# Explanation of kodi Node-RED flow

What should it look like:

Diagram

Description automatically generated

At the left, you’ll see the list of nodes that you can use in your flow.

At the right, you’ll see some context-aware menu’s (come back to that later)

In the middle, the flow.

Like most people, I write flows flowing from left to right (though flows in it used to flow top down, but those were the old days;-))

First, understand that .meta writes out every keypress to MQTT; the topic is determined like this:

meta/<recipe><device-id or URL-device>/<type action><ACTION>

meta/KODI/192.168.0.50:9090/button/CURSOR ENTER

Node-RED interfaces with .meta through this topic only; there is no meaningful payload in the message.

Okay, the flow:

Left, you see the purple box called MQTT-meta. That is where the flow starts.

It is the MQTT-listener, whose listening on 127.0.0.1:1883, which is mosquitto (MQTT) on Brain, where .meta is putting messages to. You can tell this MQTT-node the topic(s) it should listen to by specifying the topic (-mask) you want it to process.

Double click on mqtt-meta will open the properties, you’ll see IP and Topic-mask.

Two lines flow out of mqtt-meta, one to debu.temp which is a debugging node (more on it later) and one to GetActivePlayerid.

The latter is an important **function** that can do things with the message. I’m using it here to save some variables (the topic) and as we need to talk to Kodi, we also save the URL-device.

Then we do something funny: we create a totally new message that asks Kodi if it is playing something at the moment (GetActivePlayerid). That is important for all actions, it makes a difference if you press menu in a list or if you are playing a video (with this info, we can act on buttons “content-aware”.

That message flows into the green/yellow box called “Check if Playing”. This is a “**HTTP requestor node”**. The name already gives away the clue: it makes a TCP-request.

This request obviously needs to go to Kodi, as we want to ask Kodi if something is playing.

That is done by configuring the node with IP-address and port and type of request.   
We’re doing something special here as we have auto-discovery and can support multiple instances at the same time; we configure the node **Dynamically**. For that, we need to add a specific field into the message: **msg.url.**This was done when we created the message for the TCP-requestor node, in “GetActivePlayerID”. SO, we’re going back to look at the node called “GetActivePlayerID”and look at this piece of code:

1. var url = "http://"+ip+":"+port+"/jsonrpc?request="
2. flow.set("url",url); // Save the url we're working on
3. return {
4. url: url,
5. "payload":
6. {"jsonrpc": "2.0",
7. "method": "Player.GetActivePlayers",
8. "id": 99}}

Line 1 Picks up the variable “IP” that was filled with parts of the topic (IP-address of URL-device) and the port number. (Note that I’ve hardcoded the port number, it’s for a bug).

That way, we have var url filled with “http://192.168.0.50:8080/jsonrpc?request=”

Line 2 saves that variable in a variable that is available to the entire flow for reference later.

Line 3 “returns a msg” that will be output of the node; it will be send to the debug node and the HTTP-request node.

Line 4 is important. It creates the field url in the output message and fills it with the var url.

This is the field that instructs the HTTP Request node to contact KODI….

Lines 5, 6 and 7 are the actual payload that will be send to KODI, the request.

Line 8 is a required ID (by KODI), but isn’t really used further.

So, back to “Check if Playing”, the requestor node. This sends the payload it receives to the URL that was established and waits for an answer.

The answer (coming from KODI) is then analyzed by the **function-node** CheckIsPlaying.   
How, isn’t really important, but it outputs a message to the **function-node** Isolate button.

This function node could have been integrated with CheckIsplaying, but they have been developed in different points in time; I may combine them one time.

Isolate button picks up the Topic that was set by .meta from the global variable Topic and parses it to be processed by the **switch** following it.   
**Switch** does what its name implies: it switches one (or more) output paths, depending on some conditions. This is done by comparing the field “msg.button” to several values. This more or less categorizes the actions that need to be taken.

The switch in this case has obviously multiple outputs, each line is handled by a specific **function** for that button. I’ll explain one in detail as their structure is largely the same, then I’ll discuss only the important ones.

Let’s begin with **function** Power, called when “power off” or “power on”(every button that contains “Power” qualifies) is sent.   
It first slices the Button name into two, by picking up only pos 6 till 99 (the end) and makes it lowercase (easier to compare); it’s saved in variable MyPower.

It then checks if MyPower is “off” or on.

If it is on, it will send a message gelling Kodi to display a notification within Kodi (a welcome text); if it is “off”, It sends a message to Kodi to show the shutdown dialog.

The node receiving this message is again a **HTTP Requestor Node,** configured the same was to pick up the URL from the message at runtime. There’s a debug node after it that shows the response from Kodi, but only for debugging purposes

**That’s it…** this was the entire processing from input to output.

# Specific things that happen after the **switch** function (content aware).

For example, in the “cursor” **function,** we get the flow variable “MyPlayerID”. If it is 0, then we know that there’s nothing playing, so a “cursor enter” (ok) will just send “select” to Kodi, while if an item is playing, it will send “Play.pause”.

# Debugging.

We need to have a way to debug; I certainly need it;-).

That was the right part of the window. When you move your cursor to the middle right of the window, an icon with “<“ appears. You can grab it and pull it to the left; to start with, puill it quite a bit to the left, (halfway your screen is fine) .

This window is context-sensitive with various sources of information, but we’re going to use debug. Click the “bug” icon to open the debugging window:

Graphical user interface, application

Description automatically generated

Now this window is open, you can see all logging that is generated by “**debug nodes**”. Also errors during runtime can show up here. You can show the payload of a message, all fields or predefined text-strings, you name it.

With .,eta, I normally clear the window before hitting the recycle bin icon.

That’s it for now, happy experimenting.

P.s. I wrote that I change the port that NEEO-discovered…. .Meta actually discovers the correct service of KOPDI, but somehow I wasn’t able so far to communicate with Kodi over this port. I therefore replace the discovered port by the default webserver-port .That works…. If you haven’t changed the port in KODI…. It’s on my list to fix this, but there are so many issues to look after;-)