

AudioFuse HTTP API — Developer Documentation

This documentation set covers everything required to build a third-party client for the AudioFuse HTTP API exposed by **Arturia AudioFuse Control Center**.

The API lets external software read and modify the state of a connected AudioFuse interface — gain, monitoring, clock source, presets, and so on — over standard HTTP, with real-time updates over Server-Sent Events (SSE).

Audience and prerequisites

This documentation assumes you are a developer who is already familiar with:

- HTTP semantics (methods, status codes, headers, JSON bodies)
- JSON request and response formatting
- DNS-SD / Bonjour / Zeroconf service discovery (or willingness to use a library that handles it)
- Server-Sent Events (the W3C [text/event-stream](#) protocol) for real-time updates

You will need physical access to an AudioFuse 16Rig or AudioFuse Studio to test against.

Supported products

Product	Support
Arturia AudioFuse 16Rig	Supported — monitoring (stereo + immersive), presets, clock, analog input/output control
Arturia AudioFuse Studio	Supported — monitoring (with source selection and headphone outputs), clock, inputs, aux/S/PDIF/ADAT/loopback output routing (no presets)
Other AudioFuse models	Not supported

See Endpoint Reference for what differs between the products.

The API requires the **latest firmware** on the device and the **latest AudioFuse Control Center** software on the host computer. It is not enabled by default — the user must turn it on in Control Center preferences (see Overview).

Document map

#	File	What's in it
1	Overview	Concepts, supported devices, prerequisites, how the user enables the API
2	Getting Started	First request in under five minutes — curl, Python, Node

#	File	What's in it
3	Discovery & Connection	Bonjour service definition, multi-device targeting, host and port resolution
4	HTTP Protocol	Methods, content types, request and response shape, batching rules
5	Events & Subscriptions	SSE protocol, /update subscription mechanism, refresh and reconnection
6	Endpoint Reference	Every endpoint, parameter type, range, and device-availability matrix
7	Error Codes	Full status-code reference with examples
8	Integration Recipes	Worked use cases for live production, hybrid music setups, and pro studios

At-a-glance: what the API looks like

```
# Read the master monitoring volume
curl http://127.0.0.1:56894/api/v1/monitoring/volume
# → {"volume": -12.5}

# Mute the main monitor output
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring \
  -H 'Content-Type: application/json' \
  -d '{"mute": true}'
# → 200 OK
```

```
# Subscribe to mute changes and stream them
curl -X POST http://127.0.0.1:56894/api/v1/update \
  -H 'Content-Type: application/json' \
  -d '{"endpoints": ["/monitoring/mute"]}'

curl -N http://127.0.0.1:56894/api/v1/events
# → event: update
#   data: {"payload": [{"key": "/monitoring/mute", "value": true}]}
```

Versioning

The current API version is **v1**, reflected in the [/api/v1](#) URL prefix. Future breaking changes will be issued under a new prefix ([/api/v2](#), etc.) and announced in Control Center release notes. Non-breaking additions (new endpoints, new fields) may appear under [/api/v1](#) without a version bump — clients should ignore unknown fields.

Feedback

This is a new capability for AudioFuse and one of the first open HTTP control surfaces in the audio interface market. If something in this documentation is unclear or incomplete, contact Arturia support with the document filename and section heading.

This API is intended for developers building custom interfaces and workflows — for broadcast, studio, podcast, and live production environments. Support is limited to documentation issues and confirmed API bugs; general programming assistance is outside scope.

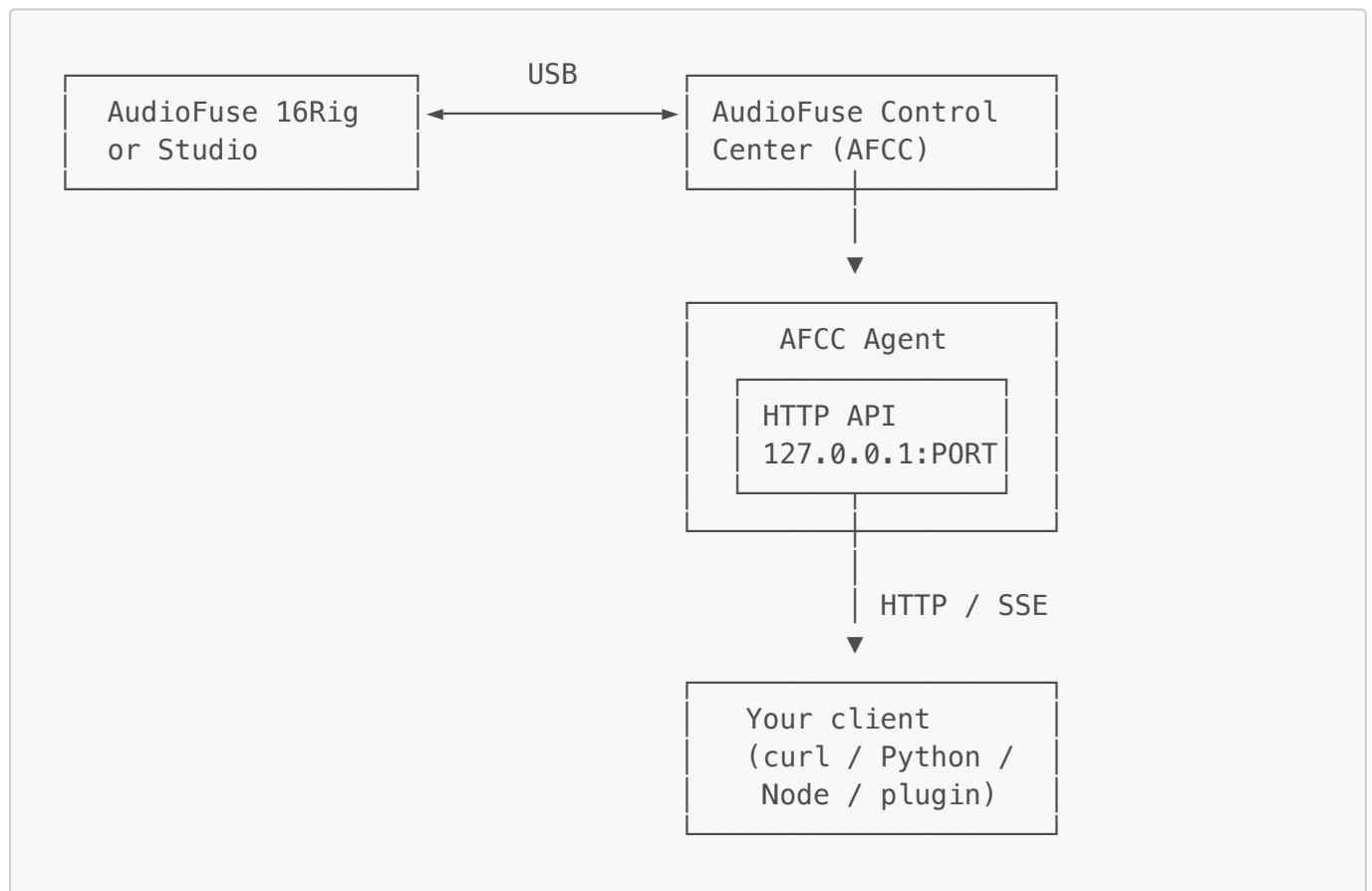
1. Overview

The AudioFuse HTTP API is a local control surface exposed by **AudioFuse Control Center** (AFCC). When enabled, AFCC opens a small HTTP server on the host computer that mirrors the device's state and accepts read and write operations against the main user-controllable parameter — gain, phantom power, monitor volume, mute, dim, mono, clock source, sample rate, presets, and more.

The API is the first of its kind for an audio interface in this product class. It is intended to enable:

- Custom hardware controllers (Stream Deck, Loupedeck, custom panels)
- DAW and broadcast-tool integrations
- Macro and automation software (Keyboard Maestro, Hammerspoon, Shortcuts, etc.)
- AI agents and assistants that can act on natural-language instructions like *"Turn on phantom power on input 1"*
- Any in-house tooling for studios, broadcasters, and content producers

Architecture



The API process lives inside *AudioFuse Control Center* background process. This means that it will **still work** with the *AudioFuse Control Center* not opened. It is bound exclusively to the loopback interface (**127.0.0.1**) and is not reachable from any other machine. The port may be dynamically re-allocated if the requested port is not available. Clients should use Bonjour / DNS-SD to discover it (see [Discovery & Connection](#)).

Supported devices

Device	Status	Notes
AudioFuse 16Rig	Supported	Monitoring (stereo + immersive), presets, clock, analog I/O
AudioFuse Studio	Supported	Core monitoring (with source selection and headphone outputs), clock, inputs, aux/S/PDIF/ADAT/loopback outputs. No <code>/preset</code> endpoints.
Any other AudioFuse model	Not supported	The API responds with <code>403 Invalid Device</code> if such a device is targeted.

Endpoint-by-endpoint device availability is enumerated in Endpoint Reference.

Prerequisites

Before any client can connect:

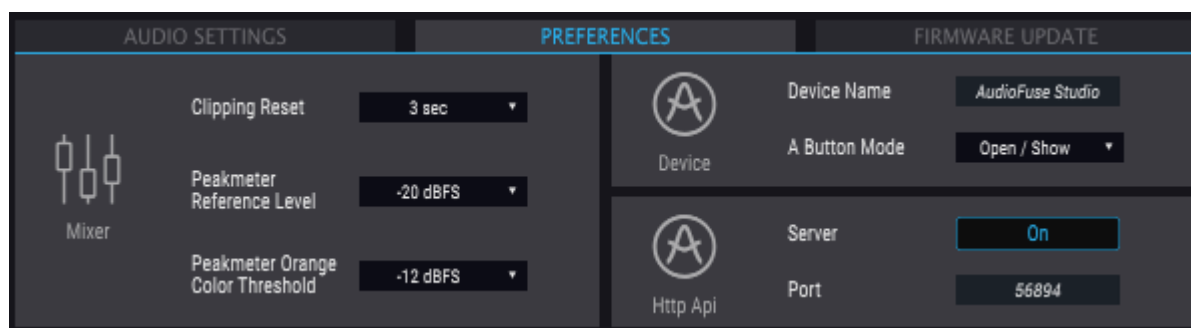
1. **AudioFuse Control Center** must be installed on the host computer. The associated background process (agent) supporting the communication between the software and the device must be running. Unless prevented, it usually automatically start during computer boot sequence.
2. The AudioFuse must be physically connected and recognized by Control Center.
3. The user must **enable the HTTP API** in Control Center preferences (see below). This setting is saved on the host computer and will be persistent across host reboot. If the HTTP API is not enabled, no service will be advertised on Bonjour and no port will be open.

