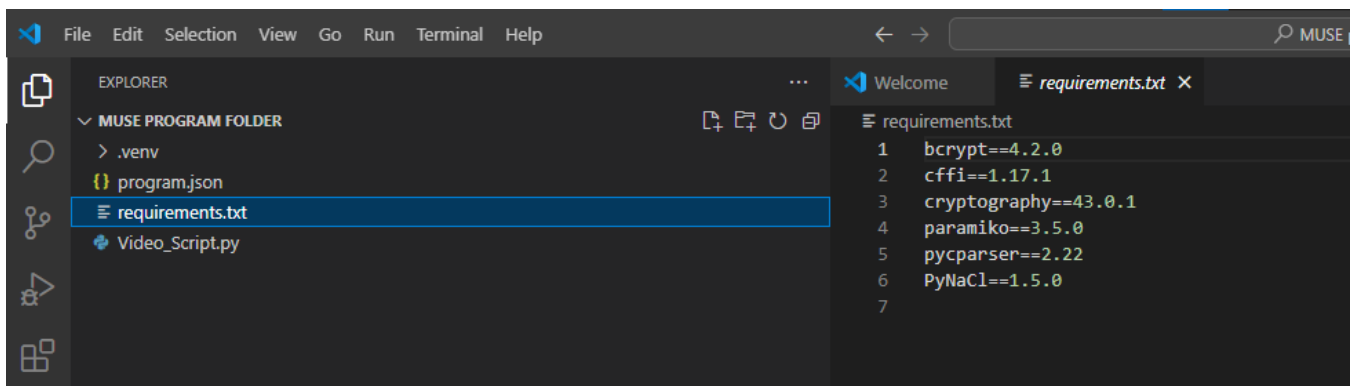

On the menu, select Terminal >> New Terminal.

type pip install [module]. Repeat for each required module.

```
pip install paramiko
```

Then freeze the module to a *requirements.txt* file:

```
pip freeze > requirements.txt
```

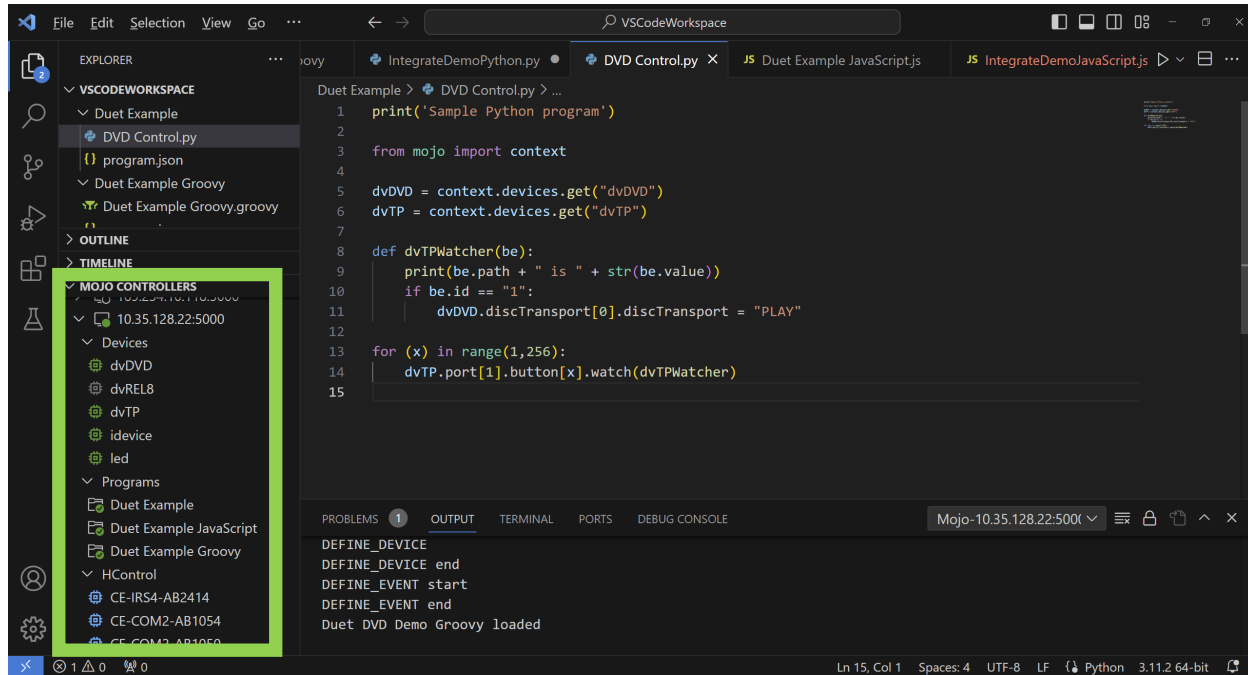This will build a *requirements.txt* and list the modules with versions:

The *requirements.txt* file should then be loaded with your scripts.

Debugging

There are two places that contain substantial debugging information for MUSE. VS Code and the CLI.

VS Code

A great deal of information can be obtained from the MOJO CONTROLLER section that was added by the Mojo VS Code extension.

We can see that we are connected to and authenticated into a controller at 10.35.128.22. The green pip indicates the state of the connection.   A yellow pip indicates the controller has been seen on the network, but authentication has not taken place. No pip means the status is unknown. To attempt a connection, use the refresh icon to initiate it. Follow any login prompts or messages that appear.

Once connected, we can see:

- Currently configured devices and their status
- Currently running programs
- HControl devices discovered near the controller

From the Devices portion of the tree, we can remove devices or ask for the descriptor. Each of these activities is available by context menu by right-clicking on the device in question

From the Programs portion of the tree, you can:

- Retrieve the script from the controller
- Restart the script
- Delete the script
- Enable/disable the script
- Attach to / Detach from the standard output of a script (print( ) and log( ) statements go here)

Attaching is the primary means of debugging a particular script. The script author can place print( ) or log( ) statements throughout the code to track the progress and expected code flow. These print statements can contain information about variable states and other helpful information.

Right-click on the running script and choose Attach to see the standard output.

Groovy syntax example:

```groovy
println('DEFINE_DEVICE')

dvTP = context.devices.get("dvTP");
dvDVD = context.devices.get("dvDVD")

println('DEFINE_DEVICE end')

println('DEFINE_EVENT start')

dvTP.port[1].button[17].watch( {
    if(it.value) { //PUSH:
        dvDVD.discTransport[0].discTransport = "PLAY"
    }
})

dvTP.port[1].button[18].watch( {
    if(it.value) { //PUSH:
        dvDVD.discTransport[0].discTransport = "STOP";
    }
})

println("Duet DVD Demo Groovy loaded")
```

…would produce the following output when restarted:

```
Program "Duet Example Groovy" attached, you will see outputs here.
DEFINE_DEVICE
DEFINE_DEVICE end
DEFINE_EVENT start
Duet DVD Demo Groovy loaded
```

Python syntax example:

```python
print('Duet Example Python')

from mojo import context

print('Grab device objects')
dvDVD = context.devices.get("dvDVD")
dvTP = context.devices.get("dvTP")
```

```python
print('Define my button callback')
def dvTPWatcher(be):
    print(be.path + " is " + str(be.value))
    if be.id == "1":
        dvDVD.discTransport[0].discTransport = "PLAY"

print('set a watch for my buttons')
for (x) in range(1,256):
    dvTP.port[1].button[x].watch(dvTPWatcher)

print('Duet Example Python script loaded')
```

…would produce the following output when run.

```
Duet Example Python
Grab device objects
Define my button callback
set a watch for my buttons
Duet Example Python script loaded
```

