ChatGPT

Building a Max for Live MIDI Effect with JavaScript (Step-by-Step Tutorial)

Introduction

In this tutorial, we will create a **Max for Live** (M4L) **MIDI Effect Device** that executes **Ableton Live API** actions based on JSON or Max **dictionary** commands. This device will let us trigger automation tasks – for example, **setting the tempo**, **creating new MIDI clips**, and **adding MIDI notes** – by sending structured commands to a JavaScript script running inside the Max device. We'll go through every step in detail, from project setup to writing the JavaScript (`live_exec.js`) and troubleshooting common errors. This guide assumes you have Ableton Live (Suite or Standard with Max for Live) and some basic familiarity with Live and Max, but no prior Max scripting experience is required.

By the end, you will have a working M4L MIDI effect where you can input a command like a JSON string or a Max dictionary (e.g. `{ "action": "set_tempo", "value": 120 }`) to make Live perform that action (change the BPM to 120 in this case). We'll use clear headings, code examples, and screenshots for clarity. Let's get started!

Prerequisites and Project Setup

Software: Ensure you have **Ableton Live Suite 11+** (which includes Max for Live) or Live with the Max for Live add-on [1] [2]. We will create the device in Ableton Live and edit it with the built-in Max editor.

Project Folder: It's a good idea to organize your device files. Create a new folder on your computer (for example, on your Desktop or in Live's User Library) to hold your Max device and its JavaScript file. We will save the Max device (.amxd file) and the JavaScript file (.js) here. Keeping them in the same folder ensures Max can find the script file [3] [4] and makes the project portable. (By default, Max will look for the `.js` in the device's folder when you first create the `js` object [4], so placing them together avoids search path issues. You can also add custom folders to Max's search path if needed [5] [6], but using the same folder is simplest.)

Creating a New Max MIDI Effect in Ableton Live

Let's create the Max for Live device in Ableton Live:

1. **Add a Max MIDI Effect Device:** In Live, open the **Browser** and navigate to **Max for Live** devices. Under Max for Live, you'll see device templates for *Max Audio Effect*, *Max Instrument*, and *Max MIDI Effect*. **Drag a "Max MIDI Effect" onto a MIDI track** (or onto an empty track area to create a new MIDI track) [7]. This loads a blank Max MIDI device onto the track. We use a MIDI effect because our JavaScript will manipulate MIDI/clips and not audio [8].

2. **Open the Max Editor:** On the device's title bar in Live, click the **edit button** (the small circular icon with lines, looks like a knob or show device parameters button). This launches the Max editor window for the device [7]. You should now see the Max patcher window, which by default contains some comment boxes (like "MIDI from Live," "Build your MIDI effect here," etc.) and objects for handling MIDI in/out.

3. **Clean the Patch (Optional):** In the Max patcher, you can delete the comment boxes or any unnecessary UI elements that came with the blank device (they are just notes). However, **do NOT delete** the objects named `midiin` and `midiout` if they are present. Those objects route MIDI through your device; keeping them ensures that any incoming MIDI notes will pass through to the track's output even if our script is doing other things [9] [10]. In short, leave `midiin` and `midiout` connected so the device doesn't block MIDI by default.

4. **Save the Device:** Before proceeding, save your Max device patch. Go to **File > Save** in the Max editor, or press ⌘**+S / Ctrl+S**. In the save dialog, navigate to your project folder created earlier, and name the device (e.g., **"JSON_Executor.amxd"**). Then click Save. Saving now is important because when we add a `js` object next, Max will use the device's saved location to find the script file.

At this point, we have an empty Max MIDI Effect device ready for customization.

Adding a JavaScript Object (`js`) to the Device

Now we'll add a Max **[js] object**, which is what allows us to run JavaScript code inside the device:

1. **Create the [js] Object:** In the Max patcher, ensure it's in *edit mode* (click the padlock icon to unlock the patch if it's locked). Create a new object box (you can press **N** on your keyboard or click the **Object** icon in the toolbar). In the text box that appears, type:

   ```
   js live_exec.js
   ```

   then press Enter. This creates a `js` object that will load a JavaScript file named `live_exec.js` [11]. (You can name your script file differently, but make sure to use the same name in the object and the file itself. We'll use "live_exec.js" as requested.)