Enabling the API

The HTTP API ships **disabled by default**. The end user must turn it on once in the Control Center preferences panel:

Preferences → Http Api → Server: On

When enabled, Control Center displays the active port number next to the toggle:



The port shown here is informational. Clients should **not** ask the user to type it manually — they should resolve it via Bonjour. The port may change between AFCC sessions.

For your client's onboarding flow: if your discovery finds no service, the most common cause is that the user has not enabled the API. Surface a clear message such as *"Open AudioFuse Control Center → Preferences → Http Api and turn the Server toggle to On."*

Security model

The API is **localhost-only with no authentication**. Any process on the user's computer that can reach **127.0.0.1** can read and write the device state. CORS is fully open, so any web page in any browser running on the same machine can talk to the API as well.

This is intentional for the current release — the API is a developer-facing local control surface, not an internet-facing service.

What you can do with it

The API exposes most of the parameters that exist in the Control Center UI:

- **Monitoring** — main output volume, mute, dim, mono, A/B speaker switching; reference level and immersive solo/mute (16Rig only); monitor source selection and headphone outputs (Studio only)
- **Inputs** — per-input gain, phase invert, link to neighbour, instrument mode, 48 V phantom; source selection and rear-panel switching (16Rig only); mode and pad settings (Studio only)
- **Outputs** — per-channel analog volume and link (16Rig only); aux, S/PDIF, ADAT, and loopback source routing (Studio only)
- **Clock** — sample rate, clock source (preferred and current), sync state
- **Presets** — recall, save, save-to-slot, rename (16Rig only)

Each of these has a corresponding endpoint documented in Endpoint Reference.

Next step

Continue to **Getting Started** for a working request in under five minutes, or jump straight to **Discovery & Connection** if you are wiring up a production client.

2. Getting Started

This page takes you from a fresh install to issuing your first API call quickly. Read Overview first if you have not.

0. Prepare the host

1. Make sure the device firmware and AudioFuse Control Center are up to date.
2. Connect the AudioFuse 16Rig or Studio over USB.
3. Open Control Center → **Preferences** → **Http Api** → set **Server** to **On**.
4. Note the port number that appears (e.g. **56894**). You will only use it directly for the manual examples below — production clients should discover it automatically (see Discovery & Connection).

1. Manual smoke test with curl

Replace **56894** with whatever port your AFCC is showing.

Read the API version

```
curl http://127.0.0.1:56894/api/v1/version
```

Expected:

```
{"version": "1.0.0"}
```

If you see a **Connection refused** or empty response, the API server is not running — recheck the toggle in Control Center.

List connected devices

```
curl http://127.0.0.1:56894/api/v1/devices
```

Expected:

```
{"devices": ["<your-device-serial>"]}
```

Read the master monitoring volume

```
curl http://127.0.0.1:56894/api/v1/monitoring/volume
```

Expected:

```
{"volume": -12.5}
```

Set the master monitoring volume

```
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/volume \
-H 'Content-Type: application/json' \
-d '{"volume": -18.0}'
```

Expected: HTTP 200 with empty body.

You should hear the change immediately. If you do not, check the Error Codes page — the response status will tell you exactly what was rejected.

2. Python version

The standard `requests` library is enough. No third-party AudioFuse SDK exists — the API is plain HTTP and JSON.

```
import requests

BASE = "http://127.0.0.1:56894/api/v1"

# GET
r = requests.get(f"{BASE}/monitoring/volume")
r.raise_for_status()
print("Current volume:", r.json()["volume"])

# PUT (single field)
r = requests.put(f"{BASE}/monitoring/volume", json={"volume": -18.0})
r.raise_for_status()

# POST (multiple fields at once)
r = requests.post(f"{BASE}/monitoring", json={"mute": False, "volume": -12.0})
r.raise_for_status()
```

A production client should resolve the port via Bonjour rather than hard-coding it. The Python recipe for that is in [Discovery & Connection](#).

3. Node.js version

Node 18+ ships `fetch` natively. No external library is required for the HTTP calls themselves.

```
const BASE = "http://127.0.0.1:56894/api/v1";

// GET
const volRes = await fetch(`${BASE}/monitoring/volume`);
const { volume } = await volRes.json();
console.log("Current volume:", volume);

// PUT (single field)
await fetch(`${BASE}/monitoring/volume`, {
  method: "PUT",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ volume: -18.0 }),
});

// POST (multiple fields at once)
await fetch(`${BASE}/monitoring`, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ mute: false, volume: -12.0 }),
});
```

For Bonjour discovery in Node, see [Discovery & Connection](#).

4. What to read next

- You want **real-time updates** when the user moves a knob in Control Center → [Events & Subscriptions](#)
- You want a **list of every endpoint** and what it does → [Endpoint Reference](#)
- You want to **handle multiple connected devices** → [Discovery & Connection](#)
- You want **working use cases** like a podcast guest mute or a studio session-recall → [Integration Recipes](#)

3. Discovery & Connection

The AudioFuse HTTP API is bound to the loopback interface on a specified **allocated port**. The port may not be stable across *AudioFuse Control Center agent* (AFCC agent) restarts. It is not exposed in any user-editable config file that could be safely edited nor read. Clients **should** discover it at runtime via Bonjour / DNS-SD.

Service definition

Property	Value
Service type	<code>_audiofusehttp._tcp.</code>
Domain	<code>local.</code>
Transport	TCP
Host	Local machine (always advertised on a local interface; bound to <code>127.0.0.1</code>)
Port	Dynamic — read from the resolved Bonjour service record
TXT records	Not used in the current release. Clients should not depend on TXT-record fields; treat them as optional and ignore.

When the user toggles the API on in Control Center, AFCC agent begins advertising the service. When the user toggles it off (or force the AFCC agent to quit), the service goes down. Production clients may **listen for both up and down events** so they can drop their connection when the API disappears.

Important: the API only listens on 127.0.0.1

Even though the service is advertised over Bonjour (which runs over local network interfaces), the HTTP server itself binds **only** to `127.0.0.1`. This means:

- The service record may include the host machine's LAN IP in its `addresses` array. **Do not connect to that address** — it will fail.
- Always connect using `http://127.0.0.1:<discovered-port>`.
- The Bonjour record is used **only** to learn the port number, not the host.

Defensive check: if you observe a Bonjour record whose addresses do not match any local interface on the host you are running on, treat it as a remote machine's advertisement and ignore it. The reference plugin shipped with the API uses exactly this filter.

Multi-device targeting

A single AFCC instance manages all connected AudioFuses on the host machine, so there is **one HTTP API and one Bonjour service**, regardless of how many AudioFuses are plugged in.

When more than one device is connected, every request that talks to a parameter endpoint (with the notable exception of paths `/version`, `/devices`, `/events`, `/update/endpoints`, `/update`) **must**

specify which device to target via the **targeted-device** HTTP header:

```
targeted-device: <device-serial>
```

Get the serial from **GET /api/v1/devices**:

```
curl http://127.0.0.1:56894/api/v1/devices
# → {"devices": ["AFR16-12345", "AFS-67890"]}

curl http://127.0.0.1:56894/api/v1/monitoring/volume \
  -H 'targeted-device: AFR16-12345'
```

If you omit the header when more than one device is connected, requests will return the value of the first listed device. When only one device is connected, the header is optional and AFCC routes to it implicitly.

Recommendation: always send **targeted-device** even with a single device. Your client becomes future-proof against the user plugging in a second AudioFuse mid-session.

Discovery recipes

curl (manual)

There is no curl-native Bonjour client. On the command line, use the OS tooling to find the port once, then call curl normally:

```
# macOS
dns-sd -B _audiofusehttp._tcp. # browse, Ctrl-C when you see it
dns-sd -L "AudioFuse Http Api" _audiofusehttp._tcp. local # resolve

# WSL/Linux (Avahi)
avahi-browse -r _audiofusehttp._tcp.

# Windows (with Bonjour Print Services or dns-sd installed)
dns-sd -B _audiofusehttp._tcp.
```

This is fine for ad-hoc testing. Don't rely on it for production clients.

Python

The **zeroconf** package is the standard. Install: **pip install zeroconf requests**.

```
import socket
from zeroconf import ServiceBrowser, ServiceListener, Zeroconf

class AudioFuseListener(ServiceListener):
    def __init__(self):
```

```

        self.port = None

    def add_service(self, zc, type_, name):
        info = zc.get_service_info(type_, name)
        if info and info.port:
            # The API is bound to 127.0.0.1 – we only need the port
            self.port = info.port
            print(f"Discovered AudioFuse HTTP API on port {self.port}")

    def update_service(self, zc, type_, name):
        pass

    def remove_service(self, zc, type_, name):
        print("AudioFuse HTTP API service went down")
        self.port = None

zc = Zeroconf()
listener = AudioFuseListener()
ServiceBrowser(zc, "_audiofusehttp._tcp.local.", listener)

# Wait until something is found, then issue requests
import time
while listener.port is None:
    time.sleep(0.1)

import requests
base = f"http://127.0.0.1:{listener.port}/api/v1"
print(requests.get(f"{base}/version").json())

zc.close()

```

For a long-lived client, keep the `Zeroconf` instance and the listener around for the lifetime of the process. Re-trigger your connection logic whenever `listener.port` changes (port *may* changes between AFCC sessions are normal).