JavaScript syntax example

```javascript
context.log('JavaScriptDemo START')

context.log('DEFINE_VARIABLE')

var dvTP;
var dvDVD;

function dvTP_button_event(event) {
    context.log('dvTP_button_event(event) called')
    context.log('event:'+event.path+" is "+event.newValue)

    if(event.newValue){ // PUSH:
        var button
        button = parseInt(event.id)
        switch(button) {
            case 9:
              dvDVD.discTransport[0].discTransport = "PLAY";
              break;
            case 10:
                dvDVD.discTransport[0].discTransport = "STOP";
              break;
            default:
              console.log("unhandled button event "+event.id)
          }
```

```
    }
}
context.log('DEFINE_DEVICE')

dvTP  = context.devices.get("dvTP")
dvDVD = context.devices.get("dvDVD")

context.log('DEFINE_EVENT')

for(let i=1;i<=255;i++)
{
    // set watches on all dvTP buttons
    dvTP.port[1].button[i].watch(dvTP_button_event);
}

context.log('Duet Example JavaScript loaded')
```

would produce the following output:

```
Program "Duet Example JavaScript" attached, you will see outputs here.
[INFO] JavaScriptDemo START
[INFO] DEFINE_VARIABLE
[INFO] DEFINE_DEVICE
[INFO] DEFINE_EVENT
[INFO] Duet Example JavaScript loaded
```

CLI (Command Line Interface)

A far more powerful interface for debugging is the CLI.   This is available from SSH, the Virtual COM port available on the USB-C connector, and the Diagnostics → Shell portion of the MUSE web server.

As discussed earlier in this document, device parameters can be retrieved and set, commands sent, devices listed, and dozens of other useful functions performed.

For the entire list of commands, type *help* in the CLI. This will display a document containing all the published commands. Hit CTRL-C to exit this document.

For details on a specific command, use the *man or help* command.   For example:

```
admin@mojo()> man network:ip
DESCRIPTION
        network:ip


    Set and get network config


SYNTAX
        network:ip [options]


OPTIONS
        -e, --ethinterface
                Set ethernet interface specifier [eth0 or eth1] (defaults to eth0)
                (defaults to eth0)
        --help
                Display this help message
        -m, --mode
                Set mode: dhcp: static: ipv6d: ipv6s. {DHCP | Static | Dynamic Ipv6 or
Static Ipv6 [Eth1 can only be static] (defaults to empty)


                (defaults to )
        -s, --subnet
                Set subnet mask [Valid only when mode is static] (defaults to empty)


                (defaults to )
        -g, --gateway
                Set gateway [Valid only when mode is static] (defaults to empty)


                (defaults to )
        -i, --ip
                Set IP address [Valid only when mode is static] (defaults to empty)


                (defaults to )
        -en, --enable
                Enable eth1 [When enabling eth1 ip, subnet, and gateway values must be
entered.]


        -d, --disable
```

```
          Disable eth1




admin@mojo()>
```

…displays all the syntax and option information for retrieving network information about the MUSE controller.

For debugging, the most useful command in the CLI is *log:tail* .   This will show any exceptions that occur, any debug information that has been printed to the log, and other real-time data that is useful.

With our Tascam DVD module running, log:tail will show transactions to and from the serial port attached to the DVD.   This behavior is defined by the Duet module itself.

```
00:11:10.169 WARN  | TimerDaemon | Duet:TASCAMDVD01UDiscDevice | 190 |
TASCAMDVD01UDiscDevice:  Comm ----------> device: BUFFER
'023E504C59634657442020202020313703'
00:11:10.262 WARN  | Thread-61 | Duet:TASCAMDVD01UDiscDevice | 190 |
TASCAMDVD01UDiscDevice:  Comm <---------- device: INCOMING DATA
'023E504C59734657442020202020323703'
```

In this case, we can see a transaction that occurred when the module decided to poll the DVD player for status. In this case, the module author decided to print the string, which is a mix of printable and non-printable ASCII characters, as an ASCII stream of the hexadecimal values of each character. Knowing this, we can see that the conversation was:

Controller to DVD: STX>PLYcFWD   17ETX   (please start playing the disc)

DVD to Controller: STX >PLYsFWD   27 ETX   (play command acknowledged)

Appendix A: The Thing API

The Mojo Universal Scripting Engine (MUSE) contains a framework for communicating with AMX devices, modules, and other AMX specific facilities: the Thing API.