2. **Position and Lock:** Place the `js live_exec.js` object in the patch. It will initially show an error in the Max Console (e.g., **"js: can't find file live_exec.js"** in red) – don't worry! Since we haven't created or saved the actual `live_exec.js` file yet, Max can't find it, hence the error [4]. We'll fix that in the next step. Now lock the patch (click the padlock or press **Ctrl+E / Cmd+E**) to avoid accidental edits.

3. **Open the JS Editor: Double-click** the [js] object. This opens the Max **JavaScript Editor** window, which is a text editor for writing your script. It should open a blank file, and the filename (live_exec.js) is shown at the top.

4. **Save the JavaScript File:** In the JS editor, go to **File > Save**. It will default to the same folder where you saved the .amxd device (the name "live_exec.js" should already be filled in) [12] . Confirm that the save location is your project folder and click **Save**. Now the error "can't find file live_exec.js" will disappear because the file exists in the device's folder [4] . The [js] object is now properly linked to *live_exec.js* in your project folder.

At this stage, our Max patch has a [js] object but nothing connected to it yet. Next, we'll set up inputs to feed commands into this JavaScript, and then write the script logic.

Building the Device's Patch: Inputs and Connections

We want to send two types of data into our `js` object: **dictionary** messages and **raw JSON** strings. We'll set up the patch so we can test both methods.

- **Dictionary Input:** Max's **[dict]** object holds structured data (similar to JSON) that can be sent as a single unit. We'll use a [dict] to compose commands in a convenient way.

- **JSON Text Input:** We'll also allow a raw JSON string to be parsed by our JS, to mimic receiving JSON from an external source or just typing it directly.

Steps to add input objects:

1. **Add a [dict] Object:** Unlock the patch and create a **[dict]** object (press **N** and type `dict commands`). By giving it a name (e.g., "commands"), we can reference this dictionary in JS if needed. This object will store our command in a structured form. Position it to the left of the [js] object.

2. **Add a Bang/Trigger:** To output the dictionary's content on demand, connect a **[button]** (bang) to the inlet of the [dict] object. This way, when we click the button, the [dict] will send out its data. By default, the [dict]'s outlet will send a message of the form `dictionary commands` (i.e., it outputs the dictionary's name and contents) into whatever it's connected to [13] [14] . We will connect it to [js].

3. **Add a JSON Text Input:** For raw JSON, one simple way is to use a **[textedit]** object (a text box). Create a [textedit] (you can find it in the Max object palette or press **N** and type `textedit`). In its **Inspector**, set "Return sends bang" to true (so that pressing Enter will output the text). Alternatively, use a plain **[message]** object to store a JSON string if you prefer (but [textedit] lets you easily type and edit JSON during testing). Label it "JSON Input" for clarity.

4. **Connect Inputs to [js]:** Connect the **outlet of [dict]** to the **inlet** of [js]. Also connect the **outlet of [textedit]** to the **same inlet** of [js]. We are using one inlet for both types of input. The [js] object can distinguish a dictionary message from a regular message internally (Max will call a special function for dictionary inputs, as we'll see). The patch wiring should look like this:

```
[button] bang      [textedit] (JSON input)
    |                   |
```

```
    v                   v
[dict commands] ------> [js live_exec.js]
```

(When the button is clicked, the [dict] outputs a "dictionary" message into [js]. When you press Enter in the textedit, it sends the raw text string into [js].)

1. **MIDI Throughput (if applicable):** Ensure the **[midiin]** object from Live's input is connected to **[midiout]** to Live's output (this was the default in the blank device). If you deleted them earlier by mistake, add them back: create an object `midiin` and one `midiout`, connect `midiin`'s output to `midiout`'s input. This passes incoming MIDI through your device unchanged [9], so using this device on a track won't block MIDI notes when you're not specifically intercepting them.

Now our Max patch setup is complete: we have a [js] to run code, a [dict] for structured input, and a [textedit] for raw JSON input, all wired up. Next, we'll write the JavaScript in *live_exec.js* to handle these inputs and perform Live API actions.