Node.js

The reference Stream Deck plugin uses `bonjour-service`. Install: `npm i bonjour-service`.

```

import Bonjour from "bonjour-service";
import os from "node:os";

const SERVICE_TYPE = "audiofusehttp"; // bonjour-service prepends `._` and `._tcp.`

function getLocalIpSet(): Set<string> {
    const ips = new Set<string>();
    for (const entries of Object.values(os.networkInterfaces())) {
        for (const e of entries ?? []) {
            if (e.family === "IPv4" && !e.internal) ips.add(e.address);
        }
    }
}

```

```

    }
  }
  return ips;
}

const bonjour = new Bonjour();
const localIps = getLocalIpSet();
let discoveredPort: number | null = null;

const browser = bonjour.find({ type: SERVICE_TYPE }, (service) => {
  const addrs = new Set(service.addresses ?? []);
  const isLocal = [...addrs].some((ip) => localIps.has(ip));
  if (!isLocal) return; // ignore advertisements from other hosts on the
  LAN
  if (!service.name?.includes("audiofusehttp")) return;

  discoveredPort = service.port;
  console.log("Discovered AudioFuse HTTP API on port", discoveredPort);
});

browser.on("down", (service) => {
  if (service.port === discoveredPort) {
    console.log("Service went down");
    discoveredPort = null;
  }
});

```

Notes from the reference implementation:

- The browser type string is passed without the leading `_` and trailing `._tcp`. — the library adds them.
- The `down` event fires when AFCC quits or the user toggles the API off. Tear down any open SSE connection at that point.
- A new advertisement may arrive on a different port; treat each `up` event as authoritative.

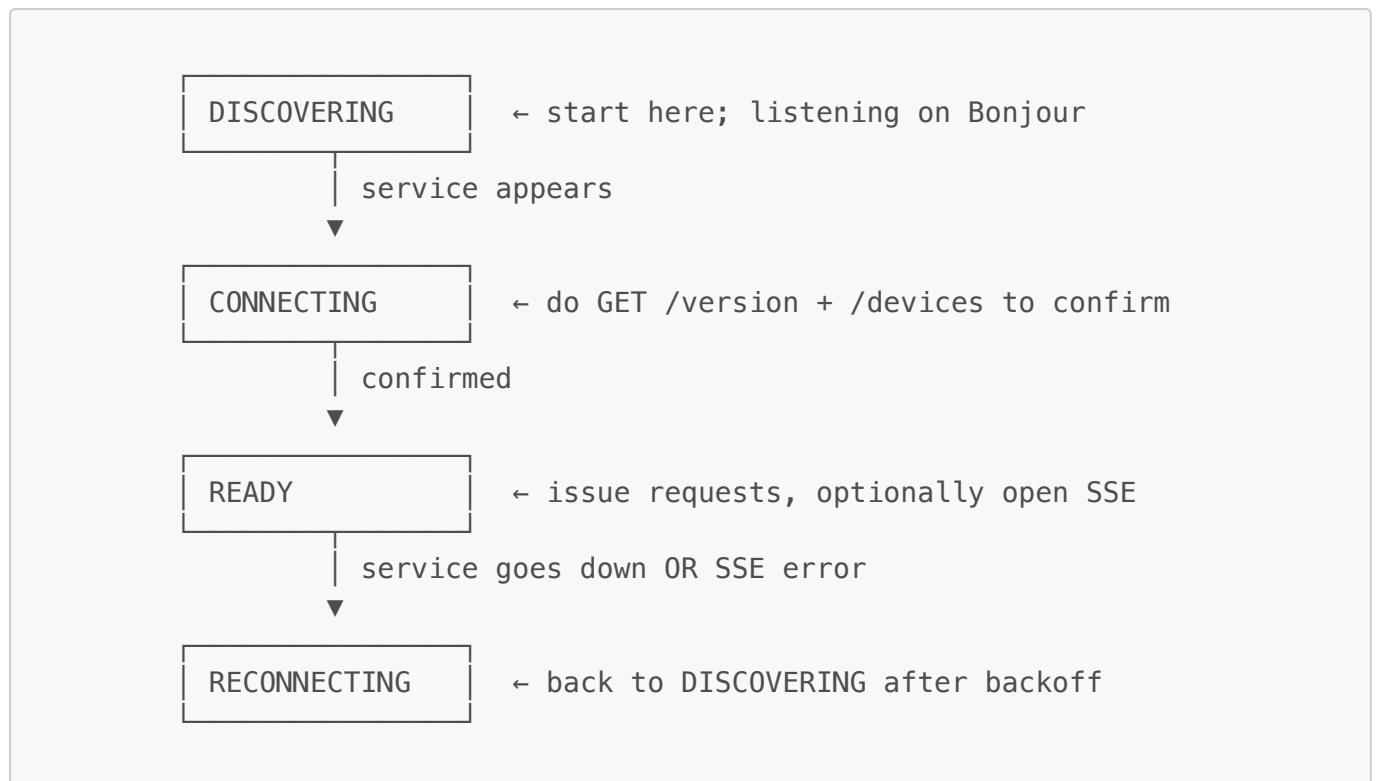
Other languages

Any DNS-SD library that can browse for `_audiofusehttp._tcp`. will work. Common picks:

Language	Library
Swift / Objective-C	<code>Network.framework</code> (Bonjour)
Java / Kotlin	<code>JmDNS</code>
Go	<code>hashicorp/mdns</code> or <code>grandcat/zeroconf</code>
Rust	<code>mdns-sd</code>
C / C++	<code>mDNSResponder</code> (Apple), <code>Avahi</code> (Linux)

Recommended connection state machine

A robust client moves through these states:



Specifics:

- **Initial confirmation:** after Bonjour resolves, call `GET /api/v1/version` once. A successful response confirms the server is live and the port is correct. Fall back to DISCOVERING if it fails.
- **Backoff:** on connection drop, wait at least 1–2 seconds before retrying. The reference SSE manager uses 2 seconds.
- **Service-down handling:** when the Bonjour `down` event fires, immediately close any open SSE connection — sockets to a stopped server may hang otherwise.

Fallback when Bonjour is unavailable

A small minority of corporate or hardened environments block multicast DNS. If your client cannot find the service after, say, 10 seconds, surface a manual override:

"Could not auto-discover AudioFuse Control Center. Open Control Center → Preferences → Http Api and copy the port shown. Paste it here:" [_____]

This is a graceful escape hatch, not a replacement for Bonjour. Do not make it the primary path.

4. HTTP Protocol

This page describes the wire-level shape of every request and response. The full per-endpoint catalogue is in Endpoint Reference; this document covers the rules that apply uniformly to all of them.

Base URL

```
http://127.0.0.1:<discovered-port>/api/v1
```

Throughout the rest of this documentation we abbreviate URLs by removing the `/api/v1` prefix. So `/monitoring/mute` means `http://127.0.0.1:<port>/api/v1/monitoring/mute`. The `/api/v1` prefix **must** appear in actual requests.

Methods

Method	When to use	Notes
GET	Read the current value of any endpoint	Available on every endpoint
PUT	Set a single field on a parent endpoint, or set the value of a leaf endpoint	Body contains exactly one field — see below
POST	Set multiple fields on a parent endpoint in one round trip	Body may contain several fields

`PUT` and `POST` are only valid on endpoints that are not read-only. Read-only endpoints respond with `405 Read-Only Resource`. Read-only endpoints are listed in grey in the Endpoint Reference table.

PUT vs POST — the practical difference

The same effect can usually be achieved with either method.

```
# Two PUTs — one field each
curl -X PUT http://127.0.0.1:56894/api/v1/input/analog/1 \
-H 'Content-Type: application/json' -d '{"gain": 2.5}'
curl -X PUT http://127.0.0.1:56894/api/v1/input/analog/2 \
-H 'Content-Type: application/json' -d '{"gain": 3.0}'
```

equals

```
# One POST — both fields, on the parent endpoint
curl -X POST http://127.0.0.1:56894/api/v1/input \
-H 'Content-Type: application/json' \
-d '{"1": {"gain": 2.5}, "2": {"gain": 3.0}}'
```

Use **POST** when you want **atomic-feeling, single-trip multi-parameter updates** (e.g. session recall, scene snapshot). Use **PUT** for **simple single-field changes** — the URL is more self-describing and easier to log.

Atomicity note: a **POST** is processed as one request, but the API does not guarantee that all fields are applied as a single hardware-level transaction. Treat it as a convenience, not a transactional primitive.

Headers

Header	Required	Purpose
Content-Type: <code>application/json</code>	Required on PUT / POST if body is present. Optional but recommended otherwise.	If present, the value must be <code>application/json</code> . Any other value yields 400 Invalid Content Type .
targeted-device: <code><serial></code>	Required when more than one AudioFuse is connected, for any endpoint other than <code>/version</code> , <code>/devices</code> , <code>/events</code> , <code>/update/endpoints</code> , <code>/update</code> . Optional otherwise.	Routes the request to the named device. Get serials from GET /devices .
Accept	Optional	The server returns JSON regardless. SSE on <code>/events</code> returns <code>text/event-stream</code> .

Request bodies

Bodies are JSON objects. For **PUT** and **POST**, the path of the URL determines the parent, and the body's keys are the children of that parent.

PUT — leaf endpoint

```
PUT /api/v1/monitoring/mute
Content-Type: application/json

{"mute": true}
```

The single field name **must** match the leaf in the URL.

PUT — parent endpoint with one child

```
PUT /api/v1/monitoring
Content-Type: application/json

{"mute": true}
```

Equivalent to the leaf form above. Allowed because **PUT** is single-field.

POST — parent endpoint, multiple children

```
POST /api/v1/monitoring
Content-Type: application/json

{"mute": false, "volume": -3.5, "dim": false}
```

POST — multiple indexed children

```
POST /api/v1/input/analog
Content-Type: application/json

{
  "1": {"gain": 2.5, "48v": true},
  "2": {"gain": 3.0, "phase_invert": true}
}
```

Indexed children are addressed by their integer index as a JSON key (always a string, since JSON object keys are strings).