Context is the handle for the Mojo engine. It has logging and device access features. The structure of context is:

```
context
• devices
    • get(name)      – get a specific device by its name
    • has(name)      – check if a specific device is defined
    • ids()          – get the list of defined devices
• log
    • level            – set/get the current logging threshold
    • trace(msg)    – issue a log message at TRACE level
    • debug(msg)    – issue a log message at DEBUG level
    • info(msg)     – issue a log message at INFO level
    • warning(msg) – issue a log message at WARNING level
    • error(msg)    – issue a log message at ERROR level
• services
    • get(serviceName) – get a service by name
• run(globals())  - a call specific to Python to continue operation once
                the script has reached its end
```

Syntax Examples: (identical for each language unless noted)

Get the device named AMX-10001 and assign it to the script variable dvTP:

```
dvTP = context.devices.get("AMX-10001")
```

Check if a device named dvLighting has been defined:

Python:
```
if context.devices.has("dvLighting"):
     # do something
```

Groovy/JavaScript:
```
if(context.device.has("dvLighting") {
     // do something
}
```

Send a message to the logger at the WARNING threshold:

```
context.log.warning("dvLighting is not defined")
```

Get Request an instance of the timeline service that triggers 1/second forever:

```
myTimeline = context.services.get("timeline")
mytimeline.start([1000],false,-1)
```

The services available vary based on what is currently installed/enabled.   At the time of this writing, there are three services: timeline, smtp, & platform.

Note: Support for the netlinxClient servicehas been deprecated and replaced by the Netlinx Driver. It may be removed in the future.

For the syntax and usage of the services, use the following CLI command:

```
doc:list
```
…to obtain the current list of services available and:

```
doc:show serviceName
```
…to show the syntax and usage description.

The Event structure

Any callback or lambda functions for a .listen passed the event structure. This contains the specific information that triggered the event.

Field	Data type	Description
path	String	The property of the device that this event refers to
id	String	A shortened version path. For ICSP, only the button number is conveyed
arguments	array	The data payload of the event, dependent on the specific event
arguments	array	The data payload of the event, dependent on the specific event
oldValue	variant	The data value before the event was processed
source	Object Ref	The object reference for the specific parameter that was updated

The most commonly encountered event within MUSE is for receiving data from a serial port. Here are some language specific examples.

Python:

```python
def dvCOM3_Event(ev):
    print('PY_ev.source    =='+str(ev.source))
    print('PY_ev.path      =='+str(ev.path))
    print('PY_ev.id        =='+str(ev.id))
    print('PY_ev.arguments:')
    print('[\'data\'] =='+str(ev.arguments['data'].decode()))

idevice.serial[3].receive.listen(dvCOM3_Event)
```

…has the following output when a serial message is received:

```
PY_ev.source    ==<mojo.PythonThing object at 0xffff956e9880>
PY_ev.path      ==serial/3/receive
PY_ev.id        ==receive
PY_ev.arguments:
['data'] ==hello from python
```

Groovy:

```
idevice.serial[3].receive.listen({ serialEvent ->
    println("serial data event on COM3")
    println('G_serialEvent.source    =='+serialEvent.source)
    println('G_serialEvent.path      =='+serialEvent.path)
    println("G_serialEvent.id        =="+serialEvent.id)
    println('G_ev.arguments:')
    println('data =='+new String(serialEvent.arguments.get("data"),"UTF-8"))

})
```

…has the following output when a serial message is received:

```
G_serialEvent.source    ==com.amx.mojo.groovy.GroovyThing@1b5a766a
G_serialEvent.path      ==serial/3/receive
G_serialEvent.id        ==receive
G_ev.arguments:
data ==hello from Groovy
```