Writing the JavaScript (live_exec.js)

Open the *live_exec.js* file in the Max JS editor (if you closed it, just double-click the [js] object again). We will implement the following in our script:

- `loadbang` **handler:** A function that runs when the device loads, used to initialize and print a confirmation.
- `dictionary` **handler:** A function to receive and handle dictionary inputs from the [dict] object.
- **JSON parsing for raw input:** We'll use a generic handler (the `anything` function) to catch raw JSON strings and parse them.

And of course, functions to perform actions like setting tempo, creating clips, and adding notes via Ableton's Live API.

Let's write the script step by step, explaining each part:

1. Loadbang: Initialization

Max will call a `loadbang()` function in our script when the device is loaded or the script is reset. We can use this to post a message to the console confirming our script is running, and to do any initialization if needed.

```
// live_exec.js

// 1. Called when device loads or script is reset
function loadbang() {
    post("live_exec.js loaded successfully\n");  // This will print in Max
Console
}
```

When you save the script with this function and **then save the Max patch** (or close/reopen it), you should see “live_exec.js loaded successfully” in the Max Console, verifying that the script initialized. (Open the Max Console from Live by clicking the blue circle icon on the device or via **Window > Max Console** in the editor.)

2. Handling Dictionary Input (function dictionary)

Max’s JS API provides a special way to receive dictionaries: if you define a function named `dictionary`, it will be called whenever a dictionary message is sent into the [js] object [15]. The dictionary’s name is passed in as an argument, and we can then use the Max `Dict` class to extract data.

We expect our dictionary to contain an “action” key (plus other keys depending on the action). For example, a dictionary might look like:

```
{
  "action": "set_tempo",
  "value": 130
}
```

or

```
{
  "action": "create_clip",
  "track": 0,
  "scene": 0,
  "length": 4
}
```

or

```
{
  "action": "add_notes",
  "track": 0,
  "scene": 0,
  "notes": [ { "pitch": 60, "start_time": 0, "duration": 1 } ]
}
```

Let’s implement `function dictionary(dictName)` to handle these. We’ll parse the dictionary and then call the appropriate Live API functions based on the `"action"` field. The Ableton Live API is accessible via the `LiveAPI` object in Max JS, which allows us to get/set properties or call functions on Live elements [16]. We will use Live API calls to perform the actions:

- **Set Tempo:** We will get a LiveAPI object for the Song (`live_set`) and set its tempo property.
- **Create Clip:** We will access the specified track’s clip slot and call its `create_clip` function to make a new clip of a given length.

- **Add Notes:** We will get the Clip object and call `add_new_notes` with a notes list.