Response bodies

GET responses

GET returns a JSON object. The shape mirrors the URL path: the leaf name appears as a key.

```
GET /api/v1/monitoring/volume
→ 200 OK
{"volume": -12.5}
```

For parent endpoints, the response includes all children:

```
GET /api/v1/monitoring
→ 200 OK
{
  "monitoring": {
    "mute": false,
    "dim": false,
    "volume": -12.5,
    "mono": false,
    "ab_speaker_set": false,
    ...
  }
}
```

```
}  
}
```

For options endpoints (e.g. `/clock/sample_rate_options`), the value is an array or object describing the legal values. See Endpoint Reference.

PUT / POST responses

Successful writes return **200 OK** with an empty body, or **200** with a status note if the request was a no-op (see **200 Request ignored**). Error responses return a status code in the 4xx / 5xx range with a JSON error body — the exact shape and full list is in Error Codes.

Data types

All values in JSON. Specific types per endpoint are listed in Endpoint Reference. Summary:

Type	JSON form	Example
Boolean	<code>true</code> / <code>false</code> (lowercase, unquoted)	<code>{"mute": true}</code>
Integer	Number, no decimal	<code>{"slot": 3}</code>
Float	Number, decimal allowed	<code>{"volume": -12.5}</code> — units are dB where applicable
String	JSON string	<code>{"current_source": "internal"}</code>
Array	JSON array	<code>{"endpoints": ["/monitoring/mute"]}</code>
Object	JSON object	<code>{"1": {"gain": 2.5}}</code>

Type mismatches (e.g. sending `"true"` instead of `true`) yield **400 Invalid Value Type**. Out-of-range values (e.g. `slot: 99` when the device has 8 slots) yield **400 Invalid Value**.

Special endpoints — quick orientation

These do not target device state and behave a little differently:

Endpoint	Methods	Purpose
<code>/version</code>	GET	Returns the API version. Read-only.
<code>/devices</code>	GET	Returns the serials of connected AudioFuses. Read-only.
<code>/events</code>	GET	SSE stream of pushed updates. See Events & Subscriptions.
<code>/update/endpoints</code>	GET	Returns the list of currently-subscribed endpoints. Read-only.
<code>/update</code>	POST	Subscribes additional endpoints to the SSE stream. See Events & Subscriptions.

These do not require the `targeted-device` header.

Combination rules

Most parent endpoints accept any combination of their children in a single **POST**. There are documented exceptions where the API rejects specific combinations because they would be semantically ambiguous or unsafe:

/preset — **name** cannot combine with **slot** or **saved**

These two are forbidden:

```
POST /api/v1/preset
{"name": "new name", "slot": 3}
```

```
POST /api/v1/preset
{"name": "new name", "saved": true}
```

Both yield **403 Invalid Request**. Rename a preset on its own:

```
PUT /api/v1/preset
{"name": "new name"}
```

/preset — **slot** + **saved: true** means "save to slot"

```
POST /api/v1/preset
{"slot": 5, "saved": true}
```

This saves the **current device state** to slot 5. It is the documented way to perform a "save as / save to slot" operation.

/preset/saved: false is meaningless

The API silently ignores it (returns **200 Request ignored**). The "saved" boolean is set to **false** automatically by the device whenever the current state diverges from the loaded preset; it cannot be set manually.

Channel-level link toggles have device-specific constraints

For example, **PUT /input/analog/3 {"link": false}** is rejected on AF16Rig with **403 Invalid Request** because the mini front line (ports 3–4) cannot be unlinked. Per-channel availability rules are in Endpoint Reference.

State-dependent parameter rules

Some writes depend on physical state. The most common:

- `PUT /input/analog/1 {"48v": true}` — rejected with `403 Invalid Request` on AF16Rig if no microphone is connected to that input. Check for the corresponding error and surface a helpful message to the user.

Discovery and confirmation flow

A polite client startup sequence:

- | | |
|---|------------------------|
| 1. Discover port via Bonjour | → port = 56894 |
| 2. GET /api/v1/version | → confirm server alive |
| 3. GET /api/v1/devices | → list serials |
| 4. (Optional) Subscribe to /events Subscriptions] | → see [Events & |
| 5. Issue normal GET / PUT / POST requests header | → with targeted-device |

Steps 2 and 3 take milliseconds and produce a great error message ("AFCC is running but no AudioFuse is connected") that you would otherwise have to infer from a `403 Device Not Found` on the first parameter call.

5. Events & Subscriptions

The HTTP API pushes parameter changes to clients in real time over **Server-Sent Events (SSE)**. This is the mechanism you use to keep your UI synchronized when the user moves a knob in Control Center, when the device fires a hardware event (e.g. clock loss), or when another client modifies state.

Concepts

There are two endpoints involved:

Endpoint	Method	Purpose
<code>/api/v1/update</code>	<code>POST</code>	Add endpoints to your subscription set.
<code>/api/v1/update/endpoints</code>	<code>GET</code>	Read the current subscription set (read-only listing).
<code>/api/v1/events</code>	<code>GET</code>	Open the SSE stream. The server pushes events for the endpoints in the subscription set.

The server **only pushes events for endpoints you have explicitly subscribed to**. Opening `/events` without first subscribing yields a connected stream that never emits anything.

Subscription model

The subscription is **stateful and additive**:

- `POST /api/v1/update` with a list of endpoints **adds** those endpoints to the active subscription set.
- The set must be **refreshed periodically** by re-posting the same list (or any superset). The server expires subscriptions after approximately **50 seconds** of inactivity. The reference Stream Deck implementation refreshes every **40 seconds** — comfortably below that limit. Pick a refresh interval in the 30–45 s range.
- The subscription is associated with the AFCC server process, **not** with a specific TCP connection. If you reconnect SSE, your subscription is still in force as long as you have been refreshing it.

Why a refresh? The server expires stale subscriptions so that a crashed client does not leave the API permanently busy emitting events nobody listens to.

Subscribing — request shape

```
POST /api/v1/update
Content-Type: application/json

{
  "endpoints": [
    "/monitoring/mute",
    "/monitoring/volume",
    "/input/analog/1/gain"
```

```
]
}
```

A successful response is **200 OK**. Endpoints not present in the API surface for the targeted device are silently ignored.

Listing your subscription

```
GET /api/v1/update/endpoints
```

Returns the list of endpoint paths the server currently considers subscribed.

SSE stream — **/events**

```
GET /api/v1/events
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/event-stream
Cache-Control: no-cache
Connection: keep-alive
```

Event format

Events are sent under the named event type **update**. Standard SSE framing applies:

```
event: update
data: {"payload": [{"key": "/monitoring/mute", "value": true}]}
```

(Each event ends with a blank line per the SSE spec.)

The **data** payload is JSON with this shape:

```
{
  "payload": [
    { "key": "/monitoring/mute", "value": true },
    { "key": "/monitoring/volume", "value": -3.5 }
  ]
}
```

Notes:

- **payload** is **always an array**. A single hardware change typically results in a single-element array, but coalesced or compound changes (e.g. preset recall) may emit several keys in one event.
- **key** is the canonical endpoint path, starting with **/**, **without** the **/api/v1** prefix.
- **value** carries the new value, with the type documented for that endpoint (boolean, int, float, string, object).
- Clients should treat unknown keys as forward-compatible noise and ignore them, not error out.

Other event types

The server may emit lifecycle or housekeeping events under other event names in future versions. **Listen specifically for **update**** and ignore any other event type rather than treating arbitrary lines as errors.

End-to-end example

curl

Two terminals are easiest. In terminal A, subscribe and stream:

```
# Subscribe
curl -X POST http://127.0.0.1:56894/api/v1/update \
  -H 'Content-Type: application/json' \
  -d '{"endpoints":["/monitoring/mute","/monitoring/volume"]}'

# Open the stream (-N disables curl's output buffering)
curl -N http://127.0.0.1:56894/api/v1/events
```

In terminal B, change something:

```
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/mute \
  -H 'Content-Type: application/json' \
  -d '{"mute": true}'
```

Terminal A immediately prints:

```
event: update
data: {"payload":[{"key":"/monitoring/mute","value":true}]}
```

Move a knob in Control Center — the same pattern fires.

Python — SSE client

The **sseclient-py** library handles SSE framing. Install: **pip install sseclient-py requests**.

```

import json
import threading
import time
import requests
import sseclient

PORT = 56894 # discovered via Bonjour in production
BASE = f"http://127.0.0.1:{PORT}/api/v1"

WATCH = ["/monitoring/mute", "/monitoring/volume", "/input/analog/1/gain"]

def subscribe():
    requests.post(f"{BASE}/update",
                  json={"endpoints": WATCH},
                  headers={"Content-Type":
"application/json"}).raise_for_status()

def refresh_loop(stop):
    while not stop.is_set():
        try:
            subscribe()
        except Exception as e:
            print("refresh failed:", e)
            stop.wait(40) # well under the 50s server expiry

def stream():
    while True:
        try:
            resp = requests.get(f"{BASE}/events", stream=True,
                                headers={"Accept": "text/event-stream"})
            client = sseclient.SSEClient(resp)
            for event in client.events():
                if event.event != "update":
                    continue
                msg = json.loads(event.data)
                for change in msg.get("payload", []):
                    print(f"{change['key']} = {change['value']}")
        except Exception as e:
            print("stream dropped, reconnecting in 2s:", e)
            time.sleep(2)

subscribe()
stop = threading.Event()
threading.Thread(target=refresh_loop, args=(stop,), daemon=True).start()
try:
    stream()
finally:
    stop.set()