JavaScript:

```
function dvCOM3_RX(serialEvent){
  context.log("serial data event on COM3")
  context.log('JS_serialEvent.source    =='+serialEvent.source)
  context.log('JS_serialEvent.path      =='+serialEvent.path)
  context.log("JS_serialEvent.id        =="+serialEvent.id)
  context.log('JS_ev.arguments:')
  context.log('data =='+String(serialEvent.arguments.data))
}

idevice.serial[3].receive.listen(dvCOM3_RX)
```

…has the following output when a serial message is received:

```
[INFO] serial data event on COM3
[INFO] JS_serialEvent.source    ==null
[INFO] JS_serialEvent.path      ==serial/3/receive
[INFO] JS_serialEvent.id        ==receive
[INFO] JS_ev.arguments:
[INFO] data ==hello from JavaScript
```

The Parameter Update Structure

Any callback or lambda functions for a .watch( ) attached to a parameter is passed this structure. This contains the specific information that triggered the parameter change event.

Field	Data type	Description
path	String	The specific parameter that has been updated
id	String	The last element of the path
value	variant	The current value of the parameter
newValue	variant	The current value of the parameter
oldValue	variant	The previous value of the parameter
normalized	float	A float value between 0 and 1, inclusive, based on the range of the value
source	Object Ref	The object reference for the specific parameter that was updated

Strangely the most commonly encountered parameter update within the MUSE environment is a button press. Button events can be queried, so are actually parameters.

Groovy:

```groovy
println('Sample Groovy program')

dvTP = context.devices.get("dvTP")

dvTP.port[1].button[2].watch( { parameterChange ->
    println('parameterChange.id        =='+parameterChange.id)
    println('parameterChange.source    =='+parameterChange.source)
    println('parameterChange.path      =='+parameterChange.path)
    println('parameterChange.value     =='+parameterChange.value)
    println('parameterChange.newValue  =='+parameterChange.newValue)
    println('parameterChange.oldValue  =='+parameterChange.oldValue)
    println('parameterChange.normalized=='+parameterChange.normalized)
} )
```

…has the following output when button 2 on port 1 is pressed:

```
parameterChange.id        ==2
parameterChange.source    ==com.amx.mojo.groovy.GroovyThing@36a1b33c
parameterChange.path      ==port/1/button/2
parameterChange.value     ==true
parameterChange.newValue  ==true
parameterChange.oldValue  ==false
parameterChange.normalized==1.0
```

Python:

```python
print('Sample Python program')

from mojo import context

dvTP = context.devices.get("dvTP")

def dvTPWatcher(parameterChange):
    print(parameterChange.path + " is " + str(parameterChange.value))
    print('parameterChange.id        =='+parameterChange.id)
    print('parameterChange.source    =='+str(parameterChange.source))
    print('parameterChange.path      =='+str(parameterChange.path))
    print('parameterChange.value     =='+str(parameterChange.value))
    print('parameterChange.newValue  =='+str(parameterChange.newValue))
    print('parameterChange.oldValue  =='+str(parameterChange.oldValue))
    print('parameterChange.normalized=='+str(parameterChange.normalized))

dvTP.port[1].button[3].watch(dvTPWatcher)

context.run(globals())
```

…has the following output when button 3 on port 1 is pressed:

```
port/1/button/3 is True
parameterChange.id         ==3
parameterChange.source     ==<mojo.PythonThing object at 0xffffb1acc970>
parameterChange.path       ==port/1/button/3
parameterChange.value      ==True
parameterChange.newValue   ==True
parameterChange.oldValue   ==False
parameterChange.normalized==1.0
```