Here's the code with comments explaining each step:

```
// 2. Handle dictionary input from [dict] object
function dictionary(dictName) {
    // Access the Max dictionary by name
    var data = new Dict(dictName);
    // Read the "action" key from the dictionary
    var action = data.get("action");
    if (!action) {
        // If no action specified, warn and stop
        post("No action specified in dictionary\n");
        return;
    }

    // Switch on the action type
    if (action === "set_tempo") {
        // Get the tempo value
        var newTempo = data.get("value");
        if (newTempo !== null) {
            // Use LiveAPI to set the tempo. "live_set" is the Song (project)
object.
            var song = new LiveAPI("live_set");
            song.set("tempo", newTempo);  // Set tempo in BPM
            post("Tempo set to " + newTempo + " BPM\n");
        } else {
            post("Error: 'value' (tempo) not provided in dictionary\n");
        }

    } else if (action === "create_clip") {
        // Get track and scene indices and clip length
        var trackIndex = data.get("track");
        var sceneIndex = data.get("scene");
        var length = data.get("length");
        if (trackIndex === null || sceneIndex === null || length === null) {
            post("Error: 'track', 'scene', or 'length' missing for
create_clip\n");
            return;
        }
        // Construct the Live API path for the clip slot: live_set tracks
[trackIndex] clip_slots [sceneIndex]
        var clipSlotPath = "live_set tracks " + trackIndex + " clip_slots " +
sceneIndex;
        var clipSlot = new LiveAPI(clipSlotPath);
        // Ensure target is a MIDI track and slot is empty, then create clip
        if (clipSlot.get("has_clip") == 0) {  // has_clip returns 0/1 (false/
```

```
true) 18
            clipSlot.call("create_clip",
length);  // Create a new clip of given length in beats 19 20
            post("Created a new MIDI clip of length " + length + " beats at
track " + trackIndex + ", scene " + sceneIndex + "\n");
        } else {
            post("Note: Clip slot " + trackIndex + ":" + sceneIndex + " already
has a clip, skipped creation\n");
        }

    } else if (action === "add_notes") {
        // Add MIDI notes to an existing clip
        var trackIndex = data.get("track");
        var sceneIndex = data.get("scene");
        var notesList = data.get("notes");  // This might return a JS array of
notes if the 'notes' key is an array
        if (trackIndex === null || sceneIndex === null || notesList === null) {
            post("Error: 'track', 'scene', or 'notes' missing for add_notes\n");
            return;
        }
        // Get the clip object at the specified track/scene
        var clipPath = "live_set tracks " + trackIndex + " clip_slots " +
sceneIndex + " clip";
        var clip = new LiveAPI(clipPath);
        if (clip.id == 0) {  // id 0 means no clip present 21
            post("Error: No clip exists at track " + trackIndex + ", scene " +
sceneIndex + " (add_notes aborted)\n");
            return;
        }
        // Prepare the notes dictionary for Live API. It expects a dict with a
"notes" key containing a list of note dicts.
        // We can call the LiveAPI function add_new_notes directly with a JS
object (Max will convert it).
        try {
            // If notesList is a Max-specific type, convert it to JS array via
JSON:
            var notesArray;
            if (notesList instanceof Atom || notesList instanceof Dict) {
                // To cover cases where 'notes' might be a Dict or Atom, use Max
dictionary JSON:
                var notesJSON = data.stringify();       // get full JSON string
of the dictionary
                var parsed = JSON.parse(notesJSON);     // parse to JS object
                notesArray = parsed.notes;
            } else {
                notesArray = notesList;  // if it's already a JS array
            }
            // Call add_new_notes with an object containing our notes array
```

```
            clip.call("add_new_notes", {notes: notesArray});  // Live API will
add these notes [22] [23]
            post("Added " + notesArray.length + " notes to clip at track " +
trackIndex + ", scene " + sceneIndex + "\n");
        } catch (e) {
            post("Error parsing notes for add_notes: " + e + "\n");
        }

    } else {
        post("Unknown action: " + action + "\n");
    }
}
```

A few things to note in the above code:

- We used `new Dict(dictName)` to interface with the Max dictionary [15]. We could also have converted it to a JSON string using `data.stringify()` and parsed that, but using `get()` for each key is straightforward for simple structures.
- The LiveAPI calls:
- `song.set("tempo", newTempo)` sets the tempo property of the Song. (The Song class property **tempo** is measured in BPM [17].)
- `clipSlot.call("create_clip", length)` creates a new MIDI clip of the given length in beats on the specified clip slot [19] [20]. We check `has_clip` first to avoid errors [18].

- `clip.call("add_new_notes", {notes: notesArray})` adds notes to a clip. The argument is a dictionary with a `"notes"` key whose value is an array of note dictionaries [23]. Each note dictionary needs at least `pitch`, `start_time`, and `duration` keys (and can include velocity, etc.) [24]. We construct the JS object `{notes: notesArray}` which Max's JS will turn into a Max dictionary under the hood and call the Live API. The Live API returns note IDs or errors if something is wrong.

- We handle the case where the clip is missing (no clip to add notes to) and log errors for missing keys.

- Note: The Max `js` environment (as of Max 8) uses an older JS engine (ES5), so we avoid modern syntax and we explicitly create objects like `{notes: notesArray}` instead of using shorthand `notesArray` (the shorthand can sometimes cause issues) [25].

3. Handling Raw JSON Input (function anything)

Now we'll write a handler for raw JSON strings. In Max, if a message comes into [js] that isn't a known message (like not a number, not a list, not a dictionary, and doesn't match a specific function name), it triggers the `anything` function in the script. The `anything` function receives the *message name* and arguments as parameters. For example, if a symbol or a list comes in, `messagename` holds the first token and `arguments` holds the rest.