```

Node 18+ ships native `fetch` but not native `EventSource`. Install: `npm i eventsource`.

```
import { EventSource } from "eventsources";

const PORT = 56894; // discovered via Bonjour in production
const BASE = `http://127.0.0.1:${PORT}/api/v1`;

const WATCH = [
  "/monitoring/mute",
  "/monitoring/volume",
  "/input/analog/1/gain",
];

async function subscribe() {
  const r = await fetch(`${BASE}/update`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ endpoints: WATCH }),
  });
  if (!r.ok) throw new Error(`subscribe failed: ${r.status}`);
}

let es: EventSource | null = null;

function connect() {
  es = new EventSource(`${BASE}/events`);

  es.addEventListener("update", (e: MessageEvent) => {
    const { payload } = JSON.parse(e.data);
    for (const change of payload) {
      console.log(`${change.key} = ${JSON.stringify(change.value)}`);
    }
  });

  es.onerror = () => {
    console.warn("stream dropped, reconnecting in 2s");
    es?.close();
    es = null;
    setTimeout(connect, 2000);
  };
}

await subscribe();
connect();

// Periodic refresh of the subscription
setInterval(() => {
  subscribe().catch((e) => console.warn("refresh failed:", e));
}, 40_000);
```

Implementation guidance

A single shared SSE connection is sufficient for an entire client. The reference Stream Deck plugin uses a singleton SSE manager with:

- **One `EventSource`** for the whole process
- A **listener registry** keyed by endpoint path, so multiple UI components can subscribe to the same key without duplicating connections
- A **periodic refresh** of the full subscription set
- **Automatic reconnection** with a fixed backoff on error
- **Teardown** when the listener registry empties or the Bonjour service goes down

If you maintain more than one SSE connection per process you will multiply the server's bookkeeping load with no benefit. Don't.

Recommended values

Setting	Recommended value	Rationale
Subscription refresh interval	30–45 s	Comfortably under the 50 s reference cadence; gives the server time-budget headroom
SSE reconnect delay	2 s	Matches the reference plugin; avoids hot loops if AFCC is restarting
Initial subscribe before connecting <code>/events</code>	Yes	Prevents a race where the stream is open but no events have been requested
Re-subscribe after reconnect	Yes (defensive)	Cheap; covers the case where AFCC was restarted while you were disconnected

What to do when the Bonjour service goes down

The API has been stopped (user toggled it off, AFCC agent quit, etc.). You should:

1. Close the SSE connection immediately (otherwise reads may hang).
2. Stop the refresh timer.
3. Return to the discovery state and wait for the service to reappear.

The reference implementation handles this in its `disconnect()` method — close the `EventSource`, clear the timer, drop the cached port.

Common pitfalls

- **Forgetting to subscribe.** `/events` returns 200 and stays open even with an empty subscription set. The symptom is silence. Always subscribe first.
- **Letting the subscription expire.** If you stop refreshing for more than ~50 s, the server may quietly drop you. Set the refresh timer when you connect, clear it when you disconnect.
- **Listening on the default `message` event instead of `update`.** Browser `EventSource` defaults to firing `onmessage` for unnamed events. AudioFuse uses **named events**, so you must use `addEventListener("update", ...)`.
- **Treating `payload` as a single object.** It is always an array, even for a single change.

- **Reconnecting on a tight loop.** Always back off at least 1 s — typically 2 s — before reconnecting after an error.

6. Endpoint Reference

This is the canonical catalogue of every endpoint exposed by the AudioFuse HTTP API. Each table column means:

- **Endpoint** — full path under `/api/v1`. `:index` is a 1-based integer (channel number).
- **Type** — JSON value type accepted by `PUT` / `POST` and returned by `GET`.
- **Access** — `RW` (readable + writable) or `RO` (read-only — `GET` only; writing yields `405 Read-Only Resource`).
- **Devices** — which AudioFuse models expose the endpoint.
- **SSE** — ✓ if this endpoint can push real-time update events (see Events & Subscriptions).
- **Notes** — semantic constraints, valid ranges, channel-index limits.

Where an endpoint is marked **16Rig only**, calling it on a Studio yields `404 Not Available`. The converse applies for **Studio only** endpoints.

Parent nodes (e.g. `/monitoring`, `/input`) always accept `GET` (returning all children) and accept `PUT` / `POST` with bodies that name children. They are listed once at the top of each section and not repeated in the table.

Special endpoints

Endpoint	Method(s)	Access	Devices	Description
<code>/version</code>	GET	RO	All	Returns the API version, e.g. <code>{"version": "1.0.0"}</code> .
<code>/devices</code>	GET	RO	All	Returns the serials of currently connected devices, e.g. <code>{"devices": ["AFR16-12345"]}</code> .
<code>/events</code>	GET	RO	All	SSE stream for pushed parameter updates. See Events & Subscriptions.
<code>/update/endpoints</code>	GET	RW	All	Returns the list of endpoints currently subscribed to <code>/events</code> .
<code>/update</code>	POST	—	All	Adds endpoints to the subscription set. Body: <code>{"endpoints": ["/path/one", "/path/two"]}</code> .

These endpoints do **not** require the `targeted-device` header even when multiple AudioFuses are connected.

`/monitoring` — main monitor controller

Parent node: `/monitoring` (RW; `GET` returns all fields, `POST` updates any subset).

Common monitor controls (Studio + 16Rig)

Endpoint	Type	Access	SSE	Notes
/monitoring/mute	boolean	RW	✓	Master output mute.
/monitoring/dim	boolean	RW	✓	Master output dim.
/monitoring/volume	float	RW	✓	Master output volume in dB.
/monitoring/mono	boolean	RW	✓	Mono fold-down on master out.
/monitoring/ab_speaker_set	boolean	RW	✓	Toggle between speaker set A and B.

Reference level — 16Rig only

Endpoint	Type	Access	SSE	Notes
/monitoring/reference_level	float	RW	✓	Reference level (dB) used by the reference-recall function.
/monitoring/is_at_reference_level	boolean	RO	✓	true when current volume equals the configured reference level.

Monitor source — Studio only

Endpoint	Type	Access	SSE	Notes
/monitoring/source	string	RW	✓	Active monitor source. Must be one of source_options .
/monitoring/source_options	object	RO	—	Enumerates the legal monitor source selections for the connected device.

Headphone outputs — Studio only

Parent nodes: /monitoring/phones (RO), /monitoring/phones/:index (RW; **GET** returns all fields for that phones output).

:index is the 1-based headphone output number.

Endpoint	Type	Access	SSE	Notes
/monitoring/phones/:index/mono	boolean	RW	✓	Mono fold-down for this headphone output.
/monitoring/phones/:index/ab_speaker_set	boolean	RW	✓	Toggle between speaker set A and B for this output.

Endpoint	Type	Access	SSE	Notes
				Index 2 only — returns 404 Not Available on index 1.
<code>/monitoring/phones/:index/source</code>	string	RW	✓	Source routed to this headphone output. Must be one of <code>source_options</code> .
<code>/monitoring/phones/:index/source_options</code>	object	RO	—	Enumerates the legal source selections for this headphone output.

Immersive monitor controls — **16Rig only**

These endpoints exist exclusively on AudioFuse 16Rig because they belong to its immersive monitoring stage. They return **404 Not Available** on Studio.

Endpoint	Type	Access	SSE	Notes
<code>/monitoring/bass_management</code>	boolean	RW	✓	Enable bass-management for the immersive bus.
<code>/monitoring/lip_sync</code>	boolean	RW	✓	A/V sync compensation.
<code>/monitoring/lfe_10db</code>	boolean	RW	✓	LFE channel +10 dB boost.
<code>/monitoring/solo_fronts</code>	boolean	RW	✓	Solo the front bed.
<code>/monitoring/solo_left_right</code>	boolean	RW	✓	Solo L/R only.
<code>/monitoring/solo_center</code>	boolean	RW	✓	Solo centre channel.
<code>/monitoring/solo_surrounds</code>	boolean	RW	✓	Solo surround channels.
<code>/monitoring/solo_heights</code>	boolean	RW	✓	Solo height channels.
<code>/monitoring/solo_lfe</code>	boolean	RW	✓	Solo LFE channel.
<code>/monitoring/mute_fronts</code>	boolean	RW	✓	Mute the front bed.
<code>/monitoring/mute_left_right</code>	boolean	RW	✓	Mute L/R only.
<code>/monitoring/mute_center</code>	boolean	RW	✓	Mute centre channel.
<code>/monitoring/mute_surrounds</code>	boolean	RW	✓	Mute surround channels.
<code>/monitoring/mute_heights</code>	boolean	RW	✓	Mute height channels.
<code>/monitoring/mute_lfe</code>	boolean	RW	✓	Mute LFE channel.

/clock — sample-rate and clock-source controller

Parent node: /clock (RW for the writable children below).

Endpoint	Type	Access	Devices	SSE	Notes
/clock/sample_rate	integer	RW	All	✓	Current sample rate in Hz (e.g. 48000, 96000). Must be one of sample_rate_options.
/clock/sample_rate_options	array	RO	All	—	Enumerates the legal sample rates supported by the connected device.
/clock/current_source	string	RO	All	✓	The clock source the device is currently locked to . May differ from preferred_source if the preferred source is missing.
/clock/preferred_source	string	RW	All	✓	The clock source the user wishes to use. Must be one of source_options.
/clock/source_options	array	RO	All	—	Enumerates the legal clock sources for the connected device.
/clock/sync	boolean	RO	All	✓	true when the device is locked and stable on its current source.

Pattern: when changing preferred_source, watch current_source and sync over SSE. The device may take a moment to lock, and may fall back to internal if the requested source is not present.

/input — input channels

Parent nodes:

- /input (RW)
- /input/analog (RW)
- /input/analog/:index (RW; GET returns all fields for the channel)

:index is the 1-based analog input number. The number of inputs and available endpoints depend on the device.

Endpoints available on both devices

Where the available index range differs between devices it is noted in the **Notes** column. Calling an endpoint outside the supported index range returns **404 Not Available**.

