

# HTTP & Websocket API For Integrators

## User Guide

---

This guide is a tutorial for external developers and show-automation builders: how to reach the DjMirror Pulse server, authenticate, make your first calls with curl , and follow the recipes that cover the common jobs (change a layer, apply a style, record a clip, hear about...

### DjMirror Pulse User Manual Series

Source: docs\_user\_manuals\HTTP\_WEBSOCKET\_API\_GUIDE.md

Revision 2026.10 • Release 1.0.743

# Table of Contents

---

Chapter 1: Connect And Make A First Call

---

Chapter 2: Authentication

---

Chapter 3: How Reads And Writes Work

---

Chapter 4: Recipes

---

Chapter 5: Live Updates: Websocket And Long-Poll

---

Chapter 6: Errors

---

Chapter 7: Where To Go Next

---

Audience: integrators wiring DjMirror Pulse into show control, companion apps, installers and scripts. End users driving the UI do not need this guide.

**Your first integration in five steps** — each is explained below:

1. Check the server answers: `getHealth` (§1).
2. Add your API key header if one is configured (§2).
3. Read the live state: `getConfig`, and remember `meta.revision` (§3).
4. Change only what you need with `PATCH` (§3, §4).
5. Open the WebSocket so you hear about changes made by the desktop UI or other clients (§5).

# Chapter 1: Connect And Make A First Call

---

The server runs by default on port **8080**; start it explicitly with `DjMirrorPulse.exe --server --port 8080` (or `"server": true, "port": 8080` in `config.json`). The WebSocket event server always listens on **HTTP port + 1** (default **8081**). All current routes live under `/api/v1`; the unversioned legacy routes still exist but new integrations should not use them.

```
curl http://127.0.0.1:8080/api/v1/health
```

```
{
  "ok": true,
  "data": { "status": "ok", "realtimeWsPort": 8081 },
  "error": null,
  "meta": { "apiVersion": "v1", "requestId": "3f9e2b1a-...", "timestamp": "2026-10-13T10:1"
}
```

Every `/api/v1` response has this envelope: `ok`, then `data` on success or `error` on failure, plus `meta`. `meta.revision` is a counter of the live configuration (see §3). The only exceptions are the long-poll subscription and the binary downloads (recordings, snapshots), which send raw bytes.

Then ask what this build can do — `getCapabilities` returns flags such as `supportsWebSocket` and, per feature (`spout`, `ndi`, `link`...), whether it is `available` and why not. **Branch on capabilities, not on version numbers.** Some features also depend on the license edition: `getLicense` tells you which are included, and a call that needs a higher edition answers `403 EDITION_REQUIRED`. The reference marks those operations with their minimum edition.

## Chapter 2: Authentication

---

- **No API key** (the default): authentication is off — every route and the WebSocket are open to anyone on the LAN who can reach the port.
- **API key set** (`DJMIRROR_API_TOKEN`, `"apiKey"` in `config.json`, or the Pairing panel): every route, legacy ones included, and the WebSocket need it, in either header: `X-API-Key: <key>` or `Authorization: Bearer <key>`. The token can also be the `deviceToken` of a phone paired by QR.
- **WebSocket with a key**: connect to `ws://<host>:<httpPort+1>/?token=<key or deviceToken>`.
- **Routes that need no key** because they authenticate themselves: the phone pairing flow (`postPairingClaim`, `getPairingInfo` and the mobile link routes).

```
curl -H "X-API-Key: $KEY" http://127.0.0.1:8080/api/v1/capabilities
```

In PowerShell use `curl.exe` (the plain `curl` is an alias of `Invoke-WebRequest`) or `Invoke-RestMethod`:

```
Invoke-RestMethod -Uri http://127.0.0.1:8080/api/v1/capabilities -Headers @{ "X-API-Key" =
```

## Chapter 3: How Reads And Writes Work

---

- **GET** never changes anything. **PUT** replaces a whole value; **PATCH** deep-merges only the keys you send.
- **getConfig** returns the full live configuration; **patchConfig** merges a partial object into it. Focused variants exist for the global settings, layers, the audio rules, artists and playlists — each is listed under its tag in the reference.
- **Validate, then store.** A **422** means the body was understood but wrong and **nothing was stored**; read **error.hint**, fix the payload and resend.
- **meta.revision** increases every time the live configuration changes in memory. Two reads with the same revision saw the same state. Writes that only touch files on disk (editing a saved style, DMX files) do not change it.