When we send a raw JSON string via [textedit], one of two scenarios happens: - If the JSON string contains **no spaces**, [textedit] will output it as a single symbol (the entire JSON is one token). In that case, `messagename` will literally be the JSON text (e.g. `{"action":"set_tempo","value":120}`) and there will be no arguments. - If the JSON string contains spaces (pretty formatting), it will be split into multiple tokens. For simplicity, we'll assume or enforce that the JSON is a single-string without spaces or we'll reconstruct it.

To handle both cases, our `anything` function will rebuild the message into one string and then attempt to `JSON.parse` it. Then it can reuse the same logic as above to execute the command. We might factor out the action-handling code into a helper (to avoid duplicating code), but for clarity we can also just replicate the action checks.

```
// 3. Handle raw JSON string input from [textedit] or message
function anything() {
    // Reconstruct the incoming message tokens into one string:
    var jsonStr = messagename;
    // If there are additional arguments (in case the JSON had spaces)
    for (var i = 0; i < arguments.length; i++) {
        // Append each argument separated by space
        jsonStr += " " + arguments[i];
    }
    // Now jsonStr should contain the full JSON string
    // Remove any leading/trailing quotes if present
    if (jsonStr.charAt(0) === '"' && jsonStr.charAt(jsonStr.length-1) === '"') {
        jsonStr = jsonStr.substring(1, jsonStr.length-1);
    }
    // Try to parse JSON
    var cmd;
    try {
        cmd = JSON.parse(jsonStr);
    } catch (e) {
        post("JSON parse error: " + e.message + "\n");
        return;
    }
    // Now 'cmd' is a JavaScript object. We expect it to have an 'action'.
    if (!cmd.action) {
        post("No action specified in JSON\n");
        return;
    }
    // Perform the action (same as above logic)
    if (cmd.action === "set_tempo") {
        if (cmd.value !== undefined) {
            var song = new LiveAPI("live_set");
            song.set("tempo", cmd.value);
            post("Tempo set to " + cmd.value + " BPM\n");
        } else {
```

```
            post("Error: 'value' not provided for set_tempo\n");
        }
    } else if (cmd.action === "create_clip") {
        var trackIndex = cmd.track, sceneIndex = cmd.scene, length = cmd.length;
        if (trackIndex === undefined || sceneIndex === undefined || length ===
undefined) {
            post("Error: track, scene, or length missing for create_clip\n");
            return;
        }
        var clipSlot = new LiveAPI("live_set tracks " + trackIndex + "
clip_slots " + sceneIndex);
        if (clipSlot.get("has_clip") == 0) {
            clipSlot.call("create_clip", length);
            post("Created a new MIDI clip of length " + length + " beats at
track " + trackIndex + ", scene " + sceneIndex + "\n");
        } else {
            post("Clip slot already occupied, no new clip created\n");
        }
    } else if (cmd.action === "add_notes") {
        var trackIndex = cmd.track, sceneIndex = cmd.scene, notesArray =
cmd.notes;
        if (trackIndex === undefined || sceneIndex === undefined || !
Array.isArray(notesArray)) {
            post("Error: track, scene, or notes missing/invalid for
add_notes\n");
            return;
        }
        var clip = new LiveAPI("live_set tracks " + trackIndex + " clip_slots "
+ sceneIndex + " clip");
        if (clip.id == 0) {
            post("Error: No clip at track " + trackIndex + ", scene " +
sceneIndex + " to add notes\n");
            return;
        }
        clip.call("add_new_notes", {notes: notesArray});
        post("Added " + notesArray.length + " notes to clip at track " +
trackIndex + ", scene " + sceneIndex + "\n");
    } else {
        post("Unknown action: " + cmd.action + "\n");
    }
}
```

A few notes on the `anything` implementation: - We concatenated `messagename` and `arguments` to reconstruct the JSON. For example, if [textedit] output the string as separate tokens, this will merge them back with spaces. - We strip quotes from the ends of the string in case the JSON came in quoted. (If you use a [message] object with a typed JSON in quotes, you might get quotes in the output.) - We used

`JSON.parse` to get a JavaScript object (`cmd`). If the JSON is malformed, we catch the error and report it. - Then we perform the same checks and actions as in the dictionary case.