Endpoint	Type	Access	Notes
/input/analog/:index/link	boolean	RW	Links the channel to its neighbour as a stereo pair. AF16Rig : all inputs. Studio : index 5–8 only.
/input/analog/:index/gain	float	RW	Input gain in dB. AF16Rig : all inputs. Studio : index 5–8 only.
/input/analog/:index/phase_invert	boolean	RW	Polarity flip on the input. AF16Rig : all inputs. Studio : index 1–4 only.
/input/analog/:index/inst	boolean	RW	Hi-Z instrument mode. AF16Rig : index 1–2 only. Studio : index 1–4 only.
/input/analog/:index/48v	boolean	RW	48 V phantom power. AF16Rig : index 1–2 only. Studio : index 1–4 only. On AF16Rig, enabling phantom is rejected with 403 Invalid Request if no microphone is connected to that input.

16Rig-only input endpoints

Endpoint	Type	Access	Index range	Notes
/input/analog/:index/source	string	RW	All inputs	Source selection for the channel (e.g. line / mic / instrument flavour). Allowed values depend on hardware.
/input/analog/:index/rear	boolean	RW	1–4 only	Selects the rear-panel input instead of the front. Only available on the first four channels.

Studio-only input endpoints

Endpoint	Type	Access	Index range	Notes
<code>/input/analog/:index/mode</code>	string	RW	5–8 only	Input mode selection for high-numbered channels. Must be one of <code>mode_options</code> .
<code>/input/analog/:index/mode_options</code>	object	RO	5–8 only	Enumerates the legal mode values for this input.
<code>/input/analog/:index/pad</code>	string	RW	All inputs	Input pad setting.
<code>/input/analog/:index/pad_options</code>	object	RO	All inputs	Enumerates the legal pad values for this input.

`/output` — output channels

Parent node: `/output` (RW).

The available output sub-paths differ significantly between devices. AudioFuse 16Rig exposes direct per-channel analog outputs; AudioFuse Studio exposes aux sends, S/PDIF, ADAT, and loopback.

Analog outputs — **16Rig only**

Parent nodes: `/output/analog` (RW), `/output/analog/:index` (RW for index 3–10).

Endpoint	Type	Access	Index range	Notes
<code>/output/analog/:index/link</code>	boolean	RW	3–10 only	Links the channel to its neighbour as a stereo pair.
<code>/output/analog/:index/volume</code>	float	RW	3–10 only	Per-output volume in dB.

Outputs 1 and 2 are the main monitor outputs and are controlled through `/monitoring/volume` rather than per-channel endpoints. Outputs outside 3–10 return `404 Not Available`.

Aux outputs — **Studio only**

Parent nodes: `/output/aux` (RW), `/output/aux/:side` (RW).

`:side` must be `l` (left) or `r` (right).

Endpoint	Type	Access	Notes
<code>/output/aux/:side/reamp</code>	boolean	RW	Enables re-amp mode on this aux output side.

Endpoint	Type	Access	Notes
<code>/output/aux/:side/link</code>	boolean	RW	Links the left and right aux outputs as a stereo pair.
<code>/output/aux/:side/source</code>	string	RW	Source routed to this aux output side. Must be one of <code>source_options</code> .
<code>/output/aux/:side/source_options</code>	object	RO	Enumerates the legal source selections for this aux side.
<code>/output/aux/:side/volume</code>	float	RW	Volume in dB for this aux output side.

S/PDIF output — Studio only

Parent node: `/output/spdif` (RW).

Endpoint	Type	Access	Notes
<code>/output/spdif/source</code>	string	RW	Source routed to the S/PDIF output. Must be one of <code>source_options</code> .
<code>/output/spdif/source_options</code>	object	RO	Enumerates the legal source selections for the S/PDIF output.

ADAT output — Studio only

Parent nodes: `/output/adat` (RW), `/output/adat/:index` (RW).

`:index` is the 1-based ADAT channel number.

Endpoint	Type	Access	Notes
<code>/output/adat/:index/source</code>	string	RW	Source routed to this ADAT output channel. Must be one of <code>source_options</code> .
<code>/output/adat/:index/source_options</code>	object	RO	Enumerates the legal source selections for this ADAT channel.

Loopback output — Studio only

Parent node: `/output/loopback` (RW).

Endpoint	Type	Access	Notes
<code>/output/loopback/source</code>	string	RW	Source routed to the loopback output. Must be one of <code>source_options</code> .

Endpoint	Type	Access	Notes
<code>/output/loopback/source_options</code>	object	RO	Enumerates the legal source selections for the loopback output.

`/preset` — preset management — **16Rig only**

These endpoints exist exclusively on AudioFuse 16Rig. On Studio, they return **404 Not Available**.

Parent node: `/preset` (RW, with combination rules — see below).

Endpoint	Type	Access	SSE	Notes
<code>/preset/slot</code>	integer	RW	✓	Currently loaded preset slot. Valid range: 1–8 . Setting it recalls that preset. Out-of-range values yield 400 Invalid Value .
<code>/preset/name</code>	string	RW	✓	Name of the currently loaded preset. Renaming cannot be combined with other fields in a single POST (see below).
<code>/preset/saved</code>	boolean	RW (with rules)	✓	true when the loaded preset matches the device state; false when there are unsaved edits. See special semantics below.
<code>/preset/names</code>	string array	RO	—	List of all 8 slot names in slot order.

`/preset` — special combination rules

Three combinations have non-obvious behaviour:

1. Save current edits to the loaded preset

```
PUT /api/v1/preset
{"saved": true}
```

Persists any in-RAM modifications to the currently loaded slot. **200 OK** on success; **200 Request ignored** if there is nothing to save.

2. Save current device state to a specific slot ("save as")

```
POST /api/v1/preset
{"slot": 5, "saved": true}
```

This is the documented idiom for *Save to slot 5*. It both writes the current state to slot 5 **and** loads that slot. (After this call, **slot** is 5 and **saved** is **true**.)

3. Forbidden combinations

```
POST /api/v1/preset
{"name": "new name", "slot": 3}
```

```
POST /api/v1/preset
{"name": "new name", "saved": true}
```

Both yield **403 Invalid Request**. Renaming is its own atomic operation:

```
PUT /api/v1/preset
{"name": "new name"}
```

4. **saved: false** is a no-op

```
PUT /api/v1/preset
{"saved": false}
```

The server returns **200 Request ignored**. The "saved" flag is set to **false** automatically by the device when the current state diverges from the loaded preset; clients cannot force it.

Device availability summary

A compact view of what is and is not available per device. Yes = present, No = not present (returns 404), Partial = available on a subset of indices (see notes).

Endpoint group	AudioFuse Studio	AudioFuse 16Rig	Notes
/version , /devices , /events , /update*	Yes	Yes	
/clock/*	Yes	Yes	
/input/analog/:index/link	Partial	Yes	Studio: index 5–8 only. 16Rig: all inputs.
/input/analog/:index/gain	Partial	Yes	Studio: index 5–8 only. 16Rig: all inputs.

Endpoint group	AudioFuse Studio	AudioFuse 16Rig	Notes
/input/analog/:index/phase_invert	Partial	Yes	Studio: index 1–4 only. 16Rig: all inputs. (1)
/input/analog/:index/inst	Partial	Partial	Studio: index 1–4. 16Rig: index 1–2 only. (1)
/input/analog/:index/48v	Partial	Partial	Studio: index 1–4. 16Rig: index 1–2 only. (1)
/input/analog/:index/source	No	Yes	16Rig only.
/input/analog/:index/rear	No	Partial	16Rig only, index 1–4.
/input/analog/:index/mode, mode_options	Partial	No	Studio only, index 5–8.
/input/analog/:index/pad, pad_options	Yes	No	Studio only, all inputs.
/output/analog/*	No	Partial	16Rig only, index 3–10.
/output/aux/*	Yes	No	Studio only.
/output/spdif/*	Yes	No	Studio only.
/output/adat/*	Yes	No	Studio only.
/output/loopback/*	Yes	No	Studio only.
/monitoring core (mute, dim, volume, mono, ab_speaker_set)	Yes	Yes	
/monitoring/reference_level, is_at_reference_level	No	Yes	16Rig only.
/monitoring/source, source_options	Yes	No	Studio only.
/monitoring/phones/*	Yes	No	Studio only.
/monitoring immersive (bass_management, lip_sync, lfe_10db, solo_, mute_ per-channel)	No	Yes	16Rig only.
/preset/*	No	Yes	16Rig only.

Warning (1): Pad and Phase invert is only possible with a plugged connector in the corresponding input, instrument activation only with a plugged Jack, and 48V power only with an XLR plugged.

Sample-rate and clock-source values

The exact set of legal values is reported by the device in `/clock/sample_rate_options` and `/clock/source_options`. **Do not hard-code** these values in your client — read them at startup. The available rates and sources can vary between firmware revisions and product variants.

A typical AudioFuse 16Rig response:

```
GET /api/v1/clock/sample_rate_options
→ {"sample_rate_options": {"keys": [44100, 48000, 88200, 96000, 176400, 192000], "values": ["44.1kHz", "48kHz", "88.2kHz", "96kHz", "176.4kHz", "192kHz"]}}
```

```
GET /api/v1/clock/source_options
→ {"source_options": {"keys": ["internal", "word_clock", "adat", "spdif"], "values": ["Internal", "Word Clock", "ADAT", "S/PDIF"]}}
```

Always validate user choices against the values returned here before sending a **PUT**.

7. Error Codes

The API uses standard HTTP status codes. Successful operations return **200 OK**; everything else is a 4xx or 5xx with a short descriptive reason. This page lists every status the server can return, what it means, and what your client should do about it.