A tiny helper that unwraps the envelope and turns **ok: false** into an exception (JavaScript, browser or Node 18+):

```
async function api(path, options = {}) {
  const res = await fetch(`http://127.0.0.1:8080/api/v1${path}`, {
    ...options,
    headers: { "Content-Type": "application/json", "X-API-Key": KEY, ...options.headers },
  });
  const envelope = await res.json();
  if (!envelope.ok) throw new Error(`${envelope.error.code}: ${envelope.error.message}`);
  return envelope.data;
}
```

## Chapter 4: Recipes

Each recipe lists the operations it uses; click one for the exact fields and every possible answer. The examples use `curl`; add `-H "X-API-Key: $KEY"` when a key is set.

### Change a layer

Find the layer ids, then patch only the fields you want.

1. `getConfigLayerIds` lists every layer and sub-layer id (`videoBackgroundView`, `waveFormView`, `lightingView` ...).
2. `patchConfigLayersById` merges your fields into that layer.
3. `patchConfigLayersVisibility` is the quick show/hide switch.

```
# Brightness up on the video layer, nothing else touched
curl -X PATCH http://127.0.0.1:8080/api/v1/config/layers/videoBackgroundView \
  -H "Content-Type: application/json" -d '{"brightness": 1.2}'

# Hide the waveform
curl -X PATCH http://127.0.0.1:8080/api/v1/config/layers/visibility \
  -H "Content-Type: application/json" -d '{"layerId": "waveFormView", "enabled": false}'
```

### Apply a saved style

A style is a saved look (colours, media, layers). Applying one is a single, atomic server-side load.

1. `getVisualsStyles` lists the style names (`listStylesMetadata` adds whether each is installed).
2. `postVisualsStylesByNameApply` loads it onto the live configuration. An optional `overrides` object is deep-merged on top, handy for "this style, but brighter".

```
curl -X POST "http://127.0.0.1:8080/api/v1/visuals/styles/03-Rave-Neon-v1/apply" \
  -H "Content-Type: application/json" -d '{"overrides": {"elements": {"videoBackgroundView
```

Changing `currentStyle` through `patchConfig` only moves the "active style" pointer; `apply` is the call that actually loads the look.

### Run an action from a script

The same actions the schedule, the timecode cues and the MIDI/OSC/HID controllers fire — switch style, artist or playlist, trigger a DMX scene, mark the downbeat... — can be run on demand with `executeAction`.

```
curl -X POST http://127.0.0.1:8080/api/v1/actions/execute \
  -H "Content-Type: application/json" -d '{"action": "style", "target": "03-Rave-Neon-v1"}
```

To run the action of something you already stored, use `executeScheduleEntry` or `postTimecodeCuesEntriesByEntryIdExecute`.

## Record a clip and take a snapshot

1. `postRecordingStart` (optional `fps`, `maxDurationSec`, `fileName`), then `postRecordingStop`, which answers once the file is fully written.
2. `getRecordingStatus` tells whether it is running and whether the clip has audio.
3. `getRecordingList` and `getRecordingDownloadByFileName` fetch the file.
4. A still image: `postRecordingSnapshot`, then `getRecordingSnapshotDownloadByFileName`.

```
curl -X POST http://127.0.0.1:8080/api/v1/recording/start -H "Content-Type: application/json"
# ... later
curl -X POST http://127.0.0.1:8080/api/v1/recording/stop
curl -o clip.mp4 http://127.0.0.1:8080/api/v1/recording/download/<fileName>
```

The [Recording & Snapshots guide](#) covers the same workflow from the UI.

## Send the picture to OBS or Resolume

Spout shares the output as a texture on the same computer; NDI publishes it on the network. Both are checked and controlled the same way: look at the status, set the options, start, stop.

- Spout: `getSpoutStatus`, `patchSpoutConfig`, `postSpoutStart`, `postSpoutStop`.
- NDI: `getNdiStatus`, `patchNdiSettings`, `startNdiBridge`, `stopNdiBridge`.
- Several cropped outputs from one file: the Spout layout and NDI layout operations are under their tags in the reference; the [NDI & Spout guide](#) explains the layouts themselves.

```
curl http://127.0.0.1:8080/api/v1/spout/status
curl -X POST http://127.0.0.1:8080/api/v1/spout/start
```

Starting a Spout or NDI output replaces what a running OBS or Resolume receives: do it deliberately.

## Light the rig

DMX fixtures, scenes and patterns are read and written with `GET` / `PUT` / `PATCH` under the **DMX** tag (every write is validated and stores nothing on a `422`). For live control during a show, use the runtime operations: `postDmxRuntimeBlackout` for a blackout, `postDmxRuntimeSilence` for the fade and for forcing silence while you rehearse, and `getDmxRuntimeStatus` to read the live state.

```
curl -X POST http://127.0.0.1:8080/api/v1/dmx/runtime/blackout -H "Content-Type: application/json"
```

## Automate with the schedule or timecode

Time-of-day automation is the **Schedule** tag (entries, import/export and a simulated clock to rehearse it); timecode cues are the **Timecode** tag. Both store the same kind of action as `executeAction`. See the [Programming Schedule](#) and [Timecode Shows](#) guides for how they behave.