By handling both dictionary input and raw JSON, we've made our script flexible. In practice, you might decide to use one method consistently (dictionary is convenient within Max; JSON might be used if receiving from external sources or just for copy-paste ease).

Important: After writing or editing the JavaScript, **save the JS file (⌘/Ctrl+S)**. Every time you save the JS, Max will automatically reload and run it (triggering `loadbang` again). Check the Max Console for any syntax errors (they will show up in red). Fix any typos or missing commas indicated by error messages before proceeding.

Testing the Device

With our device patch and script in place, it's time to test the functionality. We will try both dictionary-based commands and raw JSON:

Testing with Max Dictionary Input

1. **Open the dict editor:** In the Max patch, double-click the **[dict commands]** object. A dictionary editor window appears. You can type JSON-like content here (Max's dictionary uses JSON syntax for editing [26]).

2. **Set Tempo via dictionary:** For example, enter the following in the dict editor:

   ```
   {
     "action": "set_tempo",
     "value": 90
   }
   ```

   (This will set Live's tempo to 90 BPM.) Close the editor (or hit **Ctrl+Enter** inside it to apply changes). Now click the **[button]** connected to [dict]. This sends the dictionary to the [js]. In the Max Console, you should see a message "Tempo set to 90 BPM", and in Ableton Live the Tempo (top-left of Live's window) should change to **90**.

3. **Create a Clip via dictionary:** Edit the dictionary to:

   ```
   {
     "action": "create_clip",
     "track": 0,
     "scene": 0,
     "length": 4
   }
   ```

This intends to create a 4-beat clip in track index 0 (the first track) at scene index 0 (the first scene). Before clicking the button, make sure **Track 1 in Ableton Live is a MIDI track**, since `create_clip` only works on MIDI tracks (audio tracks can only have audio clips) [27]. If your first track is an Audio track, either change the track number to a MIDI track or convert the first track to MIDI. Now click **[button]** to send the command. In the Max Console you should see "Created a new MIDI clip…", or if a clip was already there you'll see the message we coded for that case. Over in Live, you should see a new empty MIDI clip appear in the first track, first clip slot (Session View). It will be 4 beats long (one bar if in 4/4 time). *Tip: if you don't see it, ensure you're in Session View and not Arrangement, and scroll to the track/scene in question.*

4. **Add Notes via dictionary:** Now let's add a MIDI note. Enter this into the dictionary:

```
{
  "action": "add_notes",
  "track": 0,
  "scene": 0,
  "notes": [ { "pitch": 60, "start_time": 0, "duration": 1 } ]
}
```

This will add a note to the clip we just created. The note is MIDI pitch 60 (Middle C, C3 in Ableton's naming) [28], starting at beat 0 (beginning of the clip) and lasting 1 beat. Click the **[button]** to send it. In the Max Console you should see "Added 1 notes to clip at track 0, scene 0" (or similar), and in Live's Session View, double-click the clip we created (track 1, scene 1) to open the Clip view. You should see a MIDI note in the clip starting at the beginning and one beat long. For example, here's a screenshot of a clip with a single C3 note added by the script (the note in the piano roll corresponds to pitch 60, and spans 1 beat from 0.0 to 1.0).

If the note doesn't appear, make sure the clip exists (the add_notes command will fail if no clip is present). Also verify the track/scene indices are correct. Index 0 refers to the first track or scene; if you intended a different slot, adjust accordingly.

1. **Add multiple notes:** You can add multiple notes at once by including more objects in the "notes" array. For instance:

```
{
  "action": "add_notes",
  "track": 0,
  "scene": 0,
  "notes": [
    { "pitch": 60, "start_time": 0, "duration": 1 },
    { "pitch": 64, "start_time": 2, "duration": 1 }
  ]
}
```

This would add two notes (C3 at beat 0, and E3 at beat 2, each 1 beat long). Sending that dictionary would result in two notes in the clip (you can try this and observe the clip contents update). The `add_new_notes` function can add many notes in one call [29] [30] – the Live API will handle inserting them all at once.