At-a-glance table

Status	Reason	Client action
200	OK	Treat as success.
200 Request ignored	Write was a no-op	Treat as success; suppress UI feedback if it would be misleading.
400 Invalid Content Type	Content-Type header was wrong	Fix your headers.
400 Empty Request Body	Write request had no body	Add a body.
400 Request Body Parsing Failed	Body was not valid JSON	Fix the JSON.
400 Invalid Value	A value was out of range	Validate before sending.
400 Invalid Value Type	A value's JSON type was wrong	Validate before sending.
403 Device Not Found	No matching AudioFuse connected	Check targeted-device header / poll /devices .
403 Invalid Device	Device does not support the API	Tell the user; nothing to retry.
403 Invalid Request	Combination, channel, or device-state rule violation	Fix the request shape; see specific sub-cases.
404 Not Available	Endpoint does not exist on this device	Use the device-availability matrix in Endpoint Reference.
405 Read-Only Resource	Tried to write a read-only endpoint	Use GET only.
429 Too many requests	Too many attempts in on a short amount of time.	Use throttling to or event coalescing to reduce the number of calls
500 Unexpected Error	Hardware / driver fault	Surface to user; suggest restart.

Success

200 OK

The request completed successfully. For **GET**, the body contains the requested data. For **PUT** / **POST**, the body is empty.

200 Request ignored

A write request (**PUT** or **POST**) was accepted but did not change anything because the requested state already matched the current state, or because the field has documented no-op behaviour.

The most common cases:

- **PUT /preset {"saved": true}** when the preset already has no unsaved edits.
- **PUT /preset {"saved": false}** (always ignored — see Endpoint Reference).
- Setting any value to its current value (e.g. muting an already-muted output).

This is **not an error**. Treat it as success. If your UI would normally show a "value changed" toast, suppress it for ignored requests.

Connection issues

403 Device Not Found

The request targets a device — i.e. any endpoint other than **/version**, **/devices**, **/events**, **/update**, **/update/endpoints** — but no matching device is connected.

Common causes:

- The user has unplugged the AudioFuse since your client last cached its serial.
- You are sending a **targeted-device** header with a serial that no longer matches anything.
- No AudioFuse at all is connected to AFCC.

Recommended client behaviour: refresh **GET /devices**, update your cached serial list, and retry. If **/devices** returns an empty list, surface a "Plug in your AudioFuse" message rather than retrying in a loop.

Bad request syntax (400)

The four **400** errors below are all caused by malformed requests. Treat them as developer-side bugs in your client, not transient failures — never retry them.

400 Invalid Content Type

The **Content-Type** request header was set to something other than **application/json**. The header is optional, but if present it must be **application/json**.

```
PUT /api/v1/monitoring/mute
Content-Type: text/plain
{"mute": true}
```

→ **400 Invalid Content Type**. Either drop the header or set it to `application/json`.

400 Empty Request Body

A **PUT** or **POST** was sent without a body, but the endpoint requires one.

```
POST /api/v1/monitoring  
(no body)
```

→ **400 Empty Request Body**.

400 Request Body Parsing Failed

The body could not be parsed as JSON. Typical causes: trailing commas, missing quotes, unescaped characters.

```
POST /api/v1/monitoring  
{mute: true}      ← unquoted key
```

→ **400 Request Body Parsing Failed**.

400 Invalid Value

A field's value is outside the legal range for that endpoint.

```
PUT /api/v1/preset {"slot": 9}
```

→ **400 Invalid Value** (slot range is 1–8 on AF16Rig).

```
PUT /api/v1/clock/sample_rate {"sample_rate": 12345}
```

→ **400 Invalid Value** (only the values in `sample_rate_options` are accepted).

Recommended: validate against `*_options` endpoints (e.g. `/clock/sample_rate_options`, `/clock/source_options`) before sending.

400 Invalid Value Type

The JSON type does not match the endpoint's documented type.

```
PUT /api/v1/monitoring {"mute": "true"}    ← string instead of boolean
```

→ **400 Invalid Value Type**. Send `true`, not `"true"`.

This is a common mistake when constructing JSON from user input — always coerce types before serialising.

Forbidden requests (403)

403 Invalid Device

The targeted AudioFuse model does not support the HTTP API at all (e.g. an older AudioFuse). Different from **403 Device Not Found**: this means the device is connected but ineligible.

Nothing to retry. Show the user a message explaining that this AudioFuse model does not support the HTTP API.

403 Invalid Request

A catch-all for requests that are syntactically valid JSON but semantically rejected. There are three documented sub-cases:

a. Forbidden field combination

```
POST /api/v1/preset {"slot": 3, "name": "my_preset"}
```

→ **403 Invalid Request**. `slot` and `name` cannot be set together. See Endpoint Reference for the full preset rules.

b. Channel-number incompatibility

```
PUT /api/v1/input/analog/3 {"link": false}
```

→ **403 Invalid Request** on AF16Rig. Inputs 3 and 4 (the mini front line) cannot be unlinked.

c. Device-state incompatibility

```
PUT /api/v1/input/analog/1 {"48v": true}    ← no microphone connected to  
input 1
```

→ **403 Invalid Request** on AF16Rig. The device refuses to engage 48 V phantom on a port with nothing plugged in.

Recommended: when you receive **403 Invalid Request**, surface a context-aware message rather than a generic error. The same status covers three quite different failure modes; the only way to distinguish them is the request you sent.

404 Not Available

The endpoint exists in the API in general but is **not available on the targeted device**. Examples:

- `GET /api/v1/preset/slot` on AudioFuse Studio (presets are 16Rig-only).
- `GET /api/v1/monitoring/solo_center` on AudioFuse Studio (immersive features are 16Rig-only).
- `GET /api/v1/input/analog/9/inst` on any device (`inst` only exists for inputs 1–2).
- `GET /api/v1/output/analog/2/volume` (output 1–2 are governed by `/monitoring`, not per-channel endpoints).

Recommended: build your endpoint surface from the device-availability matrix in Endpoint Reference at startup, based on which device(s) `GET /devices` reports.

405 Read-Only Resource

You sent a write to a read-only endpoint.

```
PUT /api/v1/clock/source_options {"source_options": ["a", "b"]}
```

→ **405 Read-Only Resource**. Read-only endpoints are listed with **RO** in Endpoint Reference. They include all `*_options`, `current_source`, `sync`, `is_at_reference_level`, `names`, `version`, `devices`, and `update/endpoints`.

429 Too many requests

Requests are forwarded to and validated by the hardware. The firmware enforces a rate limit to give the device time to apply each change before the next one arrives.

```
PUT /api/v1/clock/sample_rate {"sample_rate": 96000}
→ 429 Too many requests
```

Recommended: do not hammer the API in a tight loop. Coalesce rapid UI interactions (e.g. a scrubbing slider) so that only the final settled value is sent. If you do receive a **429**, back off briefly and retry — it is a transient condition, not a permanent rejection.

500 Unexpected Error

An internal failure — usually a hardware or driver problem rather than a problem with your request. The most common trigger is `/clock/preferred_source`, which can fail if the audio driver loses track of the device.

Recommended: surface the failure to the user and suggest

1. Checking that the AudioFuse is still connected and powered.
2. Restarting the AudioFuse.
3. Restarting AudioFuse Control Center.
4. Reproducing the issue and contacting Arturia support if it persists.

Do not retry indefinitely on **500** — back off, surface the problem, and let the user act.

Recommended retry policy

Status range	Retry?	Backoff
200	n/a (success)	—
400	No. Client bug.	—
403 Device Not Found	Yes, after refreshing /devices .	500 ms then exponential up to 5 s.
403 Invalid Device / 403 Invalid Request	No. Surface to user.	—
404 Not Available	No. Endpoint genuinely missing.	—
405 Read-Only Resource	No. Client bug.	—
429 Too many requests	Yes, but slow down. Coalesce requests and reduce send rate.	200 ms minimum between writes.
500 Unexpected Error	At most once or twice, then surface.	1 s, 3 s.

For SSE-stream errors (which are not HTTP status codes per se but socket failures), see the reconnect guidance in Events & Subscriptions.

8. Integration Recipes

This page translates real production scenarios into concrete API call sequences. The goal is to give you copy-pasteable patterns; mapping them to your specific UI surface (controller buttons, macro keys, voice commands, an LLM agent) is left to you.

The recipes are organized by user persona, each drawn from the live use cases the API was designed for:

1. Live content production — podcast, interview, streaming
2. Bedroom hybrid music setup — synth and instruments
3. Professional production / mixing / mastering studio
4. AI-agent style natural-language control

All examples assume `PORT` has been resolved via `Bonjour` (see Discovery & Connection). For brevity, the examples hard-code a port — your client should not.

1. Live content production

Per-guest mute with state feedback

Goal: a single button per guest that mutes the corresponding input and shows red when muted, green when live. Works for podcast hosts on AF16Rig (inputs 1 and 2 are mic preamps) and for interview rigs.

State source: the API does not have a "channel mute" concept on inputs (mute is a monitor-side concept). The cleanest pattern is to hold the channel's gain at its working value when "live" and snap it to a sentinel value (e.g. `-inf` or the floor) when "muted", or — preferred — to use a DAW-side mute keyed to a flag you toggle on the device. For pure interface-level muting of guest mics, toggle phantom or use the input source switch as a hard kill, but the recommended approach is to **drive a DAW-side input mute from this button** while showing the channel state via SSE.

A working pattern using gain as the kill switch:

```
# Mute guest 1 (input 1)
curl -X PUT http://127.0.0.1:56894/api/v1/input/analog/1/gain \
  -H 'Content-Type: application/json' \
  -d '{"gain": -60.0}'

# Unmute guest 1 (restore the working value you stored)
curl -X PUT http://127.0.0.1:56894/api/v1/input/analog/1/gain \
  -H 'Content-Type: application/json' \
  -d '{"gain": 28.0}'
```

To make the LED reflect reality (so external changes from Control Center are picked up), subscribe to the channel:

```
POST /api/v1/update
{"endpoints": ["/input/analog/1/gain"]}
```

Then in the SSE handler, redraw red when `value <= -50.0` (your "muted" threshold) and green otherwise.

Hold-to-cough — momentary mute while held

```
# On key down
requests.put(f"{BASE}/monitoring/mute", json={"mute": True})

# On key up
requests.put(f"{BASE}/monitoring/mute", json={"mute": False})
```

Use `/monitoring/mute` for "kill the master cleanly during a coughing fit", or use the per-input gain pattern above for a single-channel cough mute on the host's mic.