JavaScript:

```javascript
function dvTP_button_change(buttonChange) {
    context.log('dvTP_button_change(buttonChange) called')
    context.log('button:'+buttonChange.path+" is "+ buttonChange.newValue)
    context.log("JS_buttonChange.id        =="+ buttonChange.id)
    context.log('JS_buttonChange.source    =='+ buttonChange.source)
    context.log('JS_buttonChange.path      =='+ buttonChange.path)
    context.log('JS_buttonChange.value     =='+ buttonChange.value)
    context.log('JS_buttonChange.newValue  =='+ buttonChange.newValue)
    context.log('JS_buttonChange.oldValue  =='+ buttonChange.oldValue)
    context.log('JS_buttonChange.normalized=='+ buttonChange.normalized)


idevice.serial[3].receive.listen(dvCOM3_RX)
```

…has the following output when button 1 on port 1 is pressed:

```
[[INFO] dvTP_button_change(buttonChange) called

[INFO] button:port/1/button/1 is true
[INFO] JS_buttonChange.id        ==1
[INFO] JS_buttonChange.source    ==true
[INFO] JS_buttonChange.path      ==port/1/button/1
[INFO] JS_buttonChange.value     ==true
[INFO] JS_buttonChange.newValue  ==true
[INFO] JS_buttonChange.oldValue  ==false
[INFO] JS_buttonChange.normalized==1
```

The Parameter structure – set/get

When manipulating the value of a particular Thing parameter

Field	Data type	Description
value	variant	The new value which cause the parameter change callback
normalized	float	A float value between 0 and 1 inclusive, based on the value's range
min	variant	The minimum value of a numerical parameter
max	variant	The maximum value of a numerical parameter
defaultvalue	variant	The default value of a parameter
type	float	The data type of this specific parameter
enums	enumeration	For an enumeration, the specific data points available

Syntax Examples:

In Python, to retrieve all the properties of the parity setting on COM3:

```python
print("parity value:"       + str(idevice.serial[3].parity.value))
print("parity normalized:"  + str(idevice.serial[3].parity.normalized))
print("parity min:"         + str(idevice.serial[3].parity.min))
print("parity max:"         + str(idevice.serial[3].parity.max))
print("parity defaultvalue:"+ str(idevice.serial[3].parity.defaultvalue))
print("parity type:"        + str(idevice.serial[3].parity.type))
print("parity enums:"       + str(idevice.serial[3].parity.enums))
```

Returns:

```
parity value:NONE
parity normalized:0.0
parity min:0
parity max:4
parity defaultvalue:NONE
parity type:enum
parity enums:['NONE', 'EVEN', 'ODD', 'MARK', 'SPACE']
```

In JavaScript, to retrieve all the properties of the parity setting on COM3:

```javascript
context.log("parity value:"        + idevice.serial[3].parity.value)
context.log("parity normalized:"   + idevice.serial[3].parity.normalized)
context.log("parity min:"          + idevice.serial[3].parity.min)
context.log("parity max:"          + idevice.serial[3].parity.max)
context.log("parity defaultvalue:" + idevice.serial[3].parity.defaultvalue)
context.log("parity type:"         + idevice.serial[3].parity.type)
context.log("parity enums:"        + idevice.serial[3].parity.enums)
```

Returns:

```
[INFO] parity value:NONE
[INFO] parity normalized:0
[INFO] parity min:0
[INFO] parity max:4
[INFO] parity defaultvalue:NONE
[INFO] parity type:enum
[INFO] parity enums:[NONE,EVEN,ODD,MARK,SPACE]
```

In Groovy, to retrieve all the properties of the parity setting on COM3:

```groovy
println("parity value:"        + idevice.serial[3].parity.value)
println("parity normalized:"   + idevice.serial[3].parity.normalized)
println("parity min:"          + idevice.serial[3].parity.min)
println("parity max:"          + idevice.serial[3].parity.max)
println("parity defaultvalue:" + idevice.serial[3].parity.defaultvalue)
println("parity type:"         + idevice.serial[3].parity.type)
println("parity enums:"        + idevice.serial[3].parity.enums)
```

Returns:

```
parity value:NONE
parity normalized:0.0
parity min:0
parity max:4
parity defaultvalue:NONE
parity type:enum
parity enums:[NONE, EVEN, ODD, MARK, SPACE]
```

Appendix B: LDAP Implementation Details

Overview

The process of verifying credentials and obtaining user authorization is designed to support most organizations requirements for 'least privilege'. The account used to search LDAP to provide user objects for authentication never needs access to user information. Authorization lookups are performed as the authenticated user and as such, no elevated permission is required.