Testing with Raw JSON Strings

Now let's test sending commands as raw JSON text, using the [textedit] we set up:

1. **Set Tempo (JSON):** Click the [textedit] box and type a JSON string, for example:

   ```
   {"action":"set_tempo","value":120}
   ```

   (Notice we keep it in one line without spaces to avoid any tokenization issues in Max.) Then press **Enter** (or click outside the box). The textedit will send the string to [js]. Our `anything()` function will parse it and perform the action. You should see "Tempo set to 120 BPM" in the console and Live's tempo should change to 120.

If your JSON had spaces or line breaks, our code attempts to reconstruct it. For example, you could also try a pretty-printed JSON:

```
{
  "action": "set_tempo",
  "value": 133
}
```

It might be sent as separate pieces, but the script will join them. Check that the tempo changes to 133. If you see a "JSON parse error" in the Max Console, then the text might not have been reconstructed properly – in that case, try removing unnecessary spaces or use the one-liner format. (Our simple approach should handle most cases, but very complex or multiline input might require a more robust assembly of tokens.)

1. **Create Clip (JSON):** Similarly, test creating a clip with a JSON string:

   ```
   {"action":"create_clip","track":0,"scene":1,"length":8}
   ```

   This would create an 8-beat clip in track 0, scene 1 (i.e., first track, second scene). Try it and verify in Live that the clip appears in the specified slot. The console should confirm the creation. (If that scene already had a clip, you'll get the "already has a clip" message instead.)

2. **Add Notes (JSON):** Finally, test adding notes with raw JSON. Assuming you just created a clip in track 0 scene 1, send:

```
{"action":"add_notes","track":0,"scene":1,
 "notes":[ {"pitch":72,"start_time":0,"duration":0.5},
           {"pitch":74,"start_time":1,"duration":0.5} ]}
```

This should add two notes (C5 and D5 if 72 and 74, starting at beat 0 and beat 1 respectively, half-beat duration). The console should say 2 notes added, and you can open that clip in Live to see them.

Our device is now responding to both dictionary inputs and raw JSON text to control Ableton Live!

Troubleshooting Tips

Even with careful setup, you might run into some common issues. Here are some troubleshooting tips for errors and unexpected behavior:

- **"js: can't find file *live_exec.js*" error:** This means the [js] object could not locate the script file. To fix this, ensure you saved *live_exec.js* in the **same folder** as the .amxd device [12] [4]. If the device isn't saved yet, save it and then re-save the JS. Max looks in the device's directory by default for the script [4]. Alternatively, add your folder to Max's File Preferences (Options > File Preferences) [6] or give [js] an absolute path, but keeping them together is easiest.

- **No output or action not happening:** Check the **Max Console** for any errors. If there's a JavaScript syntax error, the console will show it and the script may not function at all. Fix any errors indicated (line numbers are usually given for script errors). If there are no errors but nothing happens, add some `post()` statements in your functions to trace execution. For example, post the raw JSON string you got in `anything()` to ensure it's what you expect.

- **"No action specified" warnings:** If our script prints "No action specified in dictionary/JSON," then the input didn't have an `"action"` key at top level, so the script didn't know what to do. Ensure your JSON/dict is formatted as we described, with an `"action"` field spelled correctly. It's case-sensitive ("set_tempo" is not the same as "SetTempo").

- **"Unknown action: X":** This means the `"action"` key was received, but we didn't handle that value in the script. Check for typos in the action name or add handling for the new action if you intended a new feature.

- **"Error: … missing for …":** These are messages we programmed to appear if required keys are missing. Double-check your input structure. For example, if you use `"create_clip"`, you must provide track, scene, and length. For `"add_notes"`, ensure **notes** is an array of note objects (and that the keys inside each note are spelled correctly – e.g., `start_time` not `startTime`, etc., and remember Max dictionary doesn't support `true/false` booleans, use 0/1 or omit optional keys).

- **Dictionary structure issues:** Max dictionaries can be a bit tricky with nested arrays/dicts. If you find your [dict] isn't outputting the notes list correctly, one approach is to construct the dictionary via messages or use `dict.pack` messages. In our JS, we attempted to handle both pure JS arrays and Max dictionaries. Using JSON text inside the [dict] editor (as we did) is usually fine. If you attempt to

build a dictionary via messages, note that the `dict` **object uses the message** `append` **or** `set` to add data, and adding a list of dictionaries requires careful use of those (which can be cumbersome). In such cases, it might actually be easier to just send a JSON string to the JS for complex structures. If needed, refer to the Cycling '74 docs on dictionaries for how to format nested data [14] [31] or the forum discussions on formatting `add_new_notes` dictionaries (where they wrap the list in another dict as a workaround) [32] [33].