## Pair a phone

The phone-pairing flow (QR session, claim, approval, device list) is the **Pairing** tag, and [Mobile Remote](#) explains it from the user's side. A paired phone gets a one-time `deviceToken` that works like an API key.

## Chapter 5: Live Updates: Websocket And Long-Poll

| Channel          | Endpoint                                           | Latency | Notes                                                            |
|------------------|----------------------------------------------------|---------|------------------------------------------------------------------|
| WebSocket events | <code>ws://&lt;host&gt;:&lt;httpPort+1&gt;/</code> | < 1 s   | Everything below; needs port HTTP+1 open                         |
| HTTP long-poll   | <code>getConfigSubscribe</code>                    | seconds | Config changes only; works through any firewall that allows HTTP |

Both announce changes made from the desktop UI as well as by other API clients. The long-poll answers **one** event and closes, so you re-subscribe each time; it never carries the DMX, recording, Spout or NDI events, which are WebSocket-only. The full list of events is in the reference, under **WebSocket events** in its introduction.

Events arrive in two shapes: configuration changes, and named events.

```
{ "type": "event", "eventType": "config.changed", "resource": "config", "action": "updated",
  "data": { "...": "only the changed keys" }, "requestId": "...", "timestamp": "..." }

{ "type": "event", "eventType": "custom.event", "resource": "event", "action": "triggered",
  "name": "spout.started", "data": {}, "requestId": "...", "timestamp": "..." }
```

A minimal listener. Omit `?token=` when no API key is set; the port is `realtimeWsPort` from `getHealth`.

```
function connect() {
  const ws = new WebSocket(`ws://127.0.0.1:8081/?token=${KEY}`);
  ws.onmessage = (m) => {
    const ev = JSON.parse(m.data);
    if (ev.eventType === "config.changed") console.log("config changed:", ev.data);
    else if (ev.eventType === "custom.event") console.log(ev.name, ev.data);
  };
  ws.onclose = () => setTimeout(connect, 5000); // reconnect every ~5 s
}
connect();
```

Take the `getConfig` snapshot **after** the socket is open, so a change between the two cannot be missed. The server sends a `ping` event every 30 seconds to keep idle connections alive.

## Chapter 6: Errors

Every failure has the same shape, with the HTTP status to match; `hint` and `details` appear only when the endpoint has something to add (a parser message, for example).

```
{ "ok": false, "data": null, "error": { "code": "INVALID_PAYLOAD", "message": "...", "hint": ... }
```

| HTTP | Meaning                                            | Typical <code>error.code</code>                                   | What to do                                                        |
|------|----------------------------------------------------|-------------------------------------------------------------------|-------------------------------------------------------------------|
| 400  | Body is not valid JSON, or a parameter is missing  | <code>INVALID_JSON</code> , <code>INVALID_PARAMS</code>           | Fix the request                                                   |
| 401  | Missing or wrong API key                           | <code>UNAUTHORIZED</code> , <code>DEVICE_REVOKED</code>           | Send <code>X-API-Key</code> or <code>Authorization: Bearer</code> |
| 403  | License or phone limit                             | <code>EDITION_REQUIRED</code> , <code>MOBILE_LIMIT_REACHED</code> | Activate or upgrade; unpair stale phones                          |
| 404  | Unknown id, name or file                           | <code>NOT_FOUND</code>                                            | List again and use an id that exists                              |
| 409  | Wrong state for the call                           | <code>ALREADY_RUNNING</code> , <code>NOT_RUNNING</code>           | Read the status first                                             |
| 422  | Validation failed — <b>nothing was stored</b>      | <code>INVALID_PAYLOAD</code> , <code>EMPTY_PATCH</code>           | Read <code>error.hint</code> , fix, resend                        |
| 503  | Feature not available on this build or not running | <code>NOT_AVAILABLE</code>                                        | Check <code>capabilities</code>                                   |

Each operation in the reference lists the codes it can really answer.

Good practice: prefer `config.json` over command-line key flags on shared machines (process lists leak arguments); treat `422` as "my request was wrong", not as a transient fault; and never poll faster than you need — subscribe instead.

## Chapter 7: Where To Go Next

---

- **Every route, field and status code:** the [API reference](#), generated from `openapi-api-v1.yaml`. Use that file to generate a client in your language.
- **The same control for AI assistants:** [MCP Server for Users](#).
- **Something not working:** [Troubleshooting](#).