Sound-effect / jingle channel un-mute timed to a cue

```
async function fireJingle(inputIndex, durationMs) {
  await fetch(`${BASE}/input/analog/${inputIndex}/gain`, {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ gain: 0.0 }),
  });
  setTimeout(async () => {
    await fetch(`${BASE}/input/analog/${inputIndex}/gain`, {
      method: "PUT",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ gain: -60.0 }),
    });
  }, durationMs);
}
```

Record-arm + level-trim nudges without leaving frame

A "+1 dB" / "-1 dB" pair on a controller, debounced into a single `PUT`. Read first, modify, write — do not assume the previous value:

```
def nudge_input_gain(index: int, delta_db: float):
    cur = requests.get(f"{BASE}/input/analog/{index}/gain").json()["gain"]
    new = max(-12.0, min(60.0, cur + delta_db)) # clamp to your safe
    range
    requests.put(f"{BASE}/input/analog/{index}/gain", json={"gain": new})
```

2. Bedroom hybrid music setup

One-button setup switch — "synth jam" vs "guitar tracking"

A single **POST** to a parent endpoint applies many parameters in one round trip. This is the right shape for snapshot-style scene recall.

```
# Snapshot A – modular / synth jam: line-level inputs, no phantom, monitor
at -12
# recipe here assume AudioFuse 16Rig
curl -X POST http://127.0.0.1:56894/api/v1/input/analog \
  -H 'Content-Type: application/json' \
  -d '{
    "1": {"gain": 6.0, "inst": false, "48v": false, "phase_invert":
false},
    "2": {"gain": 6.0, "inst": false, "48v": false, "phase_invert":
false}
  }'
curl -X POST http://127.0.0.1:56894/api/v1/monitoring \
  -H 'Content-Type: application/json' \
  -d '{"volume": -12.0, "dim": false, "mono": false}'

# Snapshot B – guitar tracking: input 1 in instrument mode, input 2 mic
with phantom
# recipe here assume AudioFuse 16Rig
curl -X POST http://127.0.0.1:56894/api/v1/input/analog \
  -H 'Content-Type: application/json' \
  -d '{
    "1": {"gain": 22.0, "inst": true, "48v": false},
    "2": {"gain": 36.0, "inst": false, "48v": true}
  }'
curl -X POST http://127.0.0.1:56894/api/v1/monitoring \
  -H 'Content-Type: application/json' \
  -d '{"volume": -8.0, "dim": false}'
```

Tip: store snapshots as JSON blobs in your client's own storage. The on-device preset slots (`/preset`, 16Rig only) are fewer, less granular, and serve a different purpose — they capture the device's full state, including routing.

For an AF16Rig user who wants the same snapshot model but **persisted on the device**, use **POST** `/preset {"slot": <n>, "saved": true}` to capture the current state into one of the eight slots, then recall it later with **PUT** `/preset {"slot": <n>}`.

Mono-sum and phase-flip mix sanity check

```
# Mono sum on
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/mono \
  -H 'Content-Type: application/json' -d '{"mono": true}'
```

```
# Flip polarity on input 1 (e.g. compare two close-mic'd guitar amps)
curl -X PUT http://127.0.0.1:56894/api/v1/input/analog/1/phase_invert \
  -H 'Content-Type: application/json' -d '{"phase_invert": true}'
```

Tracking-mode scene — dim mains, bring up cue, flip A/B

A single multi-field **POST** is the natural fit:

```
await fetch(`${BASE}/monitoring`, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    dim: true,
    ab_speaker_set: true,    // switch to the cue speaker pair
    mono: false,
  }),
});
```

Reference-track A/B against the DAW master

The reference-level recall is built into the API for exactly this:

```
# Set reference level (do this once at calibration time)
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/reference_level \
  -H 'Content-Type: application/json' -d '{"reference_level": -18.0}'

# A/B button: jump to reference for the reference track
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/volume \
  -H 'Content-Type: application/json' -d '{"volume": -18.0}'

# Visualize whether you are at reference (R0):
curl http://127.0.0.1:56894/api/v1/monitoring/is_at_reference_level
# → {"is_at_reference_level": true}
```

Subscribe to `/monitoring/is_at_reference_level` to keep an "AT REF" indicator live.

3. Professional studio

Most of the recipes in this section assume **AudioFuse 16Rig**. Studio-compatible variants are noted where they apply.

Session-profile recall — film 48k vs music 96k vs Atmos bed

Bundle a sample-rate change, a clock-source change, a speaker-set switch, and immersive flags into a single per-project recall.

```

PROFILES = {
    "Film 48k": {
        "clock": {"sample_rate": 48000, "preferred_source":
"internal"},
        "monitoring": {"ab_speaker_set": False, "mono": False,
"bass_management": False},
    },
    "Music 96k": {
        "clock": {"sample_rate": 96000, "preferred_source":
"internal"},
        "monitoring": {"ab_speaker_set": False, "mono": False},
    },
    "Atmos 48k bed": {
        "clock": {"sample_rate": 48000, "preferred_source":
"internal"},
        "monitoring": {"bass_management": True, "lfe_10db": True, "mono":
False},
    },
}

def recall(profile_name: str):
    p = PROFILES[profile_name]
    requests.post(f"{BASE}/clock", json=p["clock"]).raise_for_status()
    requests.post(f"{BASE}/monitoring",
json=p["monitoring"]).raise_for_status()

```

Validate first: before sending a sample-rate change, fetch `/clock/sample_rate_options` and ensure the value is supported. A wrong rate will be rejected with `400 Invalid Value`.

For an AF16Rig user, the same effect can also be persisted on the device via `/preset` — useful if the studio has a hardware controller that recalls slots without going through the API.

Speaker selection — Mains → Mids → NS10s → Atmos array

The shipping monitoring endpoints are an A/B pair (`ab_speaker_set`). For more than two speaker sets, drive the routing in the DAW or in your client and use `ab_speaker_set` as a sub-toggle. Show which set is active by reading the boolean back over SSE:

```

POST /api/v1/update
{"endpoints": ["/monitoring/ab_speaker_set"]}

```

Monitor-controller keys with state — dim, mute, mono, talkback

Each is a single-field write; the LED state follows the response and the SSE pushes:

```

curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/dim -d '{"dim":
true}' -H 'Content-Type: application/json'

```

```
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/mute -d '{"mute": true}' -H 'Content-Type: application/json'
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/mono -d '{"mono": true}' -H 'Content-Type: application/json'
```

For talkback specifically, the API does not expose a dedicated **talkback** endpoint — implement it as a DAW-side bus driven by your controller, or as a temporary gain change on the talkback mic input.

Surround → binaural fold-down for headphone check (16Rig)

Use the immersive solo / mute keys to listen to specific stems on headphones without rerouting the session:

```
# Solo only the L/R of the immersive bed
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/solo_left_right \
      -H 'Content-Type: application/json' -d '{"solo_left_right": true}'

# Or solo only surrounds
curl -X PUT http://127.0.0.1:56894/api/v1/monitoring/solo_surrounds \
      -H 'Content-Type: application/json' -d '{"solo_surrounds": true}'
```

These fields are mutually independent on the API; if your UI enforces "exactly one solo at a time", clear the others in the same **POST /monitoring** body.

4. AI-agent style natural-language control

The API is well suited to LLM-driven control surfaces — the endpoint paths are essentially function names, the JSON shapes are easy to generate, and the response codes give the agent rich error feedback. A typical pattern:

1. **Bootstrap the agent** with the endpoint inventory: feed it **GET /devices**, **GET /clock/sample_rate_options**, **GET /clock/source_options**, and the device-availability matrix from Endpoint Reference.
2. **Tool-call shape**: expose two tools — **get(path)** and **set(path, body, method='PUT')**. That is enough for the agent to satisfy any user instruction the API supports.
3. **Always read before write** for nudges or relative changes ("turn it up a bit"): **GET** the current value, compute, **PUT** the new value.

Worked example, "Turn on phantom on input 1":

```
# Pseudocode for the agent's tool calls
get("/devices")
# → {"devices": ["AFR16-12345"]}
set("/input/analog/1/48v", {"48v": True}, method="PUT")
# If the device returns 403 Invalid Request, the agent should reply:
```

```
# "I tried to enable phantom on input 1 but the device refused – usually  
# this happens when no microphone is connected to that input."
```

Worked example, "Drop the monitor by 3 dB":

```
cur = get("/monitoring/volume")["volume"]  
set("/monitoring/volume", {"volume": cur - 3.0})
```

Worked example, "Recall preset 4 if you can" (16Rig):

```
devices = get("/devices")["devices"]  
# Inspect the model from the serial prefix (e.g. AFR16-* = 16Rig);  
# or call /preset/names – a 404 tells the agent presets are unavailable.  
names = get("/preset/names")  
set("/preset", {"slot": 4})
```

Safety guardrails for agent-driven setups

- **Cap volume changes.** An agent that misinterprets *"a bit louder"* could slam the monitors. Clamp `/monitoring/volume` writes to a range you decide is safe.
- **Confirm destructive operations.** `POST /preset {"slot": N, "saved": true}` overwrites a slot — gate this behind explicit confirmation.
- **Subscribe to changes** so the agent's mental model stays in sync if the user touches Control Center directly.

Cross-cutting tips

- **Read before write for relative changes.** The API has no "increment by" semantics; clients must compute new absolute values themselves.
- **Coalesce snapshot recalls into a single `POST` per parent.** Fewer round trips = lower latency = less audible glitching during recall.
- **Subscribe to anything you display.** External changes (Control Center, other clients, hardware buttons on the device) will not be reflected if you only `GET` once.
- **Treat `200 Request ignored` as success.** It commonly fires for redundant writes during snapshot recall and is not an error.
- **Derive your endpoint surface from `/devices`.** Hard-coding 16Rig-only endpoints into a Studio session causes 404s that scare users.