- **Live API errors:** Some errors may come not from our code, but from Ableton Live if, for example, an operation is invalid. These errors also appear in the Max Console. For instance, if you try to create a clip on an audio track, Live will log an error *"Can't create clip in audio track"*. Or if you try to add notes that overlap or are out of clip range, it might not add them. Ensure the context is correct (MIDI track for MIDI clips, etc.). The official Ableton **Live Object Model (LOM) documentation** is very helpful to understand what functions and properties are available and their constraints [20] [23].

- **Indices and IDs:** Remember that tracks and scenes are zero-indexed in the API (0 is first). If you use track 0 and scene 0 in commands but your Live set's first track is an Audio track, you might be unintentionally targeting an audio track. Track 0 means the leftmost track in the Session (excluding the Master track). If you need to target a different track, adjust the index. Similarly, if you want to target a specific clip slot by its **clip ID** (which LiveAPI can also use), that's more advanced – sticking to the path strings as we did is simpler for now.

- **Persistence:** If you save and close your Live set, the M4L device (and the dict content) may reset when reopened (the device will reload fresh). If you need to preserve dictionary content between sessions, consider adding a *preset* or using the **pattr** system or even writing to a file. But that's beyond this tutorial's scope.

Finally, if you need more information on the Max objects and the Live API: - Consult the official **Cycling '74 Max documentation** on the `[js]` object and JS usage (e.g., the **JavaScript in Max** reference, and the **Dict** reference) [14] [31]. - Check out the **Ableton Live Object Model reference** on Cycling '74's site for details on classes like Song, Track, ClipSlot, Clip, etc. (We cited a few relevant sections above, such as the `create_clip` function of ClipSlot [20] and the `add_new_notes` function of Clip [23], which lists all the note properties you can use, like velocity, mute, etc.) - The **Max for Live Community** (Cycling '74 forums, Ableton forums, etc.) is a great place to search for examples. The tutorial series by Adam Murray is an excellent resource for JavaScript in Max for Live [34] [35], and much of what we implemented here is in line with those best practices.

Congratulations on creating a powerful Max for Live device! You can now extend this framework by adding more commands and corresponding script functions. For example, you could implement actions to arm tracks, start/stop clips (`clipSlot.call("fire")` to launch a clip [36]), change device parameters, etc. Just be sure to consult the Live API documentation for the correct function or property names. Happy hacking!

---

[1] [3] [4] [7] [8] [9] [10] [11] [12] JavaScript in Ableton Live: Getting Started - Adam Murray's Blog
https://adammurray.link/max-for-live/js-in-live/getting-started/

2 34 35 JavaScript in Ableton Live Overview - Adam Murray's Blog
https://adammurray.link/max-for-live/js-in-live/

5 6 javascript - Where do I put .js files in Max/MSP? - Stack Overflow
https://stackoverflow.com/questions/30213069/where-do-i-put-js-files-in-max-msp

13 14 15 26 31 Dictionaries | Cycling '74 Documentation
https://docs.cycling74.com/userguide/dictionaries/

16 Max JS API | Cycling '74 Documentation
https://docs.cycling74.com/apiref/js/

17 Song - Live Object Model | Cycling '74 Documentation
https://docs.cycling74.com/apiref/lom/song/

18 19 22 25 27 28 29 30 JavaScript in Ableton Live: Generating MIDI Clips - Adam Murray's Blog
https://adammurray.link/max-for-live/js-in-live/generating-midi-clips/

20 21 36 ClipSlot - Live Object Model | Cycling '74 Documentation
https://docs.cycling74.com/apiref/lom/clipslot/

23 24 Clip - Live Object Model | Cycling '74 Documentation
https://docs.cycling74.com/apiref/lom/clip/

32 33 Javascript dictionary and Live API issues - Javascript Forum | Cycling '74
https://cycling74.com/forums/javascript-dictionary-and-live-api-issues