Assumptions and Prerequisites

Assumptions made about the LDAP implementation or environment in which the AMX client will participate include:

1. Must support simple authentication.
2. The account setup for a bind DN must have search capability along with the necessary permissions to read the 'uid', 'cn', 'member' and 'objectclass' attributes.
3. When a search is performed to find a DN with the specified user ID, a search must return one and only one object if the user exists. No object will be returned if an account does not exist for that user ID.
4. An account is considered valid if a user can authenticate/bind. No other attributes are considered during the authentication process.
5. AMX LDAP implementation supports both encrypted and un-encrypted connections using SSL.
6. When a person authenticates, that account must have access to all the attributes defined by RFC 2798 with the following exception:
a. User passwords are not necessarily accessible for anything except to perform a bind to the directory (for example, this attribute may not be directly available to the user).
7. The bind DN must have the ability to search for group membership. (This ability is similar to RMS requirements.)
8. When a person authenticates, that account must have access to "cn" attributes for all groups of which it is a member.
9. Group membership for users is defined by the Role assigned to the user. Use GroupOfNames as the objectClass for group mapping. GroupOfUniqueNames is not supported due to ambiguities associated with implementations which use unique IDs appended to membership DNs.
10. When performing searches for group membership, no restrictions exist which would the restrict returning the full list of objects for which the user is a member with the possible exception of reasonable response timeouts. AMX LDAP implementation does not support paged search results.
11. AMX LDAP implementation does not support following referrals.

Example: Setting a User's Access Rights

To give AMX equipment users access rights to the Controller, group memberships for administrators and users are defined by the Role Name setting when establishing Roles. Two records need to be created in the database:

- One that represents users with user management administrative privileges. The factory default settings include an administrator user which includes all administrative privileges.
- Another that represents users with user privileges. The factory default settings include a user which includes Device Management, Firmware Update, Network Management, and Security Control privileges.

NOTE: You can create as many groups as necessary according to your policies, but you should create at least two groups to separate administrators from other users.

IMPORTANT: The common name of the LDAP group on the server must match the name of the Role assigned to the user on the controller.

Administrator Access Example

Administrator Access	
LDAP Server Configuration	**MUSE Controller Configuration**
Example: **dn:** cn=administrator,ou=groups,ou=Dallas, dc=example,dc=com **objectClass:** groupOfNames **objectClass:** top **cn:** All_Permissions **member:** uid=DallasAdminUser1,ou=people, ou=Dallas,dc=example,dc=com **member:** uid=ICSPUser,ou=people, ou=Dallas,dc=example,dc=com	On the Role Security Details page, create a Role with the Administrator groupOfNames cn, or use the existing administrator role. *Example:* **Role Name:** All_Permissions

User Access Example

Administrator Access	
LDAP Server Configuration	**MUSE Controller Configuration**
Example: **dn**: cn=controller01User,ou=groups, ou=Dallas,dc=example,dc=com **objectClass**: groupOfNames **objectClass**: top **cn**: Studio **member**: uid=DallasUser1,ou=people, ou=Dallas,dc=example,dc=com **member**: uid=DallasUser2,ou=people, ou=Dallas,dc=example,dc=com	On the Role Security Details page, create a Role with a name which matches the groupOfNames cn. *Example:* **Role Name:** Studio

NOTE: If the DN of a user is in both the administrator groupOfNames and the user groupOfNames, the administrative privileges take precedence over user privileges.

3000 RESEARCH DRIVE, RICHARDSON, TX 75082 AMX.com | 800.222.0193 | 469.624.8000 | +1.469.624.7400 | fax 469.624.7153

Last Revised: 2024-